Extending Zero-Knowledge PCPs Beyond NP

A Thesis
Presented to
The Established Interdisciplinary Committee for
Mathematics and Computer Science
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Patrick Norton

March 31, 2025

Approved for the Committee
(Mathematics and Computer Science)

Zajj Daugherty

Adam Groce

Contents

List of Algorithms

Introduction

The P vs NP problem is perhaps the most important open problem in complexity theory.

Chapter 1

Preliminaries

1.1 Turing machines

Central to our definitions of complexity is that of a Turing machine. This is the most common mathematical model of a computer, and is the jumping-off point for mant variants. There are many ways to think of a Turing machine, but the most common is that of a small machine that can read and write to an arbitrarily-long "tape" according to some finite set of rules. We give a more formal definition below, and then we will attempt to take this definition into a more manageable form.

TODO:

Figure 1.1: A Turing machine

Definition 1.1.1 ([31, Def. 3.1])**.** A *Turing machine* (abbreviated TM) is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ where Q, Σ, and Γ are all finite sets and

1. Q is the set of *states*,
2. Σ is the *input alphabet*,
3. Γ is the *tape alphabet*,
4. $\delta\colon Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the *transition function*,
5. $q_0 \in Q$ is the *start state*,
6. $q_a \in Q$ is the *accept state*,
7. $q_r \in Q$ is the *reject state*, with $q_a \neq q_r$.

While we have this formalism here as a useful reference, even here we will most frequently refer to Turing machines in a more intuitionisitc form. There are several ways we will think about Turing machines.

The first way to think about a Turing machine is as a little computing box with a tape. We let the box read and write to the tape, and each step it can move the tape one space in either direction. At some point, the machine can decide it is done, in which case we say it "halts"; however it does not necessarily need to halt. For this paper, we will only think about machines that *do* halt, and in particular we will care about how many it takes us to get there. Further, we will use this informalism as a base from which we can define our Turing machine variants intuitively, without needing to deal with the (potentially extremely convoluted) formalism.

Another way we think about a Turing machine is as an algorithm. Perhaps the foundational paper of modern computer science theory, the *Church-Turing thesis* [33], states that any actually-computable algorithm has an equivalent Turing machine, and vice versa. We will use this fact liberally; in many cases we will simply describe an algorithm and not deal with putting it into the context of a Turing machine. If we have explained the algorithm well enough that a reader can execute it (as we endeavor to do), then we know a Turing machine must exist.

TODO:

Figure 1.2: A nondeterministic Turing machine

Definition 1.1.2. A *nondeterministic Turing machine* is

Definition 1.1.3. A *multitape Turing machine* is

Definition 1.1.4. A *probabilistic Turing machine* is

1.2 Complexity classes

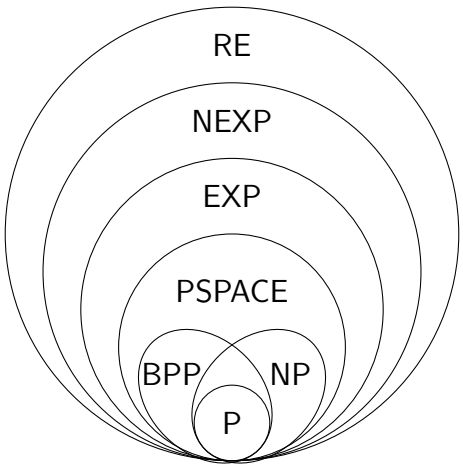

Figure 1.3: The relationship between the complexity classes for this paper

Complexity classes are the main way we think about the hardness of problems in computer science. A complexity class is a collection of languages that all share a common level of difficulty.

We start with a relatively straightforward example of a complexity class: the class of languages that a Turing machine can recognize. First, we need to define what recognition is in order to make a complexity class out of it.

Definition 1.2.1 ([31, Def. 3.2])**.** A language L is *recognized* by a Turing machine M if for all strings $s \in L$, M halts in the accept state when given s as input.

Now, since our complexity classes are about *languages*, we naturally wish to extend our notion of recognition to a statistic on languages.

Definition 1.2.2. A language L is *Turing-recognizable* (frequently just *recognizable*) if it is recognized by some Turing machine.

Now that we have a property of languages, it is straightforward for us to turn it into a complexity class.

Definition 1.2.3. The class RE is the class of all Turing-recognizable languages.

For most other classes, we want our Turing machines to halt on *all* inputs, not just those in the class. From a practical perspective, this is useful because it tells us that we can be certain about whether any given string is in the given language. From here on, we will generally care about how much of some resource our machines take when making their decision, as opposed to whether or not they can.

1.2.1 Time complexity

The most intuitive (and most important) notion of complexity is that of time complexity. Time complexity is the answer of the question of how long it takes to solve a problem. We begin with an abstract base for our time classes, and will then introduce some specific ones that we care about.

Definition 1.2.4 ([2, Def. 1.19])**.** Let $f\colon \mathbb{N} \to \mathbb{N}$ be a function. The class $\mathsf{DTIME}(f(n))$ is the class of all problems computable by a deterministic Turing machine in $O(f(n))$ steps for some constant $c > 0$.

While DTIME is a useful base to start from, it is rare that we deal with DTIME classes directly.

Definition 1.2.5 ([2, Def. 1.20])**.** The complexity class P is the class

$$\mathsf{P} = \bigcup_{c>0} \mathsf{DTIME}(n^c).$$

The class P is perhaps the most important complexity class. Mathematically, we care about P because it is closed under composition: a polynomial-time algorithm iterated a polynomial number of times is still in P. Further, P turns out to generally

be invariant under change of (deterministic) computation model, which allows us to reason about P problems easily without needing to resort to the formal definition of a Turing machine. More philosophically, P generally represents the set of "efficient" algorithms in the real world.[1]

Example 1.2.5.1. The language

$$\{(p, x, y) \mid p \text{ a polynomial and } p(x) = y\}$$

is in P. We can calculate whether a string is in this language by calculating $p(x)$ (which we can do efficiently), and then comparing it to y.

As we have defined P in terms of DTIME, the question arises of whether there is an equivalent in terms of NTIME. Naturally, there is, and we call it NP.

Definition 1.2.6 ([31, Cor. 7.22])**.** The complexity class NP is the class

$$\mathsf{NP} = \bigcup_{c>0} \mathsf{NTIME}(n^c).$$

While this definition demonstrates how NP is similar to P, there are other equivalent ones that we can use. In particular, we very often like to think of NP in terms of deterministic *verifiers*. Since nondeterministic machines do not exist in real life, this definition gives a practical meaning to NP.

Example 1.2.6.1. The language SAT is the language of Boolean formulas with at least one solution. SAT is in NP: we can nondeterministically pick a potential solution and then evaluate our formula (which can be done efficiently); there will be an accepting path if and only if a solution to the formula exists.

Theorem 1.2.7 ([31, Def. 7.19])**.** NP *is exactly the class of all languages verifiable by a* P*-time Turing machine.*

Example 1.2.7.1. The language SAT we defined in Example 1.2.6.1 can be verified efficiently, where the certificate is an accepting set of variables. Since we can evaluate a Boolean formula efficiently, if we already have an accepting set of variables we can therefore verify it in P.

The next step up from polynomial complexities is that of exponential complexities. For these, instead of having the classes bounded above by a polynomial, we have the classes bounded above by 2 to the power of a polynomial. While we use 2 as the base, the value of the base turns out not to matter since for any $a, b > 1$,

$$a^{n^c} = b^{n^c \log_b(a)} \in O\left(b^{n^{c+1}}\right). \tag{1.1}$$

Definition 1.2.8 ([2, §2.6.2])**.** The complexity class EXP is the class

$$\mathsf{EXP} = \bigcup_{c>0} \mathsf{DTIME}\left(2^{n^k}\right).$$

[1] It is worth mentioning that this is a *mathematical* efficiency—there are plenty of algorithms in P that a real-world computer scientist would never dare to call efficient.

Definition 1.2.9 ([2, §2.6.2])**.** The complexity class NEXP is the class

$$\mathsf{NEXP} = \bigcup_{c>0} \mathsf{NTIME}\left(2^{n^k}\right).$$

It is immediate that $\mathsf{P} \subseteq \mathsf{EXP}$ and $\mathsf{NP} \subseteq \mathsf{NEXP}$ (since the exponential classes allow the use of more of the same resource). Of slightly less-trivial interest is the relationship between NP and NEXP.

Theorem 1.2.10. $\mathsf{NP} \subseteq \mathsf{EXP}$.

Proof. If a nondeterministic machine solves a problem in $p(n)$ steps, it follows that the total number of branches is less than $a^{p(n)}$, where a is the maximum number of branches for a node within the machine. Hence, we can simulate the machine deterministically by simply enumerating every branch, giving us a total computation time of $p(n)a^{p(n)}$, which is in $O(2^{q(n)})$ for some other polynomial $q(n)$. Hence any NP problem is in EXP. □

It is perhaps illustrative to see an example of a problem in NEXP.

Definition 1.2.11 ([11, Def. 14.1])**.** The *oracle 3-satisfiability problem*, denoted O3SAT, is the language of all triplets (r, s, B), where $r, s \in \mathbb{N}^+$ and $B\colon \{0,1\}^{r+3s+3} \to \{0,1\}$ a boolean function, such that there exists a boolean function $A\colon \{0,1\}^s \to \{0,1\}$ having the property that for all $z \in \{0,1\}^r$ and $b_1, b_2, b_3 \in \{0,1\}^s$,

$$B(z, b_1, b_2, b_3, A(b_1), A(b_2), A(b_3)) = 1.$$

Theorem 1.2.12. $\mathsf{O3SAT} \in \mathsf{NEXP}$.

Proof. We present the following non-deterministic algorithm to determine if $(r, s, B) \in$ O3SAT:

Input: A triplet (r, s, B)
Output: Whether or not $(r, s, B) \in$ O3SAT

```
1  Nondeterministically choose a function A: {0,1}^s → {0,1};
2  for z ∈ {0,1}^r do
3  |  for b1, b2, b3 ∈ {0,1}^s do
4  |  |  Compute B(z, b1, b2, b3, A(b1), A(b2), A(b3));
5  |  |  if the above is not 1 then
6  |  |  |  return 0;
7  |  |  end
8  |  end
9  end
10 return 1;
```

Algorithm 1.1: A NEXP-time algorithm for determining O3SAT

First, we need to show that Algorithm 1.1 is in NEXP. Nondeterministically choosing a function from $\{0,1\}^s$ can be done in time 2^s; and the two loops will run a total of $2^r 2^s$ times, respectively. Computation of a function can be done in polynomial

time relative to its length; hence the runtime of this function is in exponential time relative to $r + s$.

One might be tempted to think that since we are given a function B as input, that our function A can be no longer than $\mathsf{poly}(|B|)$, but this is not necessarily true. We are given B in 3SAT form, and thus there are expressions of B that are polynomial with respect to $r + s$. Despite this, there are polynomial-length B instances whose A is *not* polynomial in length (since that is an arbitrary function); since runtime complexity is about the worst case there thus exist inputs that cannot be computed in polynomial time relative to their length.

TODO: □

1.2.2 Space complexity

In addition to time complexity, the an additional notion of complexity is that of space complexity. Space complexity is the question of how much space on its memory tape a machine needs in order to compute a problem. In many ways, our definitions of space complexity are analagous to those for time complexity that we have already defined. In particular, DSPACE will correspond nicely to DTIME, and NSPACE to NTIME.

Definition 1.2.13 ([2, Def. 4.1])**.** Let $f\colon \mathbb{N} \to \mathbb{N}$ be a function. A language L is in $\mathsf{DSPACE}(f(n))$ if there exists a deterministic Turing machine M such that the number of locations on the tape that are non-blank at some point during the execution of M is in $O(f(n))$.

In the same way as we have defined DSPACE for deterministic machines, we now need to define NSPACE for nondeterministic machines.

Definition 1.2.14 ([2, Def. 4.1])**.** Let $f\colon \mathbb{N} \to \mathbb{N}$ be a function. A language L is in $\mathsf{NSPACE}(f(n))$ if there exists a nondeterministic Turing machine M such that the number of locations on the tape that are non-blank at some point during the execution of M is in $O(f(n))$.

Analagously to P and NP, our two main classes of space complexity are PSPACE and NPSPACE.

Definition 1.2.15 ([2, Def. 4.5])**.** The complexity class PSPACE is the class

$$\mathsf{PSPACE} = \bigcup_{c>0} \mathsf{DSPACE}(n^c).$$

Definition 1.2.16 ([2, Def. 4.5])**.** The complexity class NPSPACE is the class

$$\mathsf{NPSPACE} = \bigcup_{c>0} \mathsf{NSPACE}(n^c).$$

Unlike with P and NP, the relationship between PSPACE and NPSPACE is well known. Due to the complexity of the proof of the theorem, we will not prove it here, as it is mostly not relevant to what we will be doing.

Theorem 1.2.17 (Savitch's theorem; [29])**.** PSPACE = NPSPACE.

Upon seeing this, one might ask why it is that we believe P ≠ NP if we know that PSPACE = NPSPACE, given they are defined analogously. The answer to this question boils down to the fact that we are able to reuse space, while we are not able to reuse time. Space on the tape that is no longer needed can be overwritten, while time that is no longer needed is gone forever.

Since PSPACE and NPSPACE are equal classes, it is relatively rare to see NPSPACE referred to. Here, we will only refer to it when it makes a class relationship clearer; most frequently when comparing NPSPACE to some other nondeterministic class.

Example 1.2.17.1. The language

$$\{(x, y) \mid x, y \text{ regexes that accept the same set of strings}\}$$

is in PSPACE.

1.2.3 Completeness

Even within a complexity class, not all problems are created equal. The notion of *completeness* gives us a mathematically-rigorous way to talk about which problems in a class are the hardest. Since putting upper bounds on hard problems naturally puts those same bounds on any easier problems, complete problems can be useful in reasoning about the relationship between complexity classes.

Definition 1.2.18 ([31, Def. 7.29])**.** A language A is *polynomial-time reducible* to a language B if a polynomial-time computable function $f\colon \Sigma^* \to \Sigma^*$ exists such that for all $w \in \Sigma^*$, $w \in A$ if and only if $f(w) \in B$.

Polynomial-time reductions are important because they give us a way to say that A is *no harder* than B. In particular, if we have an algorithm M that determines B, we can construct the following algorithm that determines A with only a polynomial amount of additional work:

Input: A string $w \in \Sigma^*$
Output: Whether $w \in A$
1 Compute $f(w)$;
2 Use M to check whether $f(w) \in B$;
3 **return** *the result of* M;

Algorithm 1.2: An algorithm to reduce A to B

Definition 1.2.19 ([31, Def. 7.34])**.** A language L is NP-complete if $L \in$ NP and every $A \in$ NP is polynomial-time reducible to L.

This is a practical use of our polynomial-time reductions: since an NP-complete language has a reduction from every other language in NP, it follows that it is *at least as hard* as any other language in NP. Of particular interest to complexity theorists is the fact that P = NP if and only if *any* NP-complete language is in P.

Example 1.2.19.1. A famous result of Cook [13], also proved around the same time by Levin and thus called the *Cook-Levin theorem*, is that the SAT problem we defined earlier in Example 1.2.6.1 is NP-complete.

The notion of completeness is very important to complexity theorists. Since these are the "hardest" problems in NP, this means that if we can do anything interesting to an NP-complete problem, we can leverage these reductions to do that interesting thing to *any* other problem in NP with only a little (i.e. polynomial) more effort. This will come in especially handy when we want to prove that complexity classes are equal or that NP is a subset of some other complexity class—since most complexity classes allow for things to change polynomially, we only need to prove that a single NP-complete element is in the other class for the subset relation to follow.

Along with completeness for NP, we have a notion of completeness for NEXP. While you might expect that the reducibility constraints might loosen (i.e. allow more complex reductions) since NEXP is more complex for NP, but this turns out not to be the case. In particular, while it might initially seem logical to allow for EXP-reductions, it turns out that NEXP is not closed under EXP-reductions, which makes a notion of completeness challenging. Despite this, we can still learn interesting things about NEXP by studing completeness under polynomial reductions.

Definition 1.2.20. A language L is NEXP-complete if $L \in \mathsf{NEXP}$ and every $A \in \mathsf{NEXP}$ is polynomial-time reducible to NEXP.

NEXP-completeness has many of the same nice properties of NP-completeness. Of particular interest to us will again be the ease with which NEXP-completeness allows us to determine subset relations, simply by proving the inclusion of a single complete language.

Theorem 1.2.21 ([5, Proposition 4.2])**.** *The language* O3SAT *(as defined in Definition 1.2.11) is* NEXP*-complete.*

Proof. We demonstrated in Theorem 1.2.12 that $\mathsf{O3SAT} \in \mathsf{NEXP}$, so all that remains is to prove that reductions exist for every NEXP language. Let $L \in \mathsf{NEXP}$, and let $x \in \{0,1\}^n$. We aim to construct an algorithm in $\mathsf{P}^{\mathsf{O3SAT}}$ that computes L.

TODO: □

Just as we have NP-completeness and NEXP-completeness for time complexity, we also have notions of completeness for space complexity. Since PSPACE = NPSPACE, instead of calling the class NPSPACE-complete, we call it PSPACE-complete.

Definition 1.2.22 ([31, Def. 8.8])**.** A language L is PSPACE-complete if $L \in \mathsf{PSPACE}$ and every $A \in \mathsf{PSPACE}$ is polynomial-time reducible to NP.

While this definition is mostly analagous to that of NP-completeness, one might wonder why we use a time complexity for our reduction when PSPACE is a space-complexity class. This is because if we were to use space complexity, we would want to use PSPACE-reductions, but that would make every language in PSPACE trivially PSPACE-complete. Since that is not a useful definition, we instead restrict ourselves to polynomial-time reductions.

Example 1.2.22.1. A result of Stockmeyer and Meyer [32] is that the language we defined in Example 1.2.17.1 is PSPACE-complete.

1.2.4 Randomized complexity

TODO:

Definition 1.2.23. A language L is in BPP if there exists a probabilistic Turing machine M such that

1. M runs in polynomial time,
2. for all $x \in L$, M accepts x with probability at least $2/3$,
3. for all $x \notin L$, M rejects x with probability at least $2/3$.

Theorem 1.2.24. $\mathsf{P} \subseteq \mathsf{BPP}$.

Theorem 1.2.25. $\mathsf{BPP} \subseteq \mathsf{PSPACE}$.

1.2.5 Counting complexity

So far, all of the problems we have seen are boolean problems, where the answer is either yes or no. There exist many interesting questions, however, where we would like more than two distinct answers. In particular, there are lots of problems where the most interesting answer is a *count* of something: in particular, questions of the type "How many objects are there such that some condition is true?" will be of interest to us.

Definition 1.2.26. A *counting problem* is a function from $\{0,1\}^*$ to $\mathbb{N}$.

Now that we have this definition, we should define a complexity class of function problems. The simplest complexity class for function problems is #P, the set of function problems that can be computed in polynomial time.

Definition 1.2.27 ([2, Def. 9.2])**.** The class #P is the class of functions $f\colon \{0,1\}^* \to \mathbb{N}$ such that there exists a polynomial $p\colon \mathbb{N} \to \mathbb{N}$ and a polynomial-time Turing machine M such that for every $x \in \{0,1\}^*$,

$$f(x) = \left|\left\{y \in \{0,1\}^{p(|x|)} \;\middle|\; M(x,y) = 1\right\}\right|. \tag{1.2}$$

As mentioned earlier, questions of the type "How many objects are there such that some condition is true?" are of the most interest of us, and here we actually define #P in terms of these problems. Equation (1.2) simply says that $f(x)$ needs to be equal to the number of inputs y on which some polynomial Turing machine accepts when given both x and y. In this case, the Turing machine M provides our condition, and a function problem is in #P exactly when we can compute this condition efficiently.

TODO: Introduce #P-completeness

Definition 1.2.28. A language is #P-complete if TODO: Do I want to use the definition with FP?

Definition 1.2.29. The function #SAT is the function that, given a Boolean formula ϕ, returns the number of distinct assignments such that ϕ is true.

Theorem 1.2.30. *#SAT is #P-complete.*

1.3 Polynomials

TODO:

Definition 1.3.1 ([25])**.** Let P be a mathematical statement. The function $[P]$ is the the function

$$[P] = \begin{cases} 1 & P \text{ is true} \\ 0 & \text{otherwise.} \end{cases} \tag{1.3}$$

This is called the *Iverson bracket*, after its inventor Kenneth Iverson, who originally included it in the programming language APL[2] [20, p. 11].

Example 1.3.1.1. Using the Iverson bracket, the Kronecker delta function can be defined as

$$\delta_{ij} = [i = j].$$

Much of our work will deal with multivariate polynomials. For a given field $\mathbb{F}$, we will denote the set of m-variable polynomials over $\mathbb{F}$ with $\mathbb{F}[x_{1,\dots,m}]$.

Definition 1.3.2 ([1, p. 8])**.** The *multidegree* of a multivariate polynomial p, written $\text{mdeg}(d)$, is the maximum degree of any variable x_i of p.

It is worth noting that for monovariate polynomials, multidegree and degree coincide. The difference between multidegree and degree is subtle, but important. We shall illustrate the difference with a simple example.

Example 1.3.2.1. Consider the polynomial $x_1^2x_2+x_2^2$. The multidegree of this polynomial is 2, while its degree is 3.

We denote by $\mathbb{F}[x_{1,\dots,m}^{\leq d}]$ the subset of $\mathbb{F}[x_{1,\dots,m}]$ of polynomials with multidegree at most d. We also need two special cases of these polynomials, which we will want to quickly be able to reference throughout the paper. Similarly, if we want to refer to polynomials with degree at most d, we will write $\mathbb{F}^{\leq d}[x_{1,\dots,m}]$.

Definition 1.3.3 ([1, p. 8])**.** A polynomial is *multilinear* if it has multidegree at most 1. Similarly, a polynomial is *multiquadratic* if it has multidegree at most 2.

Example 1.3.3.1. The polynomial $x_1x_2+4x_2x_3+x_1x_2x_3$ is multilinear. The polynomial $x_1^2x_2x_3 - 2x_1x_3 + 3x_2^2$ is multiquadratic.

[2]The original notation used parentheses, but square brackets are much less ambiguous, so that has become the standard and what we will use here.

From here, we need to define the notion of an *extension polynomial*. This gives the ability to take an arbitrary multivariate function defined on a subset of a field and extend it to be a multivariate polynomial over the *whole* field.

Definition 1.3.4 ([1, p. 8])**.** Let $\mathbb{F}$ be a finite field, $H \subseteq \mathbb{F}$, $m \in \mathbb{N}$ a number, and $f\colon H^m \to \mathbb{F}$ be a function. An *extension polynomial* of f is any polynomial $f' \in \mathbb{F}[x_{1,\dots,m}]$ such that $f(h) = f'(h)$ for all $h \in H$.

Example 1.3.4.1. Define $H = \{0,1\}^3 \subseteq \mathbb{R}^3$. Further define

$$\begin{aligned} f\colon H^3 &\to \mathbb{R} \\ (a,b,c) &\mapsto a \oplus b \oplus c, \end{aligned}$$

where $\oplus$ is the xor function; equivalently addition mod 2. Then an extension polynomial of f is the function

$$f'(x,y,z) = xyz - (x-y)(y-z)(z-x).$$

A second extension polynomial of f is the function

$$f''(x,y,z) = x + y + z - 2xy - 2yz - 2xz + 4xyz.$$

There are (at least) two important things to be gleaned from this example. First, extension polynomials are not unique: f' and f'' are not equal to each other (they are not even of the same multidegree). Second, f'' is in fact multilinear, which might be a somewhat lower multidegree than expected given we need to interpolate 8 different points. It turns out that this is not particularly unusual: while our choice of H is particularly nice, functions from this particular H still happen to be quite nice in general. Even for less-nice values of H, extension polynomials need only to be of a surprisingly low multidegree. Since polynomials of lower degree are generally easier to compute, we would like to see exactly what these low-degree extension polynomials look like and how they work.

Definition 1.3.5 ([11, §5.1])**.** Let $\mathbb{F}$ be a finite field, $H \subseteq \mathbb{F}$, $m \in \mathbb{N}$ a number, and $f\colon H^m \to \mathbb{F}$ be a function. A *low-degree extension* $\tilde{f}$ of f is an extension of f with multidegree at most $|H| - 1$.

It turns out that this is the minimum possible degree of any extension polynomial. Further, it turns out that for any f, there is a *unique* low-degree extension. Neither of these statements are particularly important for our further work, so we will not endeavor to prove them here. Something of practical use to us is an explicit formula for the low-degree extension, which we shall now calculate.

Theorem 1.3.6 ([11, §5.1])**.** *Let $\mathbb{F}$ be a finite field, $H \subseteq \mathbb{F}$, $m \in \mathbb{N}$ a number, and $f\colon H^m \to \mathbb{F}$. Then a low-degree extension $\tilde{f}$ of f is the function*

$$\tilde{f}(x) = \sum_{\beta \in H^m} \delta_\beta(x) f(\beta), \tag{1.4}$$

where δ is the polynomial

$$\delta_x(y) = \prod_{i=1}^{m} \left(\sum_{\omega \in H} \left(\prod_{\gamma \in H \setminus \{\omega\}} \frac{(x_i - \gamma)(y_i - \gamma)}{(\omega - \gamma)^2} \right) \right). \tag{1.5}$$

Proof. First, we must show $\tilde{f}$ has multidegree $|H| - 1$. First, note that $\tilde{f}$ is a linear combination of some $\delta_x$es; hence asking about the multidegree of $\tilde{f}$ is really just asking about the multidegree of δ_x. Looking at δ_x, the innermost product has $|H| - 1$ terms, each with the same y_i; thus those terms have multidegree $|H| - 1$. Summing terms preserves their multidegree, and the outer product iterates over the variables, thus it preserves multidegree as well. Thus, δ_x has multidegree $|H| - 1$.

To understand why $\tilde{f}(x)$ agrees with $f(x)$ on H, we first should look at $\delta_\beta(x)$. In particular, for all $x, y \in H^m$,

$$\delta_y(x) = [x = y] = \delta_{xy}. \tag{1.6}$$

This can be shown through some algebra which we have worked through in full detail in Appendix A. This is the reason why we have named the polynomial in Equation (1.5) as we have; it functions as the Kronecker delta function over the set H^m.

Taking the above statement, we get that for all $x \in H^m$, the only nonzero term of $\tilde{f}(x)$ is the term where $\beta = x$; thus $\tilde{f}(x) = f(x)$. Hence, $\tilde{f}$ is a low-degree extension of f. □

Of particular interest to us will be the case of low-degree extensions where $H = \{0, 1\}$. Since every field contains both 0 and 1, this will allow us to construct a set consisting of an extension for *every* field. Further, since $|H| = 2$ here, it means our low-degree extensions will be multilinear. Not only do we thus constrain our polynomial to have a very low multidegree, the δ function also dramatically simplifies in this case, which makes it much easier to reason about.

Corollary 1.3.7 ([1, §4.1])**.** *Let $\mathbb{F}$ be a finite field, $m \in \mathbb{N}$ a number, and $f \colon \{0,1\}^m \to \mathbb{F}$. Then*

$$\tilde{f}(x) = \sum_{\beta \in \{0,1\}^m} \delta_\beta(x) f(\beta) \tag{1.7}$$

is a low-degree extension of f, where δ is the polynomial

$$\delta_y(x) = \left(\prod_{i : y_i = 1} x_i \right) \left(\prod_{i : y_i = 0} (1 - x_i) \right). \tag{1.8}$$

Note that in the product bound $i : y_i = 1$, we mean the product over all numbers i such that $y_i = 1$.

As we can see, the form of δ in Equation (1.8) is much more manageable than the form in Equation (1.5), and it is perhaps more immediately apparent here why δ has the property it does. Further, since this equation has no division, it turns out that it is valid for arbitrary (non-trivial) rings, while the more complex equation is only

valid for fields. We show the algebra that brings us from the first to the second in Appendix A.

The form of δ_y defined in Equation (1.8) has further use to us than just being simpler. In particular, these δ_y form a basis of multilinear polynomials (and hence a generating set for the ring of all polynomials). This is a particularly useful basis because it allows us to reason about multilinear polynomials based solely on their outcomes on the Boolean cube.[3]

Theorem 1.3.8 ([1, §4.1])**.** *For any field $\mathbb{F}$, the set $\{\delta_x \mid x \in \{0,1\}^n\}$ forms a basis for the vector space of multilinear polynomials $\mathbb{F}^n \to \mathbb{F}$.*

Proof. Since $\delta_y(x) = 0$ for all $y \neq x \in \{0,1\}^n$, it follows that the only way to get

$$\sum_{y \in \{0,1\}^n} a_y \delta_y = 0 \tag{1.9}$$

is to have each $a_y = 0$. Hence the set of δ_x is linearly independent. Further, the vector space of multilinear polynomials has 2^n dimensions; since there are 2^n distinct δ_x polynomials, it follows that they form a basis. □

Now, we can use this fact to prove some cases where our low-degree extensions turn out to have a particularly low degree. Unfortunately, these do have a lot of qualifiers to them, but they will be useful in later theorems (in particular Lemma 1.3.10).

Theorem 1.3.9 ([1, Theorem 4.3])**.** *Let $\mathbb{F}$ be a field and $Y \subseteq \mathbb{F}^n$ be a set of t points $y_1, \ldots, y_t$. Then for at least $2^n - t$ Boolean points $w \in \{0,1\}^n$, there exists a multiquadratic extension polynomial $p \colon \mathbb{F}^n \to \mathbb{F}$ such that*

1. *$p(y_i) = 0$ for all $i \in [t]$,*
2. *$p(w) = 1$,*
3. *$p(z) = 0$ for all Boolean $z \neq w$.*

Proof. TODO: □

Lemma 1.3.10 ([1, Lemma 4.5])**.** *Let $\mathcal{F}$ be a collection of fields. Let $f \colon \{0,1\}^n \to \{0,1\}$ be a Boolean function, and for every $\mathbb{F} \in \mathcal{F}$, let $p_{\mathbb{F}} \colon \mathbb{F}^n \to \mathbb{F}$ be a multiquadratic polynomial over $\mathbb{F}$ extending f. Also let $\mathcal{Y}_{\mathbb{F}} \in \mathbb{F}^n$ for each $\mathbb{F} \in \mathcal{F}$, and define $t = \sum_{\mathbb{F} \in \mathcal{F}} |\mathcal{Y}_{\mathbb{F}}|$.*

Then, there exists a subset $B \subseteq \{0,1\}^n$, with $|B| \leq t$, such that for all Boolean functions $f' \colon \{0,1\}^n \to \{0,1\}$ that agree with f on B, there exist multiquadratic polynomials $p'_{\mathbb{F}} \colon \mathbb{F}_n \to \mathbb{F}$ (one for each $\mathbb{F} \in \mathcal{F}$) such that

1. *$p'_{\mathbb{F}}$ extends f', and*
2. *$p'_{\mathbb{F}}(y) = p_{\mathbb{F}}(y)$ for all $y \in \mathcal{Y}_{\mathbb{F}}$.*

[3] As an aside, this fact provides a relatively slick proof of the special case of our unproven statement earlier that low-degree extensions are both of minimal degree and unique.

Proof. Call $z \in \{0,1\}^n$ *good* if for every $\mathbb{F} \in \mathcal{F}$ there exists a multiquadratic poylnomial $u_{\mathbb{F},z} \colon \mathbb{F}^n \to \mathbb{F}$ such that

a. $u_{\mathbb{F},z}(y) = 0$ for all $y \in \mathcal{Y}_{\mathbb{F}}$,

b. $u_{\mathbb{F},z}(z) = 1$, and

c. $u_{\mathbb{F},z} = 0$ for all $w \in \{0,1\}^n \setminus \{z\}$.

From Theorem 1.3.9, each $\mathbb{F} \in \mathcal{F}$ can prevent at most $|\mathcal{Y}_{\mathbb{F}}|$ points from being good. Since $t = |\mathcal{Y}_{\mathbb{F}}|$, there are at least $2^n - t$ good points.

Let G be the set of good points, and thus let $B = \{0,1\}^n \setminus G$ be the set of not-good points. Define

$$p'_{\mathbb{F}}(x) = p_{\mathbb{F}}(x) + \sum_{z \in G} (f'(z) - f(z)) u_{\mathbb{F},z}(x). \tag{1.10}$$

Now, all we need is to show that $p'_{\mathbb{F}}(x)$ satisfies the two conditions from the theorem statement.

First, we show that $p'_{\mathbb{F}}$ extends f'; that is, $p'_{\mathbb{F}}(x) = f'(x)$ for all $x \in \{0,1\}^n$. There are two cases here: $x \in G$ and $x \in B$. If $x \in B$, then the sum term of Equation (1.10) is 0; hence $p'_{\mathbb{F}}(x) = p_{\mathbb{F}}(x)$. Since $p_{\mathbb{F}}(x)$ extends $f(x)$, and since $f(x) = f'(x)$ on B, this means $p'_{\mathbb{F}}(x) = f'(x)$. If $x \in G$, then the only term of the sum where $u_{\mathbb{F},z}(x)$ is nonzero is where $x = G$, as per Items b. and c. above. Hence, we have

$$p'_{\mathbb{F}}(x) = p_{\mathbb{F}}(x) + f'(x) - f(x),$$

and since $p_{\mathbb{F}}(x) = f(x)$, it follows that $p'_{\mathbb{F}}(x) = f'(x)$.

Next, we show that $p'_{\mathbb{F}}(y)$ and $p_{\mathbb{F}}(y)$ agree for all $y \in \mathcal{Y}_{\mathbb{F}}$. Since by Item a. above, we have that $u_{\mathbb{F},z}(y) = 0$, it follows that the entire sum term is zero. Therefore, $p'_{\mathbb{F}}(y) = p_{\mathbb{F}}(y)$ for all $y \in \mathcal{Y}_{\mathbb{F}}$.

As such, we have constructed a polynomial $p'_{\mathbb{F}}$ and a set B that satisfy our conditions of the theorem. □

TODO: Unpack all that

Lemma 1.3.11 ([23, Lemma 7])**.** *Let $m(x_1, \ldots, x_n)$ be a multilinear monomial. Over a field of characteristic other than 2, we have*

$$\sum_{b \in \{-1,1\}} m(b) = 0. \tag{1.11}$$

Proof. For some x_i, we can write $m = x_i \cdot m'$, where the degree of x_i in m' is 0. Then

$$\begin{aligned}\sum_{b\in\{1,-1\}^n} m(b) &= \sum_{a\in\{-1,1\}} \sum_{b'\in\{1,-1\}^{n-1}} a \cdot m'(b') \\ &= \sum_{a\in\{-1,1\}} a \cdot \left(\sum_{b'\in\{1,-1\}^{n-1}} m'(b') \right) \\ &= \left(\sum_{b'\in\{1,-1\}^{n-1}} m'(b') \right) - \left(\sum_{b'\in\{1,-1\}^{n-1}} m'(b') \right) \\ &= 0.\end{aligned}$$

□

1.4 Statistics

In this paper, we will be dealing quite a bit with computers that have access to randomness. Because these computers now have access to randomness, their outputs are no longer deterministic: they can return different results depending on the exact rolls of their random dice.

Definition 1.4.1. A *random variable* is

Definition 1.4.2. Two random variables are *statistically independent* if

Theorem 1.4.3 ([11, Claim 2])**.** *Let $\mathbb{F}$ be a finite field and D a finite set. Let $V \subseteq \mathbb{F}^D$ be a vector space, and let v be a uniform random variable over V. For any subdomains $S, S' \subseteq D$, the restrictions $v|_S$ and $v|_{S'}$ are statistically dependent if and only if there exist constants $c \in \mathbb{F}^S$ and $d \in \mathbb{F}^{S'}$ such that*

1. *There exists $w \in V$ such that $c \cdot w \neq 0$, and*
2. *For all $w \in V$, $c \cdot w = d \cdot w$.*

Proof. □

Chapter 2

Relativization

An important prerequisite to understanding algebrization is the similar, but simpler, concept of *relativization*, also called *oracle separation*. To do this, we first must define an *oracle*.

Definition 2.0.1 ([1, Def. 2.1])**.** An *oracle* A is a collection of Boolean functions $A_m\colon \{0,1\}^m \to \{0,1\}$, one for each natural number m.

There are several ways to think of an oracle; this will extend the most naturally when it comes time to define an extension oracle in Definition 3.0.1.

Another way to think of an oracle is as a subset $A \subseteq \{0,1\}^*$. This allows us to think of A as a language. Since we can do this, it gives us the ability to think of the complexity of the oracle. If we want to think about the subset in terms of our functions, we can write A as

$$A = \bigcup_{m\in\mathbb{N}} \{x \in \{0,1\}^m \mid A_m(x) = 1\}. \tag{2.1}$$

We will use the Iverson bracket defined in Definition 1.3.1 for this purpose: allowing us to think of A as the set and $[A]$ as the function.

The third way to think of an oracle is as a list of bits—this is how a Turing machine thinks of an oracle. In this context, we consider the oracle to be a string of 2^n bits b_i, where b_i is the result of A when given the binary representation of i as input. We will mostly not think of A this way explicitly, but for practical purposes this is how an oracle is encoded whenever we pass it to a Turing machine as an input.

Example 2.0.1.1. Let $m = 3$. The function

$$\begin{aligned} f\colon \{0,1\}^3 &\to \{0,1\} \\ abc &\mapsto b \end{aligned} \tag{2.2}$$

is an oracle function. We can think of f as corresponding to the set $\{010, 011, 110, 111\}$.

Example 2.0.1.2. For each $n \in \mathbb{N}$, define

$$\begin{aligned} f_n\colon \{0,1\}^n &\to \{0,1\} \\ a_1a_2\cdots a_n &\mapsto a_n. \end{aligned} \tag{2.3}$$

Then the set $\{f_n\}$ forms an oracle, whose corresponding language is the set of all binary representations of odd numbers.

TODO: Examples of all the other representations of oracles

An oracle is not particularly interesting mathematical object on its own (after all, it is simply a set of arbitrary Boolean functions); its utility comes from when it interacts with a Turing machine. A normal Turing machine does not have the facilities to interact with an oracle, so we need to define a small extension to a standard Turing machine to allow for this.

TODO:

Figure 2.1: A Turing machine with an oracle

Definition 2.0.2 ([2, Def. 3.6])**.** A *Turing machine with an oracle* is a Turing machine with an additional tape, called the *oracle tape*, as well as three special states: q_{query}, q_{yes}, and q_{no}. Further, each machine is associated with an oracle A. During the execution of the machine, if it ever moves into the state q_{query}, the machine then (in one step) takes the output of A on the contents of the oracle tape, moving into q_{yes} if the answer is 1 and q_{no} if the answer is 0.

TODO:

Figure 2.2: How we will think of an oracle TM

Of course, the question now becomes how we can effectively use an oracle in an algorithm. The previously-mentioned conception of an oracle as a set of strings is useful here. If we consider the set of strings as being a *language* in its own right, then querying the oracle is the same as determining whether a string is in the langauge, just in one step. If the language is computationally hard, this means our machine can get a significant power boost from the right oracle.

TODO:

Figure 2.3: Bit representation of an oracle

Definition 2.0.3 ([1, Def. 2.1])**.** For any complexity class $\mathcal{C}$, the complexity class $\mathcal{C}^A$ is the class of all languages determinable by a Turing machine with access to A in the number of steps defined for $\mathcal{C}$.

We will be using this definition in many places, so we should take a moment to look at it in more depth. First, it is important to realize that $\mathcal{C}^A$ is a set of *languages*, not *machines*: despite the notation, augmenting $\mathcal{C}$ with an oracle does not modify any languages, it just adds new ones that are computable. Second, since a machine can always ignore its oracle, it follows that adding an oracle can only increase the number of languages in the class, never decrease it.

Lemma 2.0.4. *For any complexity class $\mathcal{C}$ and oracle A, $\mathcal{C} \subseteq \mathcal{C}^A$.*

Proof. Let $L \in \mathcal{C}$ and M be a machine that determines L. Then the oracle machine M' that simulates M on its input and makes no queries to the oracle will also accept exactly L. Since M' is a $\mathcal{C}^A$ machine for any oracle A, it follows that $L \in \mathcal{C}^A$ and hence $\mathcal{C} \in \mathcal{C}^A$. □

While the above lemma tells us that $\mathcal{C} \subseteq \mathcal{C}^A$ always, another interesting question is when $\mathcal{C} = \mathcal{C}^A$. We do have a notion for this, called *lowness*. Lowness can be defined for both individual languages and complexity classes; we will define both here.

Definition 2.0.5. A language L is *low* for a class $\mathcal{C}$ if $\mathcal{C}^L = \mathcal{C}$.

Definition 2.0.6. A complexity class $\mathcal{D}$ is *low* for a class $\mathcal{C}$ if each language in $\mathcal{D}$ is low for $\mathcal{C}$.

Of particular interest to us will be classes that are low for *themselves*. We care about these classes because they can use other problems from the same class as a subroutine without issue; in particular recursion and iteration both work here. Thankfully, both P and PSPACE are low for themselves (it turns out NP is probably not); this allows us to easily write algorithms that recurse for classes in both of our most common classes.

Theorem 2.0.7. P *is low for itself.*

Proof. Let $L \in \mathsf{P}$ and let $K \in \mathsf{P}^L$. Let $M(L)$ be the determiner of L and $M(K)$ be the determiner of K. Further, let $\hat{M}(K)$ be the determiner of K but with access to L as an oracle. We aim to show $K \in \mathsf{P}$. Let $p_L(n)$ be a polynomial upper bound of the runtime of $M(L)$ on an input of length n, and let $p_{\hat{K}}(n)$ be similar. Since $M(K)$ can call $M(L)$ no more than $p_{\hat{K}}(n)$ times, it follows that $p_K(n) \leq p_{\hat{K}}(p_L(n))$. Hence, the runtime of $M(K)$ is bounded above by a polynomial, and thus $K \in P$. □

Theorem 2.0.8. PSPACE *is low for itself.*

Proof. The proof is very similar to that for Theorem 2.0.7, but with space instead of time. Since memory usage is bounded above by some polynomial, and polynomials are closed under composition, it follows that PSPACE is low for itself. □

2.1 Defining relativization

We are now ready to define what relativization is. First, note that relativization is a statement about a *result*: we talk about inclusions relativizing, not sets themselves.

Definition 2.1.1. Let $\mathcal{C}$ and $\mathcal{D}$ be complexity classes such that $\mathcal{C} \subseteq \mathcal{D}$. We say the result $\mathcal{C} \subseteq \mathcal{D}$ *relativizes* if $\mathcal{C}^A \subseteq \mathcal{D}^A$ for all oracles A. Conversely, if there exists A such that $\mathcal{C} \nsubseteq \mathcal{D}$, we say that the result $\mathcal{C} \subseteq \mathcal{D}$ *does not relativize*.

Definition 2.1.2. Let $\mathcal{C}$ and $\mathcal{D}$ be complexity classes such that $\mathcal{C} \nsubseteq \mathcal{D}$. We say the result $\mathcal{C} \nsubseteq \mathcal{D}$ *relativizes* if $\mathcal{C}^A \nsubseteq \mathcal{D}^A$ for all oracles A. Conversely, if there exists A such that $\mathcal{C} \subseteq \mathcal{D}$, we say that the result $\mathcal{C} \nsubseteq \mathcal{D}$ *does not relativize*.

We start with a very straightforward example of a relativizing result.

Lemma 2.1.3. *For any oracle A, $\mathsf{P}^A \subseteq \mathsf{NP}^A$. Equivalently, the result $\mathsf{P} \subseteq \mathsf{NP}$ relativizes.*

Proof. Since any deterministic Turing machine is also a nondeterministic machine, it follows that a machine that solves a P^A problem is also an NP^A machine. Hence, $\mathsf{P}^A \subseteq \mathsf{NP}^A$. □

This result tells us that not *everything* is weird in the world of relativization (although we will soon do our best to find all the weird bits): if we have a machine that can do more operations without an oracle, it can still do more operations with an oracle. Further, for the question of P vs. NP that we will discuss in Section 2.3, this means that the question we care about is whether $\mathsf{NP} \subseteq^? \mathsf{P}$ relativizes. As such, the question we are asking simplifies to determining where $\mathsf{P}^A = \mathsf{NP}^A$ and where $\mathsf{P}^A \subsetneq \mathsf{NP}^A$.

Now that we have talked about set inclusions relativizing, we need to define the other side of the coin: *proofs* can relativize as well as results. Unfortunately, this needs to be a somewhat informal definition as formally delineating different types of proof is far beyond the scope of this paper. However, the definition we offer here will be sufficient for our purposes.

Definition 2.1.4. We say a *proof relativizes* if it is not made invalid if the relevant classes are replaced with oracle classes, i.e., a proof that $\mathcal{C} \subseteq \mathcal{D}$ *relativizes* if the same proof can be used to show $\mathcal{C}^A \subseteq \mathcal{D}^A$ for all oracles A with minimal modifications.

This gives us a reason to care about relativization as a concept: if our proofs are relativizing then we know not to try to use them to prove nonrelativizing results. In particular, we will show in Section 2.3 that the famous P vs. NP problem will not relativize regardless of the outcome, and then in Section 2.4 we will show that the common proof technique of diagonalization *does* in fact relativize.

Now that we have given ourselves a reason to care about oracles and how they interact with Turing machines, we now turn to the question of how a machine can gain information about the oracle it queries. We will do this with the notion of *query complexity*.

2.2 Query complexity

The goal of query complexity is to ask questions about some Boolean function $A\colon \{0,1\}^n \to \{0,1\}$ by querying A itself. For this, we will interchangeably think of A as a *function* as well as a bit string of length $N = 2^n$, where each string element is A applied to the ith string of length n, arranged in some lexicographical order. We can

further think of the property itself as being a Boolean function; a function that takes as input the bit-string representation of A and outputs whether or not A has the given property. We will call the function representing the property f. When viewed like this, f is a function from $\{0,1\}^N$ to $\{0,1\}$. We define three types of query complexity for three of the most common types of computing paradigms: deterministic, randomized, and quantum. Nondeterministic query complexity is interesting, but it is outside the scope of this paper.

Definition 2.2.1 ([1, p. 17])**.** Let $n \in \mathbb{N}$ and let $A : \{0,1\}^n \to \{0,1\}$ be an oracle. We can write A using 2^n bits, where bit i is the output of A when given the binary representation of i as input. Define $N = 2^n$, and let $f\colon \{0,1\}^N \to \{0,1\}$ be a Boolean function. Then the *deterministic query complexity* of f, which we write $D(f)$, is the minimum number of queries made by any deterministic algorithm with access to an oracle A that determines the value of $f(A)$.

To make this more clear, let us give an example problem.

Definition 2.2.2. The OR problem is the following oracle problem:

> Let $A\colon \{0,1\}^n \to \{0,1\}$ be an oracle. The function $\mathsf{OR}(A)$ returns 1 if there exists a string on which A returns 1, and 0 otherwise.

The question is then what the deterministic query complexity of the OR function is.

Theorem 2.2.3. *The* OR *problem has a deterministic query complexity of* 2^n*.*

Proof. First, note that any algorithm that determines the OR problem can stop as soon as it queries A and gets an output of 1. Hence, for any algorithm M, let $\{s_i\}$ be the sequence of queries M makes to A on the assumption that it always recieves a response of 0. If $|\{s_i\}| \leq 2^n$, there exists some $s \in \{0,1\}^n$ not queried. In that case, M will not be able to distinguish the zero oracle from the oracle that outputs 1 only when given s. Hence, M must query every string of length n and thus the query complexity is 2^n. □

From this, we get that the OR problem cannot be solved any better than by enumerative checking. This makes intuitive sense because none of the results we get by querying A imply anything about what A will do on other values, since A can be an arbitrary function. Later on (in Section 3.1), we will look at what happens when we give ourselves access to a *polynomial*, where querying one point could tell us information about others.

For the next two definitions, since their Turing machines include some element of randomness, we only require that they succeed with a 2/3 probability. This is in line with most definitions of complexity classes involving random computers.

Definition 2.2.4 ([1, p. 17])**.** Let $f\colon \{0,1\}^N \to \{0,1\}$ be a Boolean function. Then the *randomized query complexity* of f, which we write $D(f)$, is the minimum number of queries made by any randomized algorithm with access to an oracle A that evaluates $f(A)$ with probability at least 2/3.

Definition 2.2.5 ([1, p. 17])**.** Let $f\colon \{0,1\}^N \to \{0,1\}$ be a Boolean function. Then the *quantum query complexity* of f, which we write $D(f)$, is the minimum number of queries made by any quantum algorithm with access to an oracle A that evaluates $f(A)$ with probability at least $2/3$.

2.3 Relativization of P vs. NP

An important example of relativization is that of P and NP. While the question of if $\mathsf{P} = \mathsf{NP}$ is still open, we aim to show that *regardless of the answer*, the result does not algebrize. To do this, we show that there are some oracles A where $\mathsf{P}^A = \mathsf{NP}^A$, and some where $\mathsf{P}^A \neq \mathsf{NP}^A$.

Additionally, it should be noted that the similarity of relativization to algebrization means that the structure of these proofs will return in Section 3.2 when we show the algebrization of P and NP.

2.3.1 Equality

The more straightforward of the two proofs is the oracle where $\mathsf{P}^A = \mathsf{NP}^A$, so we shall begin with that.

Theorem 2.3.1 ([7, Theorem 2])**.** *There exists an oracle A such that $\mathsf{P}^A = \mathsf{NP}^A$.*

Proof. For this, we can let A be any PSPACE-complete language. By letting our machine in P be the reducer from A to any other language in PSPACE, we therefore get that $\mathsf{PSPACE} \subseteq \mathsf{P}^A$. Similarly, if we have a problem in NP^A, we can verify it in polynomial space without talking to A at all (by having our machine include a determiner for A). Hence, we have that $\mathsf{NP}^A \subseteq \mathsf{NPSPACE}$. Further, a celebrated result of Savitch [29] (which we briefly discussed as Theorem 1.2.17) is that $\mathsf{PSPACE} = \mathsf{NPSPACE}$. Combining all these results, we get the chain

$$\mathsf{NP}^A \subseteq \mathsf{NPSPACE} = \mathsf{PSPACE} \subseteq \mathsf{P}^A \subseteq \mathsf{NP}^A. \tag{2.4}$$

This is a circular chain of subset relations, which means everything in the chain must be equal. Hence, $\mathsf{P}^A = \mathsf{NP}^A = \mathsf{PSPACE}$. □

For a slightly more intuitive view of what this proof is doing, what we have done is found an oracle that is so powerful that it dwarfs any amount of computation our actual Turing machine can do. Hence, the power of our machine is really just the same as the power of our oracle, and since we have given both the P and NP machine the same oracle, they have the same power.

2.3.2 Inequality

Having shown that an oracle exists where $\mathsf{P}^A = \mathsf{NP}^A$, we now endeavor to find one where $\mathsf{P}^A \neq \mathsf{NP}^A$. This piece of the proof is less simple than the previous section, and it uses a diagonalization argument to construct the oracle. Before we dive in to the main proof, however, we need to define a few preliminaries.

Definition 2.3.2 ([7, p. 436])**.** Let X be an oracle. The language $L(X)$ is the set

$$L(X) = \{x \mid \text{there is } y \in X \text{ such that } |y| = |x|\}.$$

Example 2.3.2.1. Consider the language $X = \{0, 11, 0100\}$. The language $L(X)$ is the language consisting of all strings of length 1, 2, and 4.

Our eventual goal will be to construct a language X such that $L(X) \in \mathsf{NP}^X \setminus \mathsf{P}^X$. Of particular note is that we can rather nicely put a upper bound on the complexity of $L(X)$ when given X as an oracle, regardless of the value of X. This fact is what gives us the freedom to construct X in such a way that $L(X)$ will not be in P^X.

Lemma 2.3.3 ([7, p. 436])**.** *For any oracle X, $L(X) \in \mathsf{NP}^X$.*

Proof. Let S be a string of length n. If $S \in L(X)$, then a witness for S is any string S' such that $|S| = |S'|$ and $S' \in X$. Since a machine with query access to X can query whether S' is in X in one step, it follows that we can verify that $S \in L(X)$ in polynomial time. □

With this lemma as a base, we can now move on to our main theorem.

Theorem 2.3.4 ([7, Theorem 3])**.** *There exists an oracle A such that $\mathsf{P}^A \neq \mathsf{NP}^A$.*

Proof. Our goal is to construct a set B such that $L(B) \notin \mathsf{P}^B$. We shall construct B in an interative manner. We do this by taking a sequence $\{P_i\}$ of all machines that recongize some language in P^A, and then constructing B such that for each machine in the sequence, there is some part of $L(B)$ it cannot recognize. This technique is called *diagonalization*, and it is used in many places in computer science theory.[1] Additionally, we define $p_i(n)$ to be the maximum running time of P_i on an input of length n. We aim to show that Algorithm 2.1 constructs B.

To begin, let us demonstrate the algorithm's soundness. First, note that since P_i runs in polynomial time, $p_i(n)$ is bounded above by a polynomial, and hence there will always exist an n as defined in line 4. Next, since there are 2^n strings of length n and since $p_i(n) < 2^n$, we know that there must be some x to make line 7 well-defined. While our algorithm allows x to be any string, if it is necessary to be explicit in which we choose, then picking x to be the smallest string in lexicographic order is a standard choice.

We should also briefly mention that this algorithm does not terminate. This is okay because we are only using it to construct the set B, which does not need to be bounded. If this were to be made practical, since the sequence of n_is is monotonically increasing, the set could be constructed "lazily" on each query by only running the algorithm until n_i is greater than the length of the query.

Next, we demonstrate that $L(B) \notin \mathsf{P}^B$. The end goal of our instruction is a set B such that if P_i^B accepts 0^n then there are no strings of length n in B, and if P_i^B

[1] This argument style is named after *Cantor's diagonal argument*, which was originally used to prove that the real numbers are uncountable [28, Thm. 2.14].

Input: A sequence of P oracle machines $\{P_i\}_{i=1}^{\infty}$
Output: A set B such that $L(B) \notin \mathsf{P}^B$
1 $B(0) \leftarrow \varnothing$;
2 $n_0 \leftarrow 0$;
3 **for** *i starting at* 1 **do**
4 Let $n > n_i$ be large enough that $p_i(n) < 2^n$;
5 Run $P_i^{B(i-1)}$ on input 0^n;
6 **if** $P_i^{B(i-1)}$ *rejects* 0^n **then**
7 Let x be a string of length n not queried during the above computation;
8 $B(i) \leftarrow B(i-1) \sqcup \{x\}$;
9 **end**
10 $n_{i+1} \leftarrow 2^n$;
11 **end**
12 $B \leftarrow \bigcup_i B(i)$;

Algorithm 2.1: An algorithm for constructing B

rejects, then there is a string of length n in B. This means that no P_i accepts $L(B)$, and hence $L(B) \notin \mathsf{NP}^B$.

The central idea behind the proper functioning of our algorithm is that adding strings to our oracle *cannot change the output if they are not queried*. This is what we do in line 4: we need our input length to be long enough to guarantee that a non-queried string exists. Since the number of queried strings is no greater than $p_i(n)$, and there are 2^n strings of length n, there must be some string not queried.

Next, we run $P_i^{B(i-1)}$ on all the strings we have already added. If it accepts, then we want to make sure that no string of length n is in B; that is, 0^n is not in $L(B)$. Hence, in this particular loop we add nothing to $B(i)$. If $P_i^{B(i-1)}$ rejects, we then need to make sure that $0^n \in L(B)$ but in a way that does not affect the output of $P_i^{B(i-1)}$. Hence, we find a string that $P_i^{B(i-1)}$ did not query (and thus will not affect the result) and add it to $B(i)$.

Having done this, we then set n_{i+1} to be 2^n. Since $p_i(n) < 2^n$, it follows that no previous machine could have queried any strings of length n_{i+1}.[2] This way, we ensure our previous machines do not accidentally have their output change due to us adding a string they queried.

Having run this over all polynomial-time Turing machines, we have a set $L(B)$ such that no machine in P^B accepts it, which tells us $L(B) \notin \mathsf{P}^B$. But, Lemma 2.3.3 already told us $L(B) \in \mathsf{NP}^B$. Hence, $\mathsf{P}^B \neq \mathsf{NP}^B$. □

[2] A word of caution: we only care about what P_i does on input n_i, *not any other input*. This is because we only need each machine to be incorrect for some i, not all i.

2.4 Diagonalization relativizes

Of course, determining that P vs NP does not relativize is only important if the proof techniques used in practice *do* in fact relativize. Rather unfortunately, it turns out that simple diagonalization is a relativizing result.

While diagonalization itself does not have a formal definition, we can still think about it informally. Looking at our construction of B, which we did using diagonalization, notice that our definition never really cared about how the P_i worked, just about the results it produced. Hence, if it were to be possible to modify Algorithm 2.1 to construct $B \in \mathsf{NP} \setminus \mathsf{P}$, the proof would remain the same if we were to replace our sequence $\{P_i\}$ with a sequence of machines in P^A for some PSPACE-complete A. However, this would lead to a contradiction, as we showed in Theorem 2.3.1 that in that case, $\mathsf{P}^A = \mathsf{NP}^A$! This tells us that a simple diagonalization argument would not suffice to determine separation between P and NP.

2.5 Arithmetization does not relativize

While we know that diagonalization relativizes, in the years since the Baker, Gill, and Solovay paper researchers have discovered proof techniques that do not in fact relativize. One of these techniques is *arithmetization*, introduced by [6].

The idea behind arithmetization is that we want to be able to reduce computational problems to algebraic ones. More specifically, we would like to reduce our problems to ones involving low-degree polynomials over a finite field (such as those seen in Section 1.3). In this paper, we will care about arithmetization for two reasons: because it is a non-relativizing technique (as we are about to see) and because we will be using it later on in this paper as an important part of several proofs.

Chapter 3

Algebrization

Algebrization, originally described by Aaronson and Wigderson [1], is an extension of relativization. While relativization deals with oracles that are Boolean functions, algebrization extends oracles to be a collection of polynomials over finite fields. Since any field contains the set $\{0,1\}$, we can think about our new oracles as *extending* some specific oracle A, so that both oracles agree on the set $\{0,1\}^n \subseteq \mathbb{F}^n$. We formalize this notion below.

TODO:

Figure 3.1: An extension oracle

Definition 3.0.1 ([1, Def. 2.2])**.** Let $A_m \colon \{0,1\}^m \to \{0,1\}$ be a Boolean function and let $\mathbb{F}$ be a finite field. Then an *extension* of A_m over $\mathbb{F}$ is a polynomial $\tilde{A}_{m,\mathbb{F}} \colon \mathbb{F}^m \to \mathbb{F}$ such that $\tilde{A}_{m,\mathbb{F}}(x) = A_m(x)$ whevever $x \in \{0,1\}^m$. Also, given an oracle $A = (A_m)$, an *extension* $\tilde{A}$ of A is a collection of polynomials $\tilde{A}_{m,\mathbb{F}} \colon \mathbb{F}^m \to \mathbb{F}$, one for each positive integer m and finite field $\mathbb{F}$, such that

1. $\tilde{A}_{m,\mathbb{F}}$ is an extension of A_m for all $m, \mathbb{F}$, and
2. there exists a constant c such that $\mathrm{mdeg}(\tilde{A}_{m,\mathbb{F}}) \leq c$ for all $m, \mathbb{F}$.

Take note that an oracle can have many different extension oracles, since one can construct an infinite number of polynomials that go through a set of points. For this reason, when dealing with oracles in practice, we will also often be interested in oracles of a particular multidegree, which limits our options for oracles in potentially-interesting ways.

Example 3.0.1.1. Consider the function we defined in Example 2.0.1.1:

$$\begin{aligned} f \colon \{0,1\}^3 &\to \{0,1\} \\ abc &\mapsto b. \end{aligned} \tag{3.1}$$

An extension of that function is the polynomial

$$\begin{aligned} \tilde{f} \colon \mathbb{F}^3 &\to \mathbb{F}^3 \\ (a,b,c) &\mapsto b. \end{aligned} \tag{3.2}$$

While this is a relatively trivial polynomial, there are more non-trivial ones, for example

$$\begin{aligned} \tilde{f}\colon \mathbb{F}^3 &\to \mathbb{F}^3 \\ (a,b,c) &\mapsto a^3c^3 + b^2 - ac. \end{aligned} \tag{3.3}$$

Notice that on $\{0,1\}$, $x^2 = x$, which allows us to see that $\tilde{f}$ is a valid extension of f.

Definition 3.0.2 ([1, Def. 2.2])**.** For any complexity class $\mathcal{C}$ and extension oracle $\tilde{A}$, the complexity class $\mathcal{C}^{\tilde{A}}$ is the class of all languages determinable by a Turing machine with access to $\tilde{A}$ with the requirements for $\mathcal{C}$.

Next, we need to formally define what algebrization is.

Definition 3.0.3 ([1, Def. 2.3])**.** Let $\mathcal{C}$ and $\mathcal{D}$ be complexity classes such that $\mathcal{C} \subseteq \mathcal{D}$. We say the result $\mathcal{C} \subseteq \mathcal{D}$ *algebrizes* if $\mathcal{C}^A \subseteq \mathcal{D}^{\tilde{A}}$ for all oracles A and finite field extensions $\tilde{A}$ of A. Conversely, if there exists A and $\tilde{A}$ such that $\mathcal{C} \not\subseteq \mathcal{D}$, we say that the result $\mathcal{C} \subseteq \mathcal{D}$ *does not algebrize*.

Definition 3.0.4 ([1, Def. 2.3])**.** Let $\mathcal{C}$ and $\mathcal{D}$ be complexity classes such that $\mathcal{C} \not\subseteq \mathcal{D}$. We say the result $\mathcal{C} \not\subseteq \mathcal{D}$ *algebrizes* if $\mathcal{C}^A \not\subseteq \mathcal{D}^{\tilde{A}}$ for all oracles A and finite field extensions $\tilde{A}$ of A. Conversely, if there exists A and $\tilde{A}$ such that $\mathcal{C} \subseteq \mathcal{D}$, we say that the result $\mathcal{C} \not\subseteq \mathcal{D}$ *does not algebrize*.

3.1 Algebraic query complexity

Similarly to how we defined query complexity in Section 2.2, our notion of algebrization requires a definition of *algebraic* query complexity.

Definition 3.1.1 ([1, Def. 4.1])**.** Let $f\colon \{0,1\}^N \to \{0,1\}$ be a Boolean function, $\mathbb{F}$ be a field, and c be a positive integer. Also, let $\mathbb{M}$ be the set of deterministic algorithms M such that $M^{\tilde{A}}$ outputs $f(A)$ for every oracle $A\colon \{0,1\}^n \to \{0,1\}$ and every finite field extension $\tilde{A}\colon \mathbb{F}^n \to \mathbb{F}$ of A with $\mathrm{mdeg}(\tilde{A}) \leq c$. Then, the deterministic algebraic query complexity of f over $\mathbb{F}$ is defined as

$$\tilde{D}_{\mathbb{F},c}(f) = \min_{M\in\mathcal{M}} \left(\max_{A,\tilde{A}:\mathrm{mdeg}(\tilde{A})\leq c} T_M(\tilde{A}) \right), \tag{3.4}$$

where $T_M(\tilde{A})$ is the number of queries to $\tilde{A}$ made by $M^{\tilde{A}}$.

Our goal here is to find the *worst*-case scenario for the *best* algorithm that calculates the property f. The difference between this and Definition 2.2.1 is twofold: first, our algorithm M has access to an extension oracle of A, and second, that we can limit our $\tilde{A}$ in its maximum multidegree. For the most part, we will focus on equations with multidegree 2, which is enough to get the results we want.

As an example, let us look at the same OR problem we defined in Definition 2.2.2.

Theorem 3.1.2 ([1, Thm. 4.4])**.** $\tilde{D}_{\mathbb{F},2}(\mathsf{OR}) = 2^n$ *for every field* $\mathbb{F}$.

Proof. First note that 2^n is an upper bound for the number of queries necessary since we can query every point in $\{0,1\}^n$, of which there are 2^n.

Let M be a deterministic algorithm and let $\mathcal{Y}$ be the set of points queried by M in the case where M always recieves 0 as a response. So long as $|\mathcal{Y}| < 2^n$, there exists by Theorem 1.3.9 a multiquadratic extension polynomial $\tilde{A}\colon \mathbb{F}^n \to \mathbb{F}$ such that $\tilde{A}(y) = 0$ for all $y \in \mathcal{Y}$ but $\tilde{A}(w) = 1$ for some $w \in \{0,1\}^n$. As such, if M queries less than 2^n points then it would not be able to tell the difference between $\tilde{A}$ and the zero function. However, $\mathsf{OR}(A) = 1$ and $\mathsf{OR}(0) = 0$, so it would get the incorrect answer for one of them. Hence if M queries fewer than 2^n points it cannot solve the OR problem. □

Note that this works even if M is adaptive: if M ever recieves a nonzero response it (correctly) knows $\mathsf{OR}(A) = 1$, so it can accept immediately. As such, we know that any contradiction must come when M has only ever seen zeros as responses.

This gives us a potentially counterintuitive property of algebraic query complexity: while it would seem that giving our machine a polynomial (and a polynomial of multidegree only 2, at that) would give us the ability to solve the hardest problems more quickly, that turns out not to be the case.

Now, while this is true for polynomials of multidegree 2, it turns out that if we restrict our oracles to being simply *multilinear* polynomials, we do get a speedup.

Theorem 3.1.3 ([23, Thm. 3])**.** $\tilde{D}_{\mathbb{F},1}(\mathsf{OR}) = 1$ *for every field* $\mathbb{F}$ *with characteristic not equal to* 2*.*

Proof. Let $A\colon \{0,1\}^n \to \{0,1\}$ and $\tilde{A}$ be our extension polynomial. Consider the value of $p(1/2,\dots,1/2)$. We aim to show that this value is equal to 0 if and only if A is the zero oracle.

Consider the function

$$p'(x_1,\dots,x_n) = p(1-2x_1,\dots,1-2x_n). \tag{3.5}$$

Since $1-2x$ is a linear polynomial, it follows that p' is itself a multilinear polynomial. Further, since the sum over $\{1,-1\}^n$ of a non-constant multilinear monomial is 0 as per Lemma 1.3.11, it follows that

$$\sum_{b\in\{-1,1\}^n} p'(b) = p'(0,\dots,0), \tag{3.6}$$

i.e., the constant term of p'. Further, from our definition of p', we have that $p'(0,\dots,0) = p(1/2,\dots,1/2)$. Hence, we have

$$\sum_{b\in\{0,1\}^n} p(b) = p(1/2,\dots,1/2). \tag{3.7}$$

Since $p(b) \geq 0$ for all $b \in \{0,1\}^n$, it follows that $p(1/2,\dots,1/2)$ is 0 if and only if $p(b) = 0$ for all $b \in \{0,1\}^n$, i.e. exactly when A is the zero function. □

3.2 Algebrization of P vs. NP

As with relativization, an important application of algebrization is in regards to the P vs. NP problem.

Definition 3.2.1 ([5, Def. 6.1])**.** A language L is PSPACE-*robust* if $\mathsf{P}^L = \mathsf{PSPACE}^L$.

Lemma 3.2.2. *Any* PSPACE*-complete language is also* PSPACE*-robust.*

Proof. First, we know from Lemma 2.0.4 that $\mathsf{P}^L \subseteq \mathsf{PSPACE}^L$. Next, let $M \in \mathsf{PSPACE}^L$, and we aim to show $M \in \mathsf{P}^L$. Since $L \in \mathsf{PSPACE}$ and PSPACE is low for itself, we know $M \in \mathsf{PSPACE}$. As such, we know there is a polynomial-time reduction f from M to L. Hence, we can compute M by running f on the input and then testing if that output is in L (using the oracle). Hence, $M \in \mathsf{P}^L$ and thus $\mathsf{P}^L = \mathsf{PSPACE}^L$. □

Lemma 3.2.3 ([5, Lemma 6.2])**.** *Let L be a* PSPACE*-robust language, with corresponding oracle A. Let $\tilde{A}$ be the unique multilinear extension oracle of A. Then the language*

$$\tilde{L} = \bigcup_{n \in \mathbb{N}} \{(x_1, \ldots, x_n, z) \in \mathbb{F}^{n+1} \mid \tilde{A}(x_1, \ldots, x_n) = z\} \tag{3.8}$$

is polynomially-equivalent to L; that is, $\tilde{L} \in \mathsf{P}^L$ and $L \in \mathsf{P}^{\tilde{L}}$.

The proof of this statement originally given in [5] has some apparent problems; we discuss these more thoroughly later on in Appendix B. Instead, we present our own proof of the above lemma.

Proof. First, we provide a polynomial-time reduction from L to $\tilde{L}$. Since for all $x \in \{0,1\}^n$, $\tilde{A}(x) = 1$ if and only if $x \in L$, it follows that

$$\begin{aligned} f \colon \Sigma^* &\to \Sigma^* \\ x &\mapsto (x, 1) \end{aligned} \tag{3.9}$$

is a polynomial-time reduction from L to L'.

Next, consider Algorithm 3.1. This algorithm simply calculates the value of $\tilde{A}(x_1, \ldots, x_n)$ directly, from the explicit definition we gave in Corollary 1.3.7, and then compares it to the value of z. As such, this is a determiner for L.

We now demonstrate that Algorithm 3.1 runs in P^L. From the definition of PSPACE-robustness, we know that we only need to show that the algorithm runs in PSPACE^L, a much weaker bound. The inner for-loop runs in polynomial *time*, hence it must run in polynomial space. The outer for-loop runs for 2^n iterations, so determining that it is in P^L is non-trivial. Beyond the inner loop (which we have already discussed), the only thing we do in the outer loop is simulate L, which can be done in one step with access to an oracle for L.

The only memory we need to simulate this oracle (beyond that for the input) is space for d and z'. We have already shown d needs polynomial space, so what remains

Input: $(x_1, \ldots, x_n, z) \in \mathbb{F}^{n+1}$
Output: Whether $\tilde{A}(x_1, \ldots, x_n) = z$

```
1  z' ← 0;
2  for k ∈ {0,1}^n do
3  |  Simulate L on input k;
4  |  if k ∈ L then
   |  |  // Compute d_k(x)
5  |  |  d ← 1;
6  |  |  for i from 1 to n do
7  |  |  |  if k_i = 1 then
8  |  |  |  |  d ← d · x_i;
9  |  |  |  else
10 |  |  |  |  d ← d · (1 − x_i);
11 |  |  |  end
12 |  |  end
13 |  |  z' ← z' + d;
14 |  end
15 end
16 return whether z = z';
```

Algorithm 3.1: Determiner for $\tilde{L}$

is z'. Since $A(x_1, \ldots, x_n) \in \{0, 1\}$, each term in the sum in Equation (1.7) is bounded above by $\delta_\beta(x)$. This means that the value of z' that we compute is bounded above by

$$2^n \max_{k \in \{0,1\}^n} \delta_k(x). \tag{3.10}$$

Since each $\delta_k(x)$ can be written in polynomial space, and 2^n can be *written* in polynomial space, it follows that z' can as well. Hence, Algorithm 3.1 is in PSPACE^L, and thus is in P^L.

Next, we show that Algorithm 3.1 determines $\tilde{L}$. As mentioned earlier, our algorithm computes $\tilde{A}$ directly through the equations given in Corollary 1.3.7. First, we show the inner loop (beginning on line 6) computes $\delta_k(x)$. We compute δ directly, through the formula described at Equation (1.8). We do this by simply iterating through each i and then multiplying d by either x_i or $1 - x_i$, as appropriate.

Second, in this case Equation (1.7) simplifies to

$$\tilde{A}_n(x_1, \ldots, x_n) = \sum_{\beta \in L} \delta_\beta(x_1, \ldots, x_n). \tag{3.11}$$

This is exactly what our outer loop does: computes the sum directly through iteration. Hence, the only thing the above algorithm does is calculate $\tilde{A}_n(x_1, \ldots, x_n)$ and then compares it to the value we were given. As such, it determines $\tilde{L}$.

Since there is a reduction from L to $\tilde{L}$, we know that L is no harder than $\tilde{L}$, and Algorithm 3.1 demonstrates that $\tilde{L} \in \mathsf{PSPACE}$. Hence, $\tilde{L}$ is PSPACE-complete. □

With that as a base, we can now move on to the main theorem. As before, the more straightforward proof is the oracle where $\mathsf{P}^{\tilde{A}} = \mathsf{NP}^{A}$, so we begin with that.

Theorem 3.2.4 ([1, Theorem 5.1])**.** *There exist A, $\tilde{A}$ such that $\mathsf{NP}^{A} = \mathsf{P}^{\tilde{A}}$.*

Proof. For this theorem, we use the same technique we did in our proof of Theorem 2.3.1: find a PSPACE-complete language A and work from there. If we let $\tilde{A}$ be the unique multilinear extension of A, Lemma 3.2.3 tells us $\tilde{A}$ is PSPACE-complete. Hence, as mentioned before, we have $\mathsf{NP}^{\tilde{A}} \subseteq \mathsf{NP}^{\mathsf{PSPACE}} \subseteq \mathsf{NPSPACE}$, and since $\mathsf{NPSPACE} = \mathsf{PSPACE}$ and we know from Theorem 2.3.1 that $\mathsf{PSPACE} \subseteq \mathsf{P}^{A}$, it follows

$$\mathsf{NP}^{\tilde{A}} = \mathsf{NP}^{\mathsf{PSPACE}} = \mathsf{PSPACE} = \mathsf{P}^{A}.$$

□

Now it is time for the other case.

Theorem 3.2.5 ([1, Theorem 5.3])**.** *There exist A, $\tilde{A}$ such that $\mathsf{NP}^{A} \neq \mathsf{P}^{\tilde{A}}$.*

Proof. Like in Theorem 2.3.4, we aim to "diagonalize": iterate over all $\mathsf{P}^{\tilde{A}}$ machines to construct a language that none of them can recognize. Also like before, we will do this by constructing an oracle extension $\tilde{A}$ such that $L(A) \notin \mathsf{P}^{\tilde{A}}$. Since we only give an algebraic extension to P and not NP, we can resuse the result from Lemma 2.3.3 that $L(A) \in \mathsf{NP}^{A}$. We shall construct $\tilde{A}$ using the following algorithm: As before, we will start by demonstrating soundness and then move on to why the constructed oracle provides the separation we seek.

Perhaps the least intuitive section of the above algorithm is the section beginning at line 8. We want to leverage Lemma 1.3.10 to show that such a solution exists. We know that $p_i(n) < 2^n$, and since $p_i(n)$ is an upper bound on the number of total queries, this tells us that there is at least one $w \in \{0,1\}^{n_i}$ not queried. From the definition of $\mathcal{Y}_{\mathbb{F}}$, we also therefore know that $\sum_{\mathbb{F}} \mathcal{Y}_{\mathbb{F}} < 2^n$. Further setting up this lemma, we will let f be the zero function and $p_{\mathbb{F}}$ be the zero polynomial.

From the lemma, we know that there is some $B \in \{0,1\}^n$ with $|B| < 2^n$ such that for all f' agreeing with f there exists a series of $p'_{\mathbb{F}}$ extending f' and agreeing with $p_{\mathbb{F}}$ on $\mathcal{Y}_{\mathbb{F}}$. As such, if we pick any $w \in \{0,1\}^n \setminus B$, then the function $f'(x) = [x = w]$ agrees with f on B, and thus we know that there exists a series of $p'_{\mathbb{F}}$ that agree with the zero polynomial on $\mathcal{Y}_{\mathbb{F}}$ and each non-w Boolean point.

Now, we know that such a solution exists, and Equation (1.10) gives us an explicit formula for our $A_{n_i,\mathbb{F}}$; thus, we know that this is in fact computable. Since this algorithm is simply for *constructing* the language, we do not care about time or space complexity, so the fact that it is computable is enough. In terms of finding the w we need, we can simply iterate try the construction for each $w \in \{0,1\}^n$ and stop as soon as we are able to construct each polynomial.

The other component of soundness is determining how we can run P_i with the extension oracle $\tilde{A}$ when $\tilde{A}$ is not yet fully constructed. What we do is when simulating P_i, we assume that any $\tilde{A}_{n_i,\mathbb{F}}$ that we have not yet queried returns zero on all queried

Input: A sequence of P oracle machines $\{P_i\}_{i=1}^{\infty}$
Output: An extension oracle $\tilde{A}$ such that $L(A) \notin \mathsf{P}^{\tilde{A}}$

$\tilde{A} \leftarrow \varnothing$;
$n_0 \leftarrow 0$;
for *i starting at* 1 **do**
  Let $n > n_i$ be large enough that $p_i(n) < 2^n$;
  Run $P_i^{\tilde{A}}$ on input 0^n;
  if $P_i^{B(i-1)}$ *rejects* 0^n **then**
    Let $\mathcal{Y}_{\mathbb{F}}$ be the set of all $y \in \mathbb{F}^{n_i}$ queried during the above computation;
    `// See Lemma 1.3.10 for why we can do this`
    Let $w \in \{0,1\}^n$ such that the following works;
    for *all* $\mathbb{F}$ **do**
      Set $\tilde{A}_{n_i,\mathbb{F}}$ to be a multiquadratic polynomial such that $\tilde{A}_{n_i,\mathbb{F}}(w) = 1$ and $\tilde{A}_{n_i,\mathbb{F}}(y) = 0$ for all $y \in \mathcal{Y}_{\mathbb{F}} \cup (\{0,1\}^{n_i} \setminus \{w\})$;
    end
  else
    Set $\tilde{A}_{n_i,\mathbb{F}} = 0$ for all $\mathbb{F}$;
  end
  $n_{i+1} \leftarrow 2^n$;
end
$B \leftarrow \bigcup_i B(i)$;

Algorithm 3.2: An algorithm for constructing $\tilde{A}$

inputs. We then make sure that any time we set an $\tilde{A}_{n_i,\mathbb{F}}$, it also returns zero on any point that we queried. Further, we ensure that each n_i is large enough that no previous machine would have queried any string of length n_i on its respective input; ergo modifying these polynomials would not have any affect on their output.

Next, we show that $L(A)$ is not in $\mathsf{P}^{\tilde{A}}$. As we did in Theorem 2.3.4, the idea is that for each polynomial-time machine P_i, that machine will return the incorrect result on the string 0^{n_i}. We do this in Algorithm 3.2 by simulating P_i on the input, and then adjusting $\tilde{A}$ based on its output. We separate this into two cases: the case where $P_i^{\tilde{A}}$ rejects 0^{n_i}, and the case where it accepts. We shall begin with the case where it accepts.

When $P_i^{\tilde{A}}$ accepts, we want to ensure that no strings of length n_i are in A. The unique low-degree extension of the zero function is the zero polynomial; hence, we set $\tilde{A}_{n_i,\mathbb{F}}$ to be 0 for all $\mathbb{F}$. This ensures $\tilde{A}(x) = 0$ for all $x \in \{0,1\}^{n_i}$, and thus $A \cap \{0,1\}^{n_i} = \varnothing$. This means $0^{n_i} \notin L(A)$ and thus $P_i^{\tilde{A}}$ is incorrect.

When $P_i^{\tilde{A}}$ accepts, we want to make sure that there is at least some string $w \in A \cap \{0,1\}^{n_i}$, but also to make sure that any polynomials we add have their values align with what $P_i^{\tilde{A}}$ already saw. As we mentioned earlier, we know that such a polynomial exists, and thus we construct it. Since our constructed polynomials tell us that $w \in A$, it follows that $0^{n_i} \in L(A)$ and hence $P_i^{\tilde{A}}$ is incorrect there as well.

Since our argument earlier told us that none of the P_i machines would have their

output affected by any of the polynomials modified outside of the corresponding iteration i, it follows that no machine P_i could recognize $L(A)$. Since P_i includes every machine recognizing a $\mathsf{P}^{\tilde{A}}$ language, it follows that $L(A) \notin \mathsf{P}^{\tilde{A}}$. □

3.3 Arithmetization algebrizes

In Section 2.5, we mentioned arithmetization as an example of a technique that does not relativize. One might in fact hope that we can continue that logic here as well, and show that arithmetization is non-algebrizing. Unfortunately, this is not the case.[1]

[1]This is no coincidence—algebrization was created pretty much entirely to break this technique.

Chapter 4

Interactive proof systems

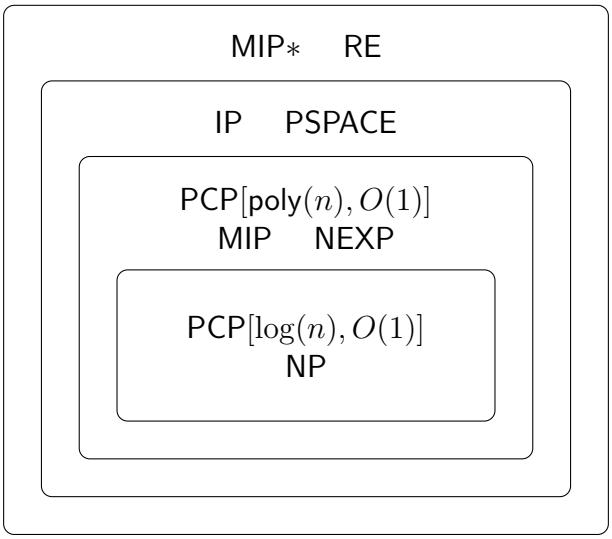

Figure 4.1: The interactive-proof classes and their relationships

Interactive proof systems are models of computation that involve multiple Turing machines exchanging messages between each other. In general, the machines are split into two categories: those that are computationally unbounded but untrustworthy (the provers), and those that are bounded but trustworthy (the verifiers). The "goal" of the system is to convince the verifier of whether or not the string is in the language. These systems almost always use randomness as part of their design: for this reason, almost all of the bounds are "with high probability" bounds and not complete mathematical certainty.

Interactive proof systems turn out to be surprisingly powerful—while the verifier only runs in polynomial time, it turns out that the interaction with the untrustworthy computer is still enough to boost the power significantly. The "classic" interactive proof model involves exactly two computers (one prover, one verifier), but many variants exist, all with distinct and interesting complexity-theoretic characteristics. In this chapter, we will introduce a good number of these variants and a few of their

interesting properties; the goal of the remaining chapters will be to prove several of the more modern interesting results involving these systems.

4.1 Interactive Turing machines

The central mechanism underlying all of the interactive proof systems we will work with is the interactive Turing machine. This machine is a variant of a standard Turing machine, but it has the ability to communicate with another machine as part of its work. When multiple interactive machines work together, they can produce a joint computation in the same way that a single non-interactive machine can. From there, an interactive proof is just a pair of interactive machines working together, with some particular constraints on what they are allowed to do.

Definition 4.1.1 ([14, Def. 4.2.1])**.** An *interactive Turing machine* is a deterministic multi-tape Turing machine with the following tapes:

- Input tape (read-only)
- Output tape (write-only)
- Two communication tapes (one read-only, one write-only)
- One-cell switch tape (read-write)
- Work tape (read-write)

In addition to these tapes, an interactive TM has a single bit $\sigma \in \{0, 1\}$ associated with it, called its *identity*. When the content of the switch tape is not equal to the machine's identity, the machine performs no computation and is called *idle*.

In most cases, we will also give the interactive machines a source of randomness as well that they can read from. Since this is so common we will treat it as the default; if we ever want a machine to not have a source of randomness we will explicitly state as such.

On its own, a single interactive Turing machine is not worth much: in order to do work with these we need to define how a pair of them interact. The chief mechanism of interacting Turing machines is that of *shared tapes*. Shared tapes are tapes where any modifications can be seen by both Turing machines immediately. While the tapes themselves are shared, the *heads* are not: the two machines are perfectly capable of looking at different entries at the same time.

Definition 4.1.2 ([14, Def. 4.2.2])**.** A pair of interactive Turing machines (M, N) are *linked* if the following are true:

1. The identity of M is distinct from the identity of N.
2. The switch tapes of M and N coincide (i.e., writing to one affects the value in both).

3. The read-only communication tape of M coincides with the write-only communication tape of N.

4. The read-only communication tape of N coincides with the write-only communication tape of M.

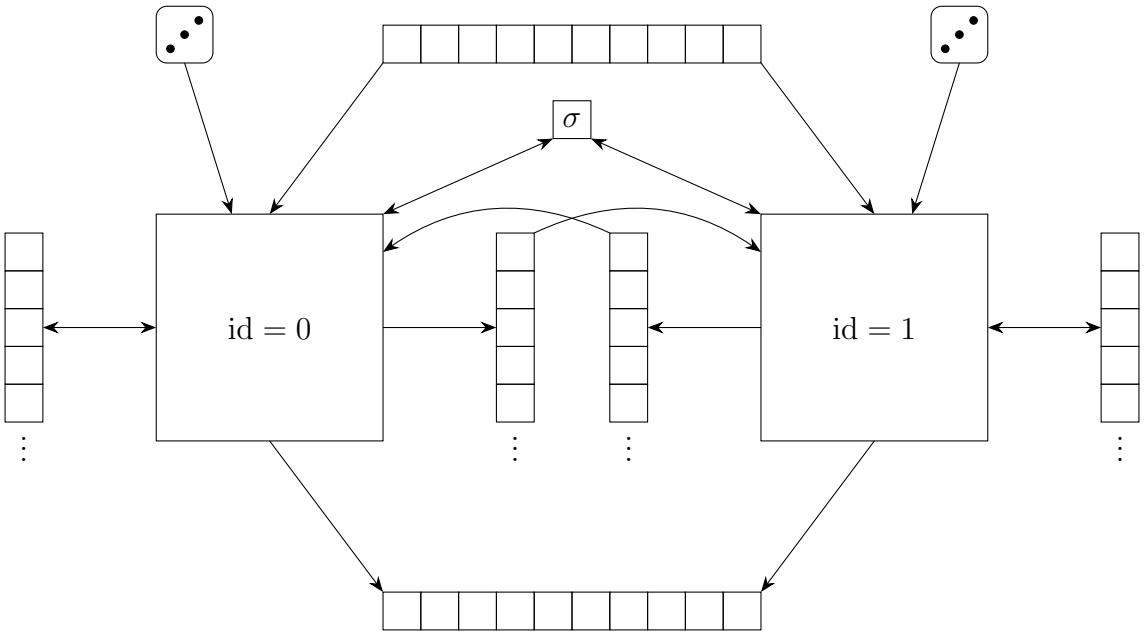

Figure 4.2: A linked pair of interactive Turing machines

We include a diagram of how a linked pair of Turing machines interact and share tape as Figure 4.2. The arrows point in the direction data is able to flow: read-only tapes have an arrow pointing from them and write-only tapes have an arrow pointing to them.

Definition 4.1.3 ([14, Def. 4.2.2])**.** The *joint computation* of a linked pair of interactive Turing machines (M, N) is, on a common input string x, the series of computation states for both M and N when each is given x as its initial input tape and when the initial value of the shared switch tape is 0. The joint computation halts when either machine halts and the halting machine is not idle.

We will denote the joint computation of machines M and N on input x by $\langle M, N\rangle(x)$. Since this output is not deterministic (it will depend on the values of the random bits read by P and V), it is important to note that this is a random variable, not an individual value.

TODO:

Figure 4.3: The flow of a joint computation of two interactive TMs

Finally, we need the concept of a "view". A view is in essence a record of what a machine in a given interaction sees: it is an ordered list of everything the machine reads in sequence. We will care quite a bit about views throughout this thesis since views are a record of the "public" information of a proof: when we begin to work with zero knowledge, we will be using the view of an interaction in order to show that no information is leaked.

Definition 4.1.4. The *view* of M in a joint computation on input string x is the sequence $(x, r, m_1, \ldots, m_n)$, where x is the input string, r is the sequence of random bits seen by M, and m_i are the random messages recieved by M from N. We will denote the view of M by $\text{View}_M^N(x)$.

4.2 Single-prover systems

Now that we have a model for letting two machines talk to each other, we can define the requirements for an interactive proof. We will begin with the simplest form—that where there is only one prover and one verifier. To make sure our proof system is useful, we need three restrictions on the machines: to restrict the complexity of the verifier (lest it simply compute the problem itself without communication), to require the verifier to generally accept whenever the input is in the language, and to require the verifier to generally reject whenever the input is not in the language.

Definition 4.2.1 ([14, Def. 4.2.4])**.** An *interactive proof system* is a pair of interactive machines (P, V) such that V is polynomial-time and the following holds:

- *Completeness*: For every $x \in L$,

$$\mathbb{P}[\langle P, V\rangle(x) = 1] \geq \frac{2}{3}.$$

- *Soundness*: For every $x \notin L$ and every interactive machine B,

$$\mathbb{P}[\langle B, V\rangle(x) = 1] \leq \frac{1}{3}.$$

While we require our system to be correct at least 2/3 of the time, our choice of probability is actually somewhat arbitrary, so long as it is at least 50%. This is because with a greater than 50% chance of success, we can simply run the checker multiple times and take the majority vote, which will allow us to get the probability arbitrarily high. Since this iteration is for a fixed number of times, it will only linearly scale the runtime and thus it does not affect whether our algorithm is an interactive proof system.

For the soundness clause, note that we require the inequality to hold for *any* interactive machine B, and not just our chosen machine P. This is important—it says that our verifier cannot be fooled reliably by a dishonest machine, so long as

	$B = P$	$B \neq P$
$x \in L$	$\mathbb{P} \geq \frac{2}{3}$	¯_(ツ)_/¯
$x \notin L$	$\mathbb{P} \leq \frac{1}{3}$	$\mathbb{P} \leq \frac{1}{3}$

Figure 4.4: Probability matrix for IP acceptance given prover and input string

x is not in the language L. In practice, what this means is that if the verifier has reason to believe that the machine it is interacting with is not P, then it should always reject immediately, as we do not care what happens with an arbitrary machine when $x \in L$. A consequence of this is that if V ever recieves back improperly-formatted or nonsense input from its prover, it will reject immediately. Similarly to what we do for ordinary Turing machines parsing their input, we will not explicitly write out that V should reject if it recieves a poorly-formatted response, as it serves little but to provide clutter.

We also do not care how V fares if $x \in L$ and P is not the correct verifier. This is because neither insisting the protocol fail or insisting the protocol succeed will be a reasonable restriction. Since $x \in L$, an alternative P' could give messages arbitrarily close to the correct P (and in some cases even send identical messages, which would be impossible to distinguish), so we cannot insist L reject, even with high probability. However, if V could accept even in the face of an arbitrarily malicious prover, or a prover that sends no useful information whatsoever, it would mean that V would have some means of computing the problem on its own; hence we would only ever be able to accept languages in BPP (since L is a BPP machine).

As with all of our interactive-proof variants, we will also define a complexity class corresponding to the set of languages with the given proof. Once we have a complexity class, we will be able to work with it in the same way we have been all the "standard" classes like P or NSPACE.

Definition 4.2.2 ([14, Def. 4.2.5])**.** The class IP is the class of all languages that have an interactive proof system.

TODO:

Definition 4.2.3. The *view* of an interactive Turing machine M_1 communicating with M_2 is the tuple $(x, m_1, \ldots, m_n)$, where m_i is the ith message recieved from M_2.

Now that we have seen the formal definition of an interactive proof, let us illustrate the formality with an example. To do so, consider the language of non-isomorphic graphs:

TODO: Move GI and GNI definition to Chapter 1?

Definition 4.2.4. The language GI (for *graph isomorphism*) is the set

$$\mathsf{GI} = \{(G_0, G_1) \mid G_0, G_1 \text{ graphs and } G_0 \cong G_1\}.$$

This language is interesting for reasons beyond the scope of this paper, especially in that it is known to be in NP but is believed to be neither in P nor NP-complete.

Definition 4.2.5. The language GNI (for *graph non-isomorphism*) is the set

$$\mathsf{GNI} = \{(G_0, G_1) \mid G_0, G_1 \text{ graphs and } G_0 \ncong G_1\}.$$

Here, we will demonstrate that GNI has an interactive proof.

FIXME: I think I want GI here (also in IP since I'm pretty sure IP = *co*IP); perhaps I should include both since the GI IP isn't particularly interesting (GI ∈ NP so P can just send the isomorphism)

Theorem 4.2.6. *The language* GNI *is in* IP.

Input: Two n-vertex graphs G_0 and G_1, and a security parameter s
Output: Whether $G_0 \ncong G_1$

if $|V(G_0)| \neq |V(G_1)|$ *or* $|E(G_0)| \neq |E(G_1)|$ **then**
  accept;
end
for $i \in [s]$ **do**
  V: Pick a random $\sigma_i \in S^n$;
  V: Pick a random $b_i \in \{0, 1\}$;
  V: Compute $H_i \leftarrow \sigma_i \cdot G_{b_i}$;
  V: Send H_i to P;
  $r_i \leftarrow$ **if** $H_i \cong G_0$ **then**
    P: Send 0 to S;
  else
    P: Send 1 to S;
  end
  if $r_i \neq b_i$ **then**
    reject;
  end
end
accept;

Algorithm 4.1: An interactive proof for the language GNI

Proof. We present an interactive protocol for GNI in Algorithm 4.1. In addition to the two graphs, we also give this algorithm one metaparameter s: the *security parameter*. This parameter does not need to be dynamic; adjusting it only affects the number of rounds of the protocol and correspondingly the probability of outputting the correct value.

First, we show V runs in polynomial time. Picking a random permutation and single bit can be done in polynomial time, and computing the action of a permutation on a graph is also polynomial.

Next, if $G_0 \ncong G_1$ and P is the honest prover, we show V will accept with probability $\geq 2/3$. Since $G_0 \ncong G_1$, it follows that $\sigma \cdot G_0 \ncong \sigma' \cdot G_1$ for any permutations σ and σ'. Hence, the honest prover will always answer with the correct $r_i = b_i$, and thus V will always accept.

If $G_0 \cong G_1$, we show V will reject with probability $\geq 2/3$, regardless of the prover. For any permutations σ and σ', we have that $\sigma \cdot G_0 \cong \sigma' \cdot G_1$, by transitivity of isomorphisms. Further, we have that S_n is exactly the class of isomorphisms on n-vertex labeled graphs, so for any isomorphic graph G, there exists exactly one σ_0 and σ_1 such that $\sigma_0 \cdot G_0 \cong G \cong \sigma_1 \cdot G_1$. Hence, the odds of b_i being 0 are 1 are equal for any given G. Thus, in each round the odds of P guessing correctly are no more than $1/2$. Further, in each round the random picks are statistically independent; hence the overall odds of P fooling V are no more than 2^{-s}. For any $s > 1$, $2^{-s} < 1/3$; hence V will reject with probability $\geq 2/3$. □

Once we have a complexity class, the question arises of how it relates to other complexity classes. For IP, Adi Shamir proved in 1992 [30] that a language has a standard interactive protocol if and only if it is in PSPACE.

TODO: How much do I want to talk about this? (Scope creep)

Theorem 4.2.7 ([30])**.** IP = PSPACE.

Proof. TODO: □

4.3 Multi-prover systems

We have now seen quite a bit of single-prover interactive proofs. A natural extension of the standard interactive proof format is to add more machines to the interaction. Since our verifiers are trusted, increasing the number of verifiers is not useful since any pair of verifiers could simply be simulated with a single verifier working twice as hard (which would keep it polynomial). However, increasing the number of provers to two turns out to give us more power than we would get with a single prover.

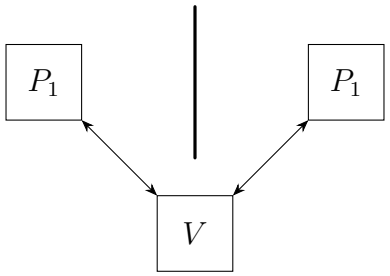

Figure 4.5: A multi-prover interactive system

Definition 4.3.1 ([14, Def. 4.11.2])**.** A *multi-prover interactive proof system* is a triplet of interactive machines (P_1, P_2, V) such that P_1 and P_2 cannot communicate, V is probabilistic polynomial-time, and the following hold.

- *Completeness*: For every $x \in L$,
$$\mathbb{P}[\langle P_1, P_2, V\rangle(x) = 1] \geq \frac{2}{3}.$$
- *Soundness*: For every $x \notin L$ and every pair of interactive machines B_1 and B_2,
$$\mathbb{P}[\langle B_1, B_2, V\rangle(x) = 1] \leq \frac{1}{3}.$$

The above definition should look rather similar to Definition 4.2.1; the only difference is now we have two provers instead of just one. The fact that the two provers cannot communicate is important: if they could, they would be able to "strategize"; that is, agree on a joint plan to make sure that their responses agree with each other. Since our provers are not required to be computationally bounded, if they could communicate it would be no different than simply having one prover. However, since the two provers cannot communicate, we gain some information from times where they lie in *different* ways: where each prover individually could say something plausible, but in combination, the provers' responses contradict.

Later on, we will see what can happen if the computers are allowed a limited form of communication: the class MIP$*$, which uses quantum provers that can share entangled bits, will be defined in Section 6.3 and then explored in more detail in Chapter 7.

TODO:

Definition 4.3.2. The *view* of a multi-prover interactive system is TODO:

TODO:

Definition 4.3.3. The class MIP is the class of languages that have a multi-prover interactive proof system.

The first question about a class like MIP is how it relates to other complexity classes we have already seen. First, if we have a single-prover system we can always convert it into a multi-prover system by simply having the verifier ignore P_2 completely; thus we get the following.

Lemma 4.3.4. IP $\subseteq$ MIP.

Proof. Let (P, V) be an interactive proof system. Consider the system (P_1, P_2, V'), where $P_1 = P$, and V' simulates V and sends all its messages to P_1. If $x \in L$, then $\langle P_1, P_2, V'\rangle(x)$ will accept with exactly the same probability as $\langle P, V\rangle(x)$. Since (P, V) is an interactive proof system, it follows that it will accept with probability at least $2/3$. If $x \notin L$, then $\langle P_1^*, P_2^*, V\rangle$ will reject with exactly the same probability as $\langle P_1^*, V\rangle$ regardless of the value of P_2^*, since the interaction is exactly the same and since we ignore the second prover completely. Again, since (P, V) is an interactive proof system, it follows that it will reject with probability at least $2/3$ in this case. Hence, (P_1, P_2, V') is a MIP system. □

A groundbreaking result by Babai, Fortnow, and Lund [5] is that MIP is exactly equal to NEXP. Since it is known that NP $\neq$ NEXP [12], this tells us that adding multiple provers gives an actual boost in computational power over just having one.

Theorem 4.3.5 ([5])**.** MIP = NEXP.

Proof. TODO: How much effort do I want to put into this? (Scope creep) We showed in Theorem 1.2.21 that O3SAT is NEXP-complete, so all we need is to demonstrate that O3SAT has a multi-prover system. □

Having seen how much more powerful systems become with two provers, one might wonder what would happen if we were to add a third. Unfortunately, it turns out that a third prover is no more powerful than just having two. We formalize this below; because we do not get any benefit from three provers we will not work with three-prover systems at all in this paper beyond this proof.

Theorem 4.3.6 ([8, Theorem 4])**.** *If we redefine* MIP *to have k provers instead of 2, the class is unchanged.*

$\hat{V}$: Generate all random bits and send results to $\hat{P}_1$;
$\hat{P}_1$: Send transaction log between $(P_1, \ldots, P_k, V)$ with the chosen randomness;
if *The simulated log is longer than the worst-case runtime of V* **then**
  reject;
end
$\hat{V}$: Choose random $i \in [k]$;
$\hat{V}$ and $\hat{P}_2$: Simulate conversation between V and P_i given coin tosses;
if *the simulated conversation does not match the result of $\hat{P}_1$* **then**
  reject;
else
  accept if and only if V would accept with the given transcript;
end

Algorithm 4.2: A 2-prover MIP simulating a k-prover MIP

Proof. Let $(P_1, \ldots, P_k, V)$ be a MIP with k provers. We show a simulator for $(P_1, \ldots, P_k, V)$ in Algorithm 4.2. Since V runs in polynomial time and all $\hat{V}$ does is look at the simulated transaction log of V (twice), it follows that $\hat{V}$ runs in polynomial time.

Let $x \in L$. By definition, we know that

$$\mathbb{P}[\langle P_1, \ldots, P_k, V\rangle(x) = 1] \geq \frac{2}{3}. \tag{4.1}$$

If $\hat{P}_1$ and $\hat{P}_2$ are honest, then the simulated conversations will match; hence $\hat{V}$ will reject if and only if P would have. Thus, $\mathbb{P}[\langle \hat{P}_1, \hat{P}_2, \hat{V}\rangle = 1] \geq 2/3$.

Let $x \notin L$, and let $\hat{P}_1^*$ and $\hat{P}_2^*$ be arbitrary verifiers. From the definition of MIP, for at least $2/3$ of the choices of randomness, the generated transcript would result in V rejecting; hence $\hat{P}_1$ must deviate from it somewhere. Since $\hat{P}_1$ deviates from the protocol at least once, it follows there is at least a $1/k$ chance that the simulated conversation between $\hat{V}$ and $\hat{P}_2$ is different from what $\hat{V}$ recieved from $\hat{P}_1$. Hence, the probability of correctly rejecting here is at least $2/3k$.

Note that if the two provers ever disagree, it must be that at least one of them is a cheating prover. Thus, if we run it at least k^2 times we will have at least one run where we catch the cheating with probability $2k^2/3k = 2/3$. Thus, we will correctly reject with probablity at least $2/3$ regardless of the cheating prover. Hence, (P_1, P_2, V) is a valid MIP system. □

4.4 Zero-knowledge proofs

Zero-knowledge proofs are a variant of interactive proofs that have certain cryptographic requirements. What we care about is the idea that zero-knowledge proofs transmit *no knowledge* other than precisely the statement trying to be proved. As an example, if the statement that you are trying to prove is "I have an instance of X", the conceptually-easiest way to prove it would be to produce the aforementioned instance. However, this would not be zero-knowledge since it also transmits the knowledge of exactly what your instance of X is.

The way we mathematically define zero-knowledge is a little tricky. The way we demonstrate that the proof is zero-knowledge is by creating a simulator S_V for each possible verifier V: a machine in P that *by itself* can reproduce the entire message log between P and V for any input.

This definition shows that no knowledge has been released because we are able to reproduce all the public information of the proof with relatively little work. If non-trivial knowledge were released by the proof, we would not be able to recreate the interaction faithfully without access to the prover.[1]

Having said that, it is not particularly obvious that there are any languages that are outside of P with perfect zero-knowledge proofs.[2] It turns out, however, that these languages do in fact exist (and are reasonably common). Abstractly, the idea behind why many of these work is that the verifier can perform a transformation on some random value, such that undoing the transformation and reliably recovering the original value is only possible with knowledge of the language. However, a simulator would have access to the randomly-chosen value, and thus it could construct a response immediately with no reference to the problem to be solved.

Definition 4.4.1 ([14, Def. 4.3.1])**.** A proof system (P, V) for a language L is *perfect zero-knowledge* if for each probabilistic polynomial-time interactive machine V^* there

[1] Exception: if $L \in \mathsf{P}$ then we can trivially recreate the interaction no matter what, but that case is not particularly interesting for the purpose of zero-knowledge proofs.

[2] To some extent, showing that there are languages *truly* outside of P would require a proof that $\mathsf{P} \neq \mathsf{NP}$ (which is unfortunately beyond the scope of this paper), but there are lots of languages strongly believed to be outside of P with zero-knowledge proofs.

exists a probabilistic polynomial-time ordinary machine M^* such that for every $x \in L$ we have the following conditions hold:

1. With probability at most $1/2$, on input x, machine M^* outputs a special symbol denoted $\perp$ (i.e. $\mathbb{P}[M^*(x) = \perp] \leq 1/2$).

2. Let $m^*(x)$ be the random variable such that

$$\mathbb{P}[m^*(x) = \alpha] = \mathbb{P}[M^*(x) = \alpha \mid M^*(x) \neq \perp] \tag{4.2}$$

   for all α. That is, let $m^*(x)$ be the distribution of non-$\perp$ values of M^*. Then $\langle P, V^* \rangle(x)$ and $m^*(x)$ are identically distributed for all $x \in L$.

In this case, we say the machine M^* is a *perfect simulator* for the interaction of V^* with P.

Looking at this definition, one might wonder why we would want the ability for M^* to output $\perp$. This is a reasonable thing to wonder, because we do not actually want this. However, we do not know of any non-trivial proof systems that do not actually output $\perp$ at least some of the time, although [15] has made significant inroads on this problem.[3] Thankfully, we can make $\mathbb{P}[\perp]$ aribitrarily small (bounded above by $2^{-\mathsf{poly}(|n|)}$), but we cannot make it truly perfect. We can do this by simply re-running the simulator every time we get a $\perp$ until we get a valid answer.

Also note that while the definition of interactive proof systems focus on cheating *provers*, the definition of zero-knowledge focuses on cheating *verifiers*. This is because we can think of the two as being resilient to different threat models. For interactive proofs, we care about verifier efficiency: our goal is to show it is possible for some computer with unbounded resources to easily convince a verifier of something, and to show that no other equally-strong computer could lie.

For zero-knowledge, our main goal is to ensure tha it is impossible for anybody except for the honest verifier to extract any information from the honest prover. The main way we do this is by ensuring that it is impossible for anybody snooping on the transaction to gain any information (hence the simulator), but we also want to ensure that the prover cannot be tricked into revealing information by a dishonest verifier. We do not care about what happens with a dishonest prover in this case because alternate provers could in theory reveal anything—there is always a prover that just dumps any relevant information to the interaction tape, for example.

As with other interactive proof systems, zero-knowledge proofs are probabilistic; in particular this means they do *not* function as proofs in the mathematical sense.

Definition 4.4.2 ([14, Def. 4.3.5])**.** The class PZK is the class of all languages with a perfect zero-knowledge proof system.

To demonstrate perfect zero-knowledge, we now show an example. In Section 4.2, we demonstrated a non-zero-knowledge interactive proof for the language GNI of non-isomorphic graphs. Here, we modify that algorithm to not reveal anything beyond simply whether G_0 and G_1 are isomorphic.

[3] More specifically, they have shown that all of NP has a zero-knowledge proof without use of $\perp$, which includes nontrivial problems in the (generally expected) case that $\mathsf{NP} \neq \mathsf{BPP}$.

Theorem 4.4.3. *The language* GI *has a perfect zero-knowledge interactive proof.*

Input: Two n-vertex graphs G_0 and G_1
Output: Whether $G_0 \cong G_1$
1 **for** $i \in [s]$ **do**
2 P: pick a random $\sigma \in S^n$;
3 P: pick a random $b \in \{0, 1\}$;
4 P: send $\sigma \cdot G_b$ to V;
5 V: pick a random $b' \in \{0, 1\}$;
6 V: send b' to P;
7 P: compute $\sigma' \in S^n$ such that $\sigma' \cdot G_{b'} = \sigma \cdot G_b$;
8 P: send σ' to V;
9 **if** $\sigma' \cdot G_{b'} \neq \sigma \cdot G_b$ **then**
10 **reject**;
11 **end**
12 **end**
13 accept;

Algorithm 4.3: A perfect zero-knowledge IP for GI

Proof. We present the algorithm as Algorithm 4.3. Our proof will consist of two stages: first, we will prove that the algorithm is a functional interactive proof for GI, and then we will show that it does not leak any knowledge.

First, if $G_0 \cong G_1$ and P is honest, then regardless of what parameters we pick, there will always exist a σ' we can compute in line 7. Hence, the condition in line 9 will never be true and hence we will always accept.

If $G_0 \ncong G_1$, then whenever $b' \neq b$ there exists no σ' such that $\sigma' \cdot G_{b'} = \sigma \cdot G_b$. Hence, if $b' \neq b$ (which happens with probability $1/2$) then regardless of what σ' is sent to V the check in line 9 will fail and hence V will reject. Since b' and b are re-rolled in each round, the probability of them being equal in all s rounds is 2^{-s}. Hence, V will accept with probability no more than 2^{-s}.

Now, we show zero-knowledge. To do this we show a simulator in Algorithm 4.4. To summarize, it attempts to simulate a single round of the interaction repeatedly until it has s successes; if it cannot do this in $2s$ tries, it outputs $\perp$.

First, we show Algorithm 4.4 runs in polynomial time. We know V^* runs in polynomial time by our hypothesis; hence line 11 runs in polynomial time. All the rest of the lines are either random choice of items, computing permutations, or simple arithmetic; all of these are also doable in polynomial time. Hence the interior of the loop runs in polynomial time, and since we iterate no more than $2s$ times, it follows that the whole algorithm is polynomial time.

Note that an individual round of this simulator can only succeed when $b = b'$: if it could output a correct response σ' when $b \neq b'$, then it would have $\sigma' \circ \sigma^{-1}$ as an isomorphism from G_0 to G_1, and thus Algorithm 4.4 would be a probabilistic

```
c ← 0;
L ← [ ];
for i ∈ [2s] do
    if c ≥ s then
        return L;
    end
    Choose random σ ∈ S_n;
    Choose random b ∈ {0, 1};
    H ← σ · G_b;
    Add H to L;
    Simulate V* on input (G_0, G_1) and recieved message H until it sends a
     message σ';
    if b = b' then
        c ← c + 1;
        Add σ to L;
    end
end
return ⊥;
```

Algorithm 4.4: A simulator for Algorithm 4.3

polynomial-time determinier for GI. Since we do not know whether or not $\mathsf{GI} \in \mathsf{BPP}$, we do not yet know how to construct any verifier that would do this.

Since V^* never recieves b and b is randomly generated, if G_0 is isomorphic to G_1, for any value of G it is just as likely that $b = 0$ as it is that $b = 1$. More formally, for all G,

$$\mathbb{P}[G \mid b = 0] = \mathbb{P}[G \mid b = 1].$$

As such, for any $(G_0, G_1) \in \mathsf{GI}$, it is impossible for any V^* to send $b' \neq b$ with probability more than $1/2$.

Since, each individual round succeeds with probability at least $1/2$, the binomial theorem tells us that the probability of exactly k successes in $2s$ tries is $\binom{2s}{k}/2^{2s}$. Hence, the probability of at least s successes in $2s$ tries is

$$\frac{1}{2^{2s}} \sum_{i=s}^{2s} \binom{2s}{i}. \tag{4.3}$$

Since $\binom{2s}{k} = \binom{2s}{2s-k}$ and the sum of $\binom{2s}{k}$ over all k is $2s$, it follows that Equation (4.3) is equal to

$$\frac{\sum_{i=s}^{2s} \binom{2s}{i}}{\sum_{i=1}^{2s} \binom{2s}{i}} = \frac{s + \sum_{i=s+1}^{2s} \binom{2s}{i}}{2 \sum_{i=s+1}^{2s} \binom{2s}{i}} \geq \frac{1}{2}. \tag{4.4}$$

Hence, the algorithm will output $\perp$ with probability no more than $1/2$.

Next, we show Algorithm 4.4 outputs an identically-distributed view to what V^* sees for all $x \in \mathsf{GI}$, regardless of what V^* is. We show that it outputs an identical view for a single round; since the simulator will simulate s rounds it follows that the total result will be identical exactly when an individual round is.

An individual round of Algorithm 4.3 has a total of three messages sent: two from P and one from V. The view of a verifier only consists of the messages from P, so after each round we should be adding two items to the log. Our honest prover sends two messages: first, a random graph $H \cong G_b$ and second, an isomorphism σ that maps $G_{b'}$ to H.

The simulator picks a random b and H in exactly the same way as P; thus H will be identically distributed in the simulator as it is in the original algorithm. Next, remember that we only care about the views being identical in the case where $(G_1, G_2) \in \mathsf{GI}$, that is, where $G_1 \cong G_2$. In this case, picking a random isomorphic copy of G_1 will give you an identically-distributed random variable to picking a random isomorphic copy of G_2. Similarly, $H \cong G_1 \cong G_2$ by construction, so picking a random isomorphic copy of H will also give you an identical distribution. As such, σ is a random isomorphism and hence the transaction (H, σ) is distributed identically to the originally-generated transcript. □

4.4.1 Commitment schemes

It should not come as too much of a surprise to learn that most intuitive proofs for a given problem are not in fact zero-knowledge.[4] As such, we will want the assistance of a few techniques that, once understood, will allow us to build zero-knowledge proofs more easily. The first of these is a *bit-commitment scheme*.

Abstractly, a bit-commitment scheme allows a machine to "commit to" a given single bit, with the intent of revealing it later on to a verifier. To do this, we need two important things: first, that the bit is not revealed to the verifier at commitment time, and second, that if the revealed bit is not equal to the committed bit, the verifier will be able to know (that is, the committer will not be able to change its choice once it has committed).

Before we can define a bit-commitment scheme formally, we do need a few preliminaries. First, we define what it means for an interaction to look like a commitment from the reciever's perspective; since the reciever needs to be convinced of the commitment, we need a definition that only considers its perspective.

Definition 4.4.4. Let $\sigma \in \{0, 1\}$. A reciever's view of an interaction $(x, r, m_1, \ldots, m_n)$ is a *possible σ-commmitment* if there exists a string s such that m_i describes the messages received by R when R uses local coins r and interacts with machine S that uses local coins s and has input $(\sigma, 1^n)$.

The above definition does not preclude a series of messages looking like it could be both a 0-commitment and a 1-commitment; consider the case of a machine S that

[4] As an example, I would certainly hope that the proofs in this text have, in fact, imparted at least some knowledge on the reader.

completely ignores its input and sends the same series of strings. In this case, the record of that interaction would be a possible commitment for any input, since the input is ignored completely.

Definition 4.4.5. A reciever's view is *ambiguous* if it is both a possible 0-commitment and a possible 1-commitment.

Now, we can define a bit-commitment scheme. In brief, a bit-commitment scheme is a scheme such that there are no ambiguous views and yet the total publicly-released information is ambiguous.

Definition 4.4.6 ([14, Def. 4.4.1])**.** A *bit-commitment scheme* is a pair of interactive probabilistic polynomial-time machines (S, R), such that

1. Both machines recieve an integer n in unary,
2. S recieves a single bit v,
3. For any PPT machine R', the output of $\langle S(0), R'\rangle(1^n)$ and $\langle S(1), R'\rangle(1^n)$ are computationally indistinguishable over all inputs n, and
4. For almost all coin tosses of R, there exists no sequence of messages from S such that the view of R is ambiguous.

TODO: More explanation

FIXME: The existence of bit-commitment schemes implies the existence of one-way functions (Go01 Exercise 13), but CFGS22 talks about *algebraic* commitment schemes without reference to one-way functions (nor do they mention $\mathsf{P} \neq \mathsf{NP}$)

The simplest examples of bit-commitment schemes involve one-way functions. A one-way function is a function that is computable in polynomial time, but whose inverse is *not* computable in polynomial time. These functions are not known to exist: in particular their existence implies $\mathsf{P} \neq \mathsf{NP}$, but they are still widely believed to exist.

```
   /* Commit                                                    */
1  TODO: ;
   /* Reveal                                                    */
2  TODO: ;
```

Algorithm 4.5: A bit-commitment scheme based on a one-way function f

TODO: Define decommitment

4.5 Probabilistically-checkable proofs

So far, all of our computational proofs have focused on the interaction between two computers, but there exist non-interactive models as well. Probabilistically-checkable proofs do not use interactive Turing machines, but instead have access to a "proof"

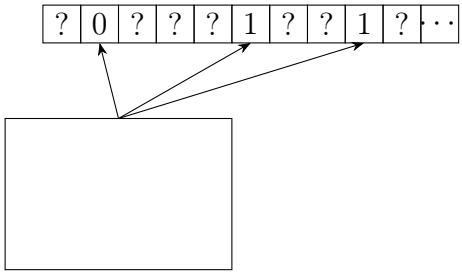

Figure 4.6: A probabilistically-checkable proof verifier

that their input is in the given language. The nontriviality is that the number of bits of the proof we can access is bounded—simply reading the entire proof will not suffice. For any string in the language, we ensure there exists a correct proof, which our algorithm must always recognize accurately. Further, for any string *not* in the language, the algorithm must reliably (but not necessarily always) reject.

In general, query-based machines (i.e. those where the *number* of queries is an important part of the complexity) come in two flavors: adaptive and non-adaptive. Adaptive-query machines allow the machine to change what locations it queries based on what it has already seen; non-adaptive machines do not allow that. In general, adaptive queries are more powerful—it turns out that any adaptive machine can be simulated by a non-adaptive machine using 2^q queries. Most interesting results about PCPs can be proven even with weaker non-adaptive machines, so that is what we will focus on for the rest of this paper.

Definition 4.5.1. Let M be a Turing machine with query access to some string $\pi \in \{0,1\}^*$. The queries by M are *non-adaptive* if the locations queried depend only on the contents of the input tape and random generator. If the queries are dependent on previous query results, then M is *adaptive*.

TODO:

Definition 4.5.2 ([2, Def. 18.1])**.** Let $L \subseteq \{0,1\}^n$ be a language and $q, r \colon \mathbb{N} \to \mathbb{N}$. A $(r(n), q(n))$-*verifier* for L is a polynomial-time probabilistic algorithm V such that

1. When given an input string $x \in \{0,1\}^n$ and random access to a string $\pi \in \{0,1\}^*$, V uses at most $r(n)$ random coins and makes at most $q(n)$ non-adaptive queries to locations of π before either accepting or rejecting.

2. If $x \in L$ then there exists a $\pi_x \in \{0,1\}^*$ such that V will always accept when given input x and random string π_x.

3. If $x \notin L$ then V will reject with probability $\geq 1/2$ for *all* random strings π.

We call the random string π the *proof*. We denote the output of V on input x and proof π with $V^\pi(x)$.

The first piece of this definition to notice is that if $x \in L$ we require L to accept unconditionally for x's corresponding proof, despite the fact that L has access to randomness. We want this because we want to construct verifiers that look for inconsistencies in the purported proof and reject if they find them—a correct proof should not contain any inconsistency at all so we should accept every time.

On the other hand, while an incorrect proof should still be noticeable *most* of the time (otherwise this would not be a particularly useful definition), if we are only querying a limited number of bits from a proof it is impossible to prevent scenarios where we only query bits that are identical to the original proof. In this case, it would be impossible for any algorithm to distinguish correct proofs from incorrect ones (outside of ignoring the proof completely and just solving the problem itself) and as such we do not require V to always correctly reject.

So far, all of our proof-complexity classes have just had a single class for all languages with the proof regardless of internal complexity, but for probabilistically-checkable proofs we actually stratify the class further. This is for multiple reasons: first, we can actually get astonishingly tight bounds on the parameters for PCPs (as we will see in Theorem 5.0.1), and second, because these are "access" complexity (i.e. we measure the number of preexisting bits actually read by the algorithm), they are actually independent of computational model, so the need for polynomial equivalence is negated.

In addition, this means PCPs become susceptible to the alphabet the proof is written in. Since individual bits carry more data when a larger alphabet is used, a larger alphabet will necessarily require fewer queries to transmit the same information. As such, while we will by default still work over the alphabet $\{0, 1\}$, there are times where we will need to be a little more specific about the alphabet we use.

Definition 4.5.3 ([2, Def. 18.1])**.** For any $q, r \colon \mathbb{N} \to \mathbb{N}$, the class $\mathsf{PCP}_\Sigma(q(n), r(n))$ is the class of all languages with a $(cq(n), dr(n))$-verifier for some $c, d \in \mathbb{N}$, where the proof is written over the alphabet Σ. When $\Sigma = \{0, 1\}$, we sometimes omit it.

A small note on notation: The text "PCP" can be used as both a complexity class and an abbreviation: in this paper when it is written PCP, we are referring to the complexity class (and therefore the set of *languages* with a probabilistically-checkable proof); when it is written PCP, we are referring to the proofs themselves.

Next, we give an exapmle of a nontrivial probabilistically-checkable proof. As with our previous examples, the language GNI provides a good example for us, as it is a relatively simple language that is still not known to be in BPP.

Theorem 4.5.4. $\mathsf{GNI} \in \mathsf{PCP}(\mathsf{poly}(n), 1)$.

Proof. We describe such a PCP in Algorithm 4.6. Next, we show that this algorithm has the properties we seek.

The verifier runs in polynomial time: both picking and computing an n-bit permutation are known to be in polynomial time with respect to n. Further, picking a random n-bit permutation is doable in $\mathsf{poly}(n)$ bits, and picking a random bit is

Input: Two n-vertex graphs G_1 and G_2
Output: Whether $G_1 \ncong G_2$
/* Proof: */
1 **for** *H a graph with n nodes* **do**
2 **if** $H \cong G_0$ **then**
3 $\pi[H] \leftarrow 0$;
4 **else**
5 $\pi[H] \leftarrow 1$;
6 **end**
7 **end**
8 **return** π;
/* Verifier: */
9 Pick random $b \in \{0, 1\}$;
10 Pick random $\sigma \in S_n$;
11 Apply σ to the vertices of G_b;
12 Accept if and only if $\pi[\sigma \cdot G_b] = b$;

Algorithm 4.6: A PCP for GNI

doable in 1 bit; hence V uses $\mathsf{poly}(n)$ bits of randomness. Lastly, we only make a single query to π, in line 12.

If $G_0 \ncong G_1$, we show V always accepts when given π as input. Since isomorphisms are transitive, we know there is no H with both $H \cong G_0$ and $H \cong G_1$. Hence, for all H, if $H \cong G_0$ then $\pi[H] = 0$ and if $H \cong G_1$ then $\pi[H] = 1$. Hence, since $\sigma \cdot G_b \cong G_b$ regardless of our choice of σ and b, we have that $\pi[\sigma \cdot G_b] = b$.

Next, if $G_0 \cong G_1$ then V rejects with probability at least $1/2$, regardless of choice of π. Since $G_0 \cong G_1$, it follows that if $H \cong G_0$ then $H \cong G_1$, and vice versa. Hence, for any graph $H = \sigma \cdot G_0$, there exists a $\sigma' \in S_n$ with $H = \sigma' \cdot G_1$. For any n-vertex graph H, this means that we are equally likely to query $\pi[H]$ with $b = 0$ and σ as we are with $b = 1$ and σ'. Since we know $\pi[H]$ can only be 0 or 1, it must be the case that $\pi[H]$ is incorrect at least half the time. Hence, V will reject with probability at least $1/2$ for any proof π. □

It is reasonable to be surprised about the fact that we only need one query to determine this problem to the constraints imposed by a PCP. This is our first clue that PCPs are surprisingly powerful: in Chapter 5 and again in Chapter 10 we will explore the extremes of the power of PCPs.

4.5.1 PCPs of proximity

Since a PCP will always only check a limited proportion of any given proof, for any strings in the language, the verifier will still accept any proof that is close to the official proof with very high probability. While this is interesting in and of itself, it can also lead to the question of how PCPs perform on *values* that are close to strings in our language. This is the notion behind PCPs of proximity: what if we weaken a

PCP to only require it to reject strings that are not sufficiently close to strings in the given language?

To define PCPs of proximity, we first need a working definition of proximity. Computer scientists have many definitions of string closeness, but for this particular problem we will be working with a mathematically simple one originally used for error-correcting codes.

Definition 4.5.5 ([18])**.** Let $x, y \in \Sigma^n$ be strings of the same length. We say the *Hamming distance* between x and y is the value

$$\Delta(x, y) = \frac{|\{i \in [n] \mid x_i \neq y_i\}|}{n}.$$

Note that $\Delta(x, y) \in [0, 1]$ for any strings x and y. This normalization is not strictly necessary in general, since Δ is only defined between strings of equal length (and there do exist cases where it is much nicer to keep the distance as a natural number). In our case, however, we would like to keep the distances all in the same bounded range; hence we normalize.

1	0	0	0	1	0	0	1
1	1	0	0	0	1	0	1

Figure 4.7: Two strings with Hamming distance 3/8

The definition of Hamming distance between two strings also generalizes to the notion of distance from a set. Informally, we say the distance from a string to a set is simply the distance to the closest element of the set.

Definition 4.5.6. Let $\varepsilon > 0$. A vector $x \in \Sigma^n$ is *ε-far* from a set $S \subseteq \Sigma^n$ if

$$\min_{y \in S}(\Delta(x, y)) \geq \varepsilon.$$

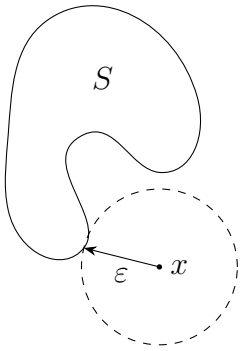

Figure 4.8: A point x that is ε-far from a set S

Now that we know what proximity means, we can define a PCP of proximity. Unlike a PCP, which is defined for any language L, a PCPP is only defined for *pair languages*, languages that consist entirely of ordered pairs of two objects. This is because we need our language to consist of pairs so that we can have a notion of distance between the elements. The good news is this will not affect us too much—lots of the languages we care about are pair languages already.

The main difference between a PCP and a PCPP is that we relax the rejection condition to only require consistent rejection for pairs (x, y) where there is no close y' where (x, y') is in the accepted language. There are many pair languages that we can think of as being made of function-value pairs (f, y) with the property that there is some x such that $f(x) = y$. When we have a language like this, we have a more intuitive explanation of PCPPs: here, a PCPP will accept any (f, y) pair where $f(x)$ is δ-close to y.

Definition 4.5.7 ([16, Def. 2.2])**.** For $\delta\colon \mathbb{N} \to [0, 1]$, a *probabilistically-checkable proof of proximity* for a language L consisting of ordered pairs (x, y) with proximity parameter δ consists of a prover P and verifier V such that the following holds for all (x, y):

1. When given an input string $x \in \{0, 1\}^n$ and random access to a string $\pi \in \{0, 1\}^*$, V uses at most $r(n)$ random coins and makes at most $q(n)$ non-adaptive queries to locations of π before either accepting or rejecting.

2. If $(x, y) \in L$, V will always accept when given input (x, y) and proof π_x.

3. If y is δ-far from the set $L(x) = \{y \mid (x, y) \in L\}$, then for every oracle π^*, V will reject on input (x, y) with probability $\geq 1/2$.

TODO: Introduce example

Theorem 4.5.8. $\mathsf{GI} \in \mathsf{PCP}[\mathsf{poly}(n), \mathsf{poly}(n)]$.

Proof. We demonstrate a PCP for GI with the specified parameters in Algorithm 4.7. We need to show a few things: that the proof and verifier follow the specified parameters, that the verifier always accepts when given a correct proof for some $(G_1, G_2) \in \mathsf{GI}$, and that the verifier reliably rejects when given a $(G_1, G_2) \notin \mathsf{GI}$.

First, an n-bit permutation can be described in $\mathsf{poly}(n)$ bits; for each bit in the permutation we write 3 bits to the proof, hence $|P| \in \mathsf{poly}(n)$. Since $|P| \in \mathsf{poly}(n)$, it follows that we can query at most $\mathsf{poly}(n)$ distinct bits. The verifier generates one random bit for every 3 bits in P, so it reads $\mathsf{poly}(n)$ bits of randomness. Finally, all the verifier does is reads $\mathsf{poly}(n)$ bits and then checks that it represents a permutation; this is all doable in polynomial time and thus the verifier is polynomial.

Next, we show V always accepts if $(G_1, G_2) \in \mathsf{GI}$ and $\pi = \pi_{(G_1,G_2)}$. The way we have designed our proof is a simple form of error-correcting code: we have just repeated the isomorphism three times and when reconstructing the isomorphism from the proof, we pick each bit from one of the three copies at random. In the event of

Input: Two graphs, G_1 and G_2
Output: Whether $G_1 \cong G_2$
`/* Proof */`
Let σ be an isomorphism from G_1 to G_2;
return (σ, σ, σ);
`/* Verifier */`
$\sigma' \leftarrow [\,]$;
for $i \in [|\pi|/3]$ **do**
  Pick random $r \in \{0, 1, 2\}$;
  Add $\pi[3r + i]$ to σ';
end
if σ' *is a representation of an isomorphism* $G_1 \to G_2$ **then**
  accept;
else
  reject;
end

Algorithm 4.7: A PCP for GI

an honest proof, all of these copies are equal to the original, so no matter what our choices of r, our reconstructed σ' will always be equal to σ (an isomorphism $G_1 \to G_2$ by definition) and thus V will accept.

Finally, we show V reliably rejects if $G_1 \ncong G_2$ regardless of what proof π is given to V. In this case, σ' will never be an isomorphism from G_1 to G_2 (since no such isomorphism exists); hence V will always reject. □

4.6 Zero-knowledge probabilistically-checkable proofs

A zero-knowledge probabilistically-checkable proof is a combination of the ideas of zero-knowledge proofs (as seen in Section 4.4) and probabilistically-checkable proofs (as seen in Section 4.5). Since we can model a PCP as an interaction between between a verifier and a proof (instead of a prover), we can model this interaction as being zero-knowledge as well.

Unlike interactive proofs, we cannot achieve *arbitrary* zero-knowledge guarantees: since the proof is non-interactive, we have no recourse against an attacker who simply reads the entire proof end-to-end. As such, we introduce a *query bound*: we limit our verifier to a certain number of queries, under which we retain the standard zero-knowledge restrictions.

Definition 4.6.1 ([17, Def. 8.6])**.** A probabilistically-checkable proof system is *zero-knowledge* with query bound q if for any verifier V' such that V' makes no more than $O(q(n))$ adaptive queries on an input of length n, there exists a probabilistic polynomial-time simulator S such that on input x, S can simulate every interaction of V' with the associated proof of x.[5]

[5]To clarify, S does *not* have access to the proof of x, just x itself.

The first thing to notice here is that for *validity* of PCPs, we parameterize over the proof (i.e. the one verifier should remain valid for all proofs π), for *zero-knowledge* we parameterize over the verifier V. This is because for zero-knowledge proofs we care about not revealing any information from the proof, no matter how clever we are about asking questions with the verifier. In that way, outside of the "happy path" (where $x \in L$ and the given string is the proof of x), the two notions are somewhat orthogonal: a PCP cares about how we react when $x \notin L$ but V is trusted, while zero-knowledge cares about what happens when V is not trusted, but the proof is.

Definition 4.6.2. The class PZK-PCP is the class of all languages that have a perfect zero-knowledge probabilistically-checkable proof.

TODO: Example (preferably GNI)

Theorem 4.6.3. *There is a perfect zero-knowledge PCP for* GNI.

1 TODO:

Algorithm 4.8: A PZK-PCP for GNI

Proof. TODO: □

4.7 Interactive probabilistically-checkable proofs

Interactive probabilistically-checkable proofs are a combination of the concepts of an interactive protocol and a probabilistically-checkable proof. The broad idea is our proof proceeds in two phases: first, the prover sends a purported proof to the verifier, after which they engage in an interactive protocol, during which the verifier can access the proof as an oracle.

Definition 4.7.1 ([24, §1.1])**.** Let L be a language, let $p, q, l \colon \mathbb{N} \to \mathbb{N}$, and let $c, s \in [0, 1]$. An *interactive probabilistically-checkable proof* for L is an interactive protocol as follows:

Input: To both P and V: a string x of length n
Input: To P alone: A string w
Output: Whether $x \in L$

1 P: Send an oracle R to V;
2 V^R: Engage in an interactive protocol with P;

Algorithm 4.9: The IPCP protocol

Next, we need to define a few properties of IPCPs. In general we will care about IPCPs with specific bounds on these properties; later on we will spend some time (in particular Section 7.1) working on optimizing bounds on some of these properties at the expense of others.

The first two complexities we will look at come from the interactive-proof portion of the IPCP. As a reminder, two important parameters we care about with regard to interactive proofs are the number of communication rounds (i.e., the number of times the switch tape flips) and the total amount of information sent between the two machines. Since these are supposed to relate to just the IP portion of the protocol, we need to modify the definitions slightly to exclude the information exchanged during the PCP phase.

Definition 4.7.2. The *round complexity* of an IPCP is the number of rounds in the second portion of the protocol.

Definition 4.7.3. The *communication complexity* of an IPCP is the total number of bits exchanged between P and V *except* for the message that contains R.

The third complexity we care about comes from the PCP portion of the IPCP. TODO:

Definition 4.7.4. The *query complexity* of an IPCP is the total number of queries that V makes to the PCP oracle R.

Definition 4.7.5. The class IPCP is the class of all languages with an interactive PCP.

Next, we give a few reasonable class inclusions regarding IPCP.

Theorem 4.7.6. $\mathsf{PCP} \subseteq \mathsf{IPCP}$ *and* $\mathsf{IP} \subseteq \mathsf{IPCP}$.

Proof. Both of these come from the definition of an interactive PCP: a regular PCP is simply the first half of the IPCP protocol where the interactive portion is useless, and an interactive proof is simply the second half of the protocol where the oracle is useless. □

The tuple-notation we used when talking about the class PCP (see Definition 4.5.3) is rather hard to read when we have this many parameters, and as such we will us the following clearer notation when talking about the various bounds on IPCP algorithms:

$$L \in \mathsf{IPCP}\left[\begin{array}{rl} \text{round complexity:} & r \\ \text{PCP length:} & \ell \\ \text{comm. complexity:} & c \\ \text{query complexity:} & q \\ \text{soundness error:} & \varepsilon \end{array}\right]$$

to mean the language L is a member of IPCP with the listed restrictions.

TODO: Does it make sense to talk about BFL90's IPCP for all of NEXP?

Definition 4.7.7. Let $\mathbb{F}$ be a field, and $d, m \in \mathbb{N}$. A *low-degree IPCP* is an IPCP instance with the following two properties:

1. The oracle sent by the honest prover P is an m-variable $\mathbb{F}$-polynomial Q with multidegree no more than d (i.e. $Q \in \mathbb{F}[X_{1,\ldots,m}^{\leq d}]$)

2. Soundness is only required to hold against provers that send oracles that are polynomials in $\mathbb{F}[X_{1,\dots,m}^{\leq d}]$.

Similarly to what we did with normal IPCP oracles, we will use the following notation to talk about low-degree IPCP instances:

$$L \in \mathsf{IPCP}\left[\begin{array}{rl} \text{round complexity:} & r \\ \text{PCP length:} & \ell \\ \text{comm. complexity:} & c \\ \text{query complexity:} & q \\ \text{oracle:} & \mathbb{F}[X_{1,\dots,m}^{\leq d}] \\ \text{soundness error:} & \varepsilon \end{array}\right].$$

We combine all the information about the degree of the oracle into one line because we have an efficient notation for multidegree-bounded polynomials, and so that we do not wind up with an exorbitant number of lines in our notation.[6]

TODO: Examples

Definition 4.7.8. The *view* of an IPCP (P, V) on input x is the random variable

$$(x, r, s_1, \dots, s_n, t_1, \dots, t_m)$$

where r is the random bits used by V, s_i are the messages from P, and t_i are the answers to V's queries to the oracle sent by P.

We denote the view of P and V on input x by $\mathrm{View}_V^P(x)$.

4.8 Zero-knowledge IPCPs

At this point, the astute reader may have noticed a trend: after each new interactive-proof variant we introduce, we then describe how to make it zero-knowledge. We will continue this trend by showing how an interactive PCP can be made zero-knowledge.

Definition 4.8.1 ([11, §5.2])**.** An interactive PCP is *perfect zero-knowledge* with query bound b when there exists a polynomial-time simulator algorithm S such that for every interactive Turing machine $\tilde{V}$ that makes no more than b queries, $S^{\tilde{V}}(x)$ and $\mathrm{View}\langle P(x), \tilde{V}(x)\rangle$ are identically distributed.

Definition 4.8.2. The class PZK-IPCP is the class of all languages with a perfect zero-knowledge IPCP.

TODO: Example probably

As with our other notations, we will write

$$L \in \mathsf{PZK\text{-}IPCP}\left[\begin{array}{rl} \text{round complexity:} & r \\ \text{PCP length:} & \ell \\ \text{comm. complexity:} & c \\ \text{query complexity:} & q \\ \text{query bound:} & b \\ \text{soundness error:} & \varepsilon \end{array}\right]$$

to show L has a perfect zero-knowledge IPCP with the listed restrictions.

[6] Having said that, there are still a lot of lines in this notation, but this is the best we can do.

Chapter 5

The PCP theorem

Theorem 5.0.1 (PCP theorem, [3])**.** *Any problem in* NP *has a probabilistically-checkable proof of constant query complexity and using a maximum of* $O(\log n)$ *random bits, and vice versa. Equivalently,* $\mathsf{NP} = \mathsf{PCP}(\log n, 1)$.

TODO:

5.1 Algebraic circuits

TODO: Mention what the size of an algebraic circuit is

Definition 5.1.1 ([2, §14.1])**.** An *algebraic circuit* is a directed acyclic graph such that

1. each leaf (called an *input node*) takes values in some field F,
2. each internal node (called a *gate*) is labeled with either $+$ or $\cdot$ (the two field operations),
3. there is one output node, and
4. each gate has in-degree no more than 2.

Optionally, there may be input nodes labeled 1 and -1 as well.

We give an example of an algebraic circuit in Figure 5.1. Notice in particular that unlike the in-degree, the out-degree is unbounded (the rightmost node in the second row has out-degree three, for example).

TODO: More

Definition 5.1.2. An *assignment* to an algebraic circuit is a labeling of its input nodes. The *result* of the assignment is the value in the output node, where the value of any $+$ gate is $a + b$ and any $\cdot$ gate is $a \times b$, where a and b are the values of the two in-nodes.

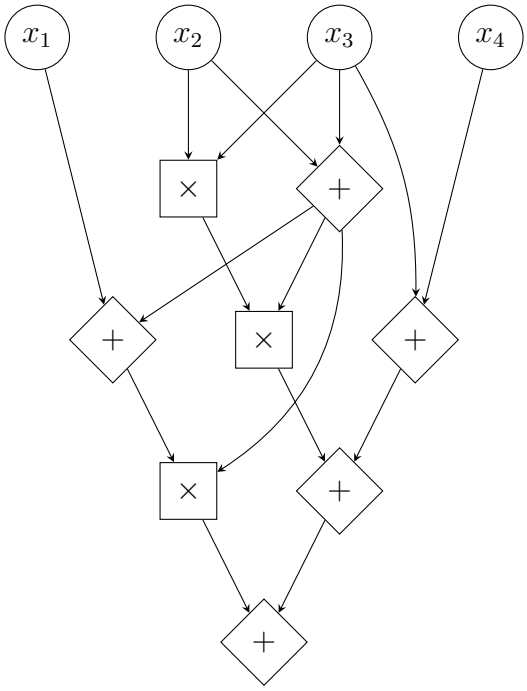

Figure 5.1: An algebraic circuit

TODO:

Definition 5.1.3. An algebraic circuit has a *satisfying assignment* when there exists a labeling of its input nodes such that the value computed by the output node is 1.

We give a satisfying assignment to Figure 5.1 as an example in Figure 5.2. Notice that we have kept the shapes of the nodes from the original in order to communicate the underlying operation.

Satisfying assignments will form the root of our circuit classes: the problem we care about computing is whether or not a circuit has a satisfying assignment. We find this a useful formulation because it works well with arbitrary algebraic formulas: since we will be working in a primarily algebraic setting for these proofs, having an example of a NP-complete problem that is itself algebraic will make for relatively easy transformations.

TODO: More

Definition 5.1.4. The language CktSAT is the language of all algebraic circuits with a satisfying assignment.

Theorem 5.1.5. CktSAT *is* NP*-complete.*

Proof. First, we need that CktSAT $\in$ NP. Evaluating a circuit can be done in polynomial time relative to its size, so a certificate that is simply the values of each input node in the satisfying assignment will suffice.

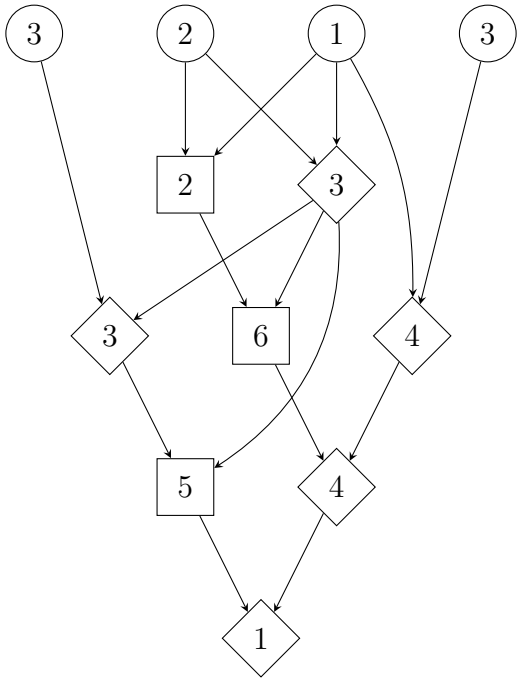

Figure 5.2: A satisfying assignment to Figure 5.1 over $\mathbb{Z}/7\mathbb{Z}$

Next, we need that there is a reduction from every problem in NP to CktSAT. We will use the fact that we already know normal SAT is NP-complete for this. First, consider the *boolean circuit problem*: similar to algebraic circuits but where the operators are boolean formulae instead of the field operators. We can construct a boolean circuit from a boolean formula by replacing every variable with an input node and every boolean operator with a gate corresponding to its formula.

From there, we can transform a boolean circuit into an algebraic circuit over the field $\mathbb{Z}/2\mathbb{Z}$. Multiplication in $\mathbb{Z}/2\mathbb{Z}$ is exactly an AND gate, so that is simply a one-to-one replacement. OR gates are slightly trickier, however. We would like the replacement to be as simple as multiplication: mapping $x \vee y$ to $x + y$. However, $T \vee T = T$ but $1 + 1 = 0$ in $\mathbb{Z}/2\mathbb{Z}$. Instead, we map $x \vee y$ to $(x + y) + (x \times y)$, which gives the answer we seek. Lastly, we map NOT gates to $x + 1$.

We have shown CktSAT $\in$ NP and we have shown a polynomial-time reduction from SAT to CktSAT; hence CktSAT is NP-complete. □

Definition 5.1.6. The language CktVal is the language of all pairs (C, w), where C is an algebraic circuit and w is a satisfying assignment (i.e., $C(w) = 1$).

Theorem 5.1.7. CktVal *is in* P.

Proof. We are given a circuit and an assignment; computing the resultant value of a circuit is possible in polynomial time relative to its length. Hence, we can simply compute $C(w)$ directly and check if the answer is 1. □

Theorem 5.1.8 ([9, Prop. 2.4]). *If* CktVal *has a PCPP, then* CktSAT *has a PCP with identical parameters.*

Proof. □

Theorem 5.1.9 ([9, Theorem 3.2]). *For any $\varepsilon > 0$,* CktVal *has a PCPP (P, V) where P is deterministic and polynomial-time, such that*

$$\mathsf{CktVal} \in \mathsf{PCPP}\left[\begin{array}{rl} \textit{rand. complexity:} & \log(n) + O(\log^{\varepsilon}(n)) \\ \textit{query complexity:} & O(1/\varepsilon) \\ \textit{prox. param.:} & \Theta(\varepsilon) \\ \textit{soundness error:} & 1/2 \end{array}\right].$$

Proof. □

5.2 Robust verifiers

Definition 5.2.1 ([9, Def. 2.6]). Let $s, \rho \colon \mathbb{N} \to [0,1]$. A PCP verifier V has *robust-soundness error* s with *robustness parameter* ρ if for all $x \notin L$, the bits read by V are ρ-close to being accepted with probability strictly less than s.

We will denote robust PCP classes with

$$\mathsf{PCP}\left[\begin{array}{rl} \text{query complexity:} & q(n) \\ \text{random complexity:} & r(n) \\ \text{robustness parameter:} & s(n) \\ \text{robust-soundness error:} & \rho(n) \end{array}\right].$$

5.3 The composition theorem

We now introduce a theorem that will allow us to compose robust PCPs with robust PCPPs to make a new (non-robust) PCP with improved bounds.

Theorem 5.3.1 ([9, Theorem 2.7]). *Let*

$$r_{\text{out}}, r_{\text{in}}, d_{\text{out}}, d_{\text{in}}, q_{\text{in}} \colon \mathbb{N} \to \mathbb{N}$$
$$\varepsilon_{\text{out}}, \varepsilon_{\text{in}}, \rho_{\text{out}}, \delta_{\text{in}} \colon \mathbb{N} \to [0,1]$$

be functions such that the following holds:

1. *The language L has a robust PCP verifier V_{out} with randomness complexity r_{out}, decision complexity d_{out}, robust-soundness error $1 - \varepsilon_{\text{out}}$, and robustness parameter ρ_{out}.*
2. *CktVal has a PCPP verifier V_{in} with randomness complexity r_{in}, query complexity q_{in}, decision complexity d_{in}, proximity parameter δ_{in}, and soundness error $1 - \varepsilon_{\text{in}}$.*
3. *For every $n \in \mathbb{N}$, $\delta_{\text{in}}(d_{\text{out}}(n)) \leq \rho_{\text{out}}(n)$.*

Then L has a standard PCP V_{comp} with

a. *randomness complexity $r_{\text{out}}(n) + r_{\text{in}}(d_{\text{out}}(n))$,*

b. *query complexity $q_{\text{in}}(d_{\text{out}}(n))$,*

c. *decision complexity $d_{\text{in}}(d_{\text{out}}(n))$, and*

d. *soundness error $1 - \varepsilon_{\text{out}}(n)\varepsilon_{\text{in}}(d_{\text{out}}(n))$.*

Algorithm 5.1: A composed PCP [9, Theorem 2.7]

Proof. We show the validity of the proof and verifier described in Algorithm 5.1. □

5.4 Alphabet reduction

Theorem 5.4.1 ([9, Lemma 2.13])**.** *Let L be a language with a PCP over the language $\{0,1\}^a$ such that*

$$L \in \mathsf{PCP}\begin{bmatrix} \textit{query complexity:} & q \\ \textit{random complexity:} & r \\ \textit{robustness parameter:} & s \\ \textit{robust-soundness error:} & \rho \end{bmatrix}.$$

Then L has a PCP over the language $\{0,1\}$ such that

$$L \in \mathsf{PCP}\begin{bmatrix} \textit{query complexity:} & r \\ \textit{random complexity:} & O(aq) \\ \textit{robustness parameter:} & s \\ \textit{robust-soundness error:} & \Omega(\rho) \end{bmatrix}.$$

Input: A PCP (P, V) over the language $\{0,1\}^a$ for L
`/* Proof */`
1 Run $P(x)$ to obtain a proof π;
2 Define the proof oracle τ by $\tau(\gamma) = \mathsf{ECC}(\pi(\gamma))$;
3 **return** (π, τ);
`/* Verifier */`

Algorithm 5.2: A boolean reduction of a PCP [16, Construction 3.6]

Proof. We show that Algorithm 5.2 is a boolean reduction algorithm. □

5.5 Robust PCP verifiers for CktSAT

5.6 Parallelizable PCPPs

Theorem 5.6.1 ([4, Theorem 2.1.9])**.**

Proof. □

Theorem 5.6.2 ([9, Prop. 2.14])**.**

Proof. □

5.7 An upper bound on minimal PCP queries

Theorem 5.7.1 ([19])**.** *Any language in* NP *has a* PCP *that queries a maximum of* 3 *bits of the proof and uses* $O(\log n)$ *random bits.*

Chapter 6

Quantum computation

6.1 Quantum computers

Definition 6.1.1. A *qubit* is a unit vector in $\mathbb{C}^2$.

Definition 6.1.2. A *operator* is a linear function $A\colon V \to V$ such that $v \cdot A \cdot v^T \geq 0$ for all $v \in V$.

6.1.1 Measurement

Definition 6.1.3. A *projective measurement* is

Definition 6.1.4. A *positive operator-valued measure* is

6.2 Quantum complexity classes

Definition 6.2.1. The class BQP is

Definition 6.2.2. The class QMA is

6.3 Quantum interactive proofs

Definition 6.3.1. The class MIP$*$ is

Theorem 6.3.2 ([21])**.** MIP$* =$ RE.

6.4 Quantum low-multidegree test

Theorem 6.4.1 ([22])**.** *There is a universal constant $C > 0$ such that the following holds. Let $(\tilde{P}_1, \tilde{P}_2, |\Psi\rangle)$ be a projective strategy that passes the $(\mathbb{F}, d, m)$-low-multidegree test with probability at least $1 - \varepsilon$. For $i \in \{1, 2\}$, $\alpha \in \mathbb{F}^m$, denote by $\{A^z_{i,\alpha}\}_{z\in\mathbb{F}}$ the measurement applied by $\tilde{P}_i$ upon recieving question α. Then there exist projective*

measurements $\{L_1^Q\}_{Q\in\mathbb{F}[X_{1,\dots,m}^{\leq d}]}$ *and* $\{L_2^Q\}_{Q\in\mathbb{F}[X_{1,\dots,m}^{\leq d}]}$ *such that for* $v \in \mathsf{poly}(m,d)(d/|\mathbb{F}|+c)^C$*, the following holds:*

1. *Consistency with* $\{A_{1,\alpha}^z\}_{z\in\mathbb{F},\alpha\in\mathbb{F}^m}$ *and* $\{A_{1,\alpha}^z\}_{z\in\mathbb{F},\alpha\in\mathbb{F}^m}$*:*

$$\mathbb{E}_{\alpha\in\mathbb{F}^m} \sum_{Q\in\mathbb{F}[X_{1,\dots,m}^{\leq d}]} \sum_{z\in\mathbb{F}\setminus\{Q(\alpha)\}} \langle\Psi \mid A_{1,\alpha}^z \otimes L_2^Q \mid \Psi\rangle \leq v, \tag{6.1}$$

$$\mathbb{E}_{\alpha\in\mathbb{F}^m} \sum_{Q\in\mathbb{F}[X_{1,\dots,m}^{\leq d}]} \sum_{z\in\mathbb{F}\setminus\{Q(\alpha)\}} \langle\Psi \mid L_1^Q \otimes A_{2,\alpha}^z \mid \Psi\rangle \leq v. \tag{6.2}$$

2. *Consistency of* $\{L_1^Q\}_Q$ *and* $\{L_2^Q\}_Q$*:*

$$\sum_{Q\neq Q'\in\mathbb{F}[X_{1,\dots,m}^{\leq d}]} \langle\Psi \mid L_1^Q \otimes L_2^Q \mid \Psi\rangle \leq v. \tag{6.3}$$

Proof. □

Chapter 7

Lifting IPCP to MIP∗

7.1 Reducing query complexity

Theorem 7.1.1 ([11, Prop. 9.2])**.** *There exists a transformation T such that, for every $m, d \in \mathbb{N}$ and finite field $\mathbb{F}$, if*

$$(P,V) \in \mathsf{IPCP}\left[\begin{array}{rl} \textit{round complexity:} & r \\ \textit{PCP length:} & \ell \\ \textit{comm. complexity:} & c \\ \textit{query complexity:} & q \\ \textit{oracle:} & \mathbb{F}[X_{1,\ldots,m}^{\leq d}] \\ \textit{soundness error:} & \varepsilon \end{array}\right],$$

then $T(P,V)$ recognizes the same language as (P,V) and

$$T(P,V) \in \mathsf{IPCP}\left[\begin{array}{rl} \textit{round complexity:} & r+1 \\ \textit{PCP length:} & \ell \\ \textit{comm. complexity:} & c + \mathsf{poly}(m,d,q) \\ \textit{query complexity:} & 1 \\ \textit{oracle:} & \mathbb{F}[X_{1,\ldots,m}^{\leq d}] \\ \textit{soundness error:} & \varepsilon + \frac{mdq}{|\mathbb{F}|-q} \end{array}\right].$$

Proof. We describe the result of T in Algorithm 7.1. First, note that this algorithm is not T directly, but the result of $T(P,V)$. So, what we require is to show that $L_{T(P,V)} = L_{(P,V)}$, and to briefly demonstrate why V' runs in polynomial time.

To show V' runs in polynomial time, note that the random choices are in polynomial time (as a reminder, random choice is polynomial relative to the length), simulating V is polynomial time through the definition of V, and computing polynomials can be done in polynomial time as well.

When P' is honest, then this will simulate V answering all its queries via ρ (which gives the correct answer to every value in A); hence V will gave the correct answer by definition.

When P' is dishonest, then we know ρ is a univariate polynomial with degree at most dq (since $\rho = R\gamma$). Further note that our random choice of both t and γ means

Input: An honest oracle $R \in \mathbb{F}[X_{1,\dots,m}^{\leq d}]$ and string s
Output: Whether or not $s \in L_{(P,V)}$
1 Let $S \subseteq \mathbb{F}$ with $|S| < q$;
2 V': choose random $r \in \mathbb{F}^m$;
3 V': choose random $t \in \mathbb{F} \setminus S$;
4 V': Simulate V on s answering every query with 0;
5 Let A be the set of queries of V;
6 V': compute polynomial $\gamma\colon \mathbb{F} \to \mathbb{F}^m$ of degree q such that $\{\gamma(s)\}_{s\in S} = A$ and $\gamma(t) = r$;
7 V': Send γ to P';
8 P': Reply with the coefficients of the polynomial $\rho = R \circ \gamma$;
9 V': query R at r and recieve answer a;
10 **if** $a \neq \rho(t)$ **then**
11 | **reject**;
12 **end**
13 V': Simulate $V(x)$, answering all queries $a \in A$ with $\rho(A)$;
14 **return** *the result of* $V(x)$;

Algorithm 7.1: A single-query, zero-knowledge transformation of an IPCP [11, Construction 4]

that $\gamma(t)$ is uniformly distributed over $\mathbb{F}^m$. Since ρ is a univariate polynomial with degree at most mdq, then it can agree with $R \circ \gamma$ at most mdq distinct points without being equal. Hence, since we query it at a random point in $\mathbb{F} \setminus S$, if the check in line 10 passes with at probability bigger than $\frac{mdq}{|\mathbb{F}|-q}$, it must mean that $\rho = R\gamma$ on more than mdq points. Thus, $\rho = R \circ \gamma$ and so we will get the correct answer for all of our queries on A. If it passes with a lower probability, then we are within our soundness error overall and so the result does not matter. □

Next, we show that Algorithm 7.1 also preserves zero knowledge. This is important for us because zero knowledge is *actually* what we care about.

Theorem 7.1.2 ([11, Prop. 9.2])**.** *If (P, V) is perfect zero-knowledge with query bound b, then $T(P, V)$ (where T is the transformation from Theorem 7.1.1) is perfect zero-knowledge with query bound $b - (mdq + 1)$.*

Proof. We again look at Algorithm 7.1. First, notice that if we make a query $r \in \gamma(\mathbb{F})$, since $\rho = R\gamma$ we could replace our call to R with a call to $\rho(\gamma^{-1}(r))$, and this would no longer count towards our query bound. Further, from what we know of the degree of ρ, we know that in the worst case, we would only need $mdq + 1$ queries to R to effectively replicate ρ. Hence, if we reduce our query bound by $mdq + 1$, it follows that we can still effectively simulate the interaction of P' and V', since we know P and V have a perfect simulator. □

7.2 A lifting algorithm

Theorem 7.2.1 ([11, Lemma 9.1]). *Let L be a language, let $m, d, q \in \mathbb{N}$, and let $\mathbb{F}$ be a finite field of size $\mathsf{poly}(m, d, q)$ sufficiently large. Then, there exists a transformation*

$$T : \mathsf{IPCP}\left[\begin{array}{rl} \textit{round complexity:} & r \\ \textit{PCP length:} & \ell \\ \textit{comm. complexity:} & c \\ \textit{query complexity:} & q \\ \textit{oracle:} & \mathbb{F}[X_{1,\ldots,m}^{\leq d}] \\ \textit{soundness error:} & \varepsilon \end{array}\right] \to \mathsf{MIP}*\left[\begin{array}{rl} \textit{number of provers:} & 2 \\ \textit{round complexity:} & r+1 \\ \textit{comm. complexity:} & c \\ \textit{soundness error:} & 1-\frac{1}{\mathsf{poly}(m,d)} \end{array}\right].$$

such that (P', V') and $T(P', V')$ recognize the same language.

Further, if the IPCP (P', V') is zero-knowledge with query bound $b \geq 2(q+1)md+3$, then the $\mathsf{MIP}*$ (P_1, P_2, V) is zero-knowledge.

1 V: Choose a random $r \in \{0, 1\}$;
2 **if** $r = 0$ **then**
3 | V: Perform the low multidegree test from Theorem 6.4.1;
4 **else**
| `// IPCP emulation`
5 | P_1 and V emulate the interaction of the IPCP $(P'', V'') = T(P', V')$ (see Theorem 7.1.1);
6 | The above emulation generates a uniform $\beta \in \mathbb{F}^m$, and a $c \in \mathbb{F}$ such that with probability $1 - \varepsilon$, $x \in \mathcal{L}$ if and only if $R(\beta) \in c$;
7 | V: ask P_2 for an evaluation of R at β;
8 | P_2: reply with an element $z \in \mathbb{F}$;
9 | V: accept if and only if $c = z$;
10 **end**

Algorithm 7.2: Construction of a $\mathsf{MIP}*$ from an IPCP [11, Construction 2]

7.2.1 Soundness of Algorithm 7.2

7.2.2 Algorithm 7.2 preserves zero-knowledge

Algorithm 7.3: A simulator for Algorithm 7.2 [11, §9.4]

Chapter 8

Low-degree IPCP with zero-knowledge

Now that we can construct a zero-knowledge MIP$*$ instance from a zero-knowledge IPCP, all that remains is to show that NEXP $\subseteq$ PZK-IPCP. From there, we will be able to leverage Theorem 7.2.1 to demonstrate NEXP $\subseteq$ PZK-MIP$*$.

8.1 AQC of polynomial summation

We first need to define the *polynomial summation problem*. We will want a lower bound on the algebraic query complexity of this problem, similar to the examples we saw in Section 3.1.

Definition 8.1.1. The *polynomial summation problem* is the following:

> Let $\mathbb{F}$ be a field with $G \subseteq \mathbb{F}$. Let $m, k, d, d' \in \mathbb{N}$ and let $Z \in \mathbb{F}[X^{\leq d}_{1,\dots,m}, Y^{\leq d'}_{1,\dots,k}]$. What is the value of the polynomial
>
> $$R(X) = \sum_{\beta \in G^k} Z(X, \beta) \in \mathbb{F}[X^{\leq d}_{1,\dots,m}]?$$

We will need a lower bound on the algebraic query complexity of this problem (where Z functions as the oracle) later on in order to help demonstrate zero-knowledge. We will do this with one additional restriction—we will also need d' to be sufficiently large relative to G, but this will not hamper us in practice. In brief, the lower bound will tell us that so long as we limit our total queries, we will not recieve *any* information about $R(X)$.

Lemma 8.1.2 ([11, Lemma 12.1])**.** *Let $\mathbb{F}$ be a field, $m, k, d, d' \in \mathbb{N}$, and G, K, L be finite subsets of $\mathbb{F}$ such that $K \subseteq L$, $d' \geq |G| - 2$, and $|K| = d + 1$. If $S \subseteq \mathbb{F}^{m+k}$ is such that there exist matrices $C \in \mathbb{F}^{L^m \times \ell}$ and $D \in \mathbb{F}^{S \times \ell}$ such that for all $Z \in \mathbb{F}[X^{\leq d}_{1,\dots,m}, Y^{\leq d'}_{1,\dots,k}]$ and all $i \in \{1, \dots, \ell\}$*

$$\sum_{\alpha \in L^m} C_{\alpha,i} \sum_{y \in G^k} Z(\alpha, y) = \sum_{q \in S} D_{q,i} Z(q), \tag{8.1}$$

then $|S| \geq \operatorname{rank}(BC)(\min(d' - |G| + 2, |G|))^k$, where $B \in \mathbb{F}^{K^m \times L^m}$ is such that the column of B indexed by α represents $Z(\alpha)$ in the basis $\{Z(\beta) \mid \beta \in K^m\}$.

Proof. First, if $d' = |G| - 2$, then $d' - |G| + 2 = 0$; hence our bound simplifies to

$$\begin{aligned} |S| &\geq \operatorname{rank}(BC) \min(0, |G|)^k \\ &\geq 0 \operatorname{rank}(BC) \\ &\geq 0, \end{aligned}$$

which is true regardless of S.

Otherwise, we can rewrite the left-hand side of Equation (8.1) as follows:

$$\sum_{\alpha \in L^m} C_{\alpha,i} \sum_{y \in G^k} Z(\alpha, y) = \sum_{\alpha \in L^m} C_{m,i} \sum_{\beta \in K^m} b_{\beta,\alpha} \sum_{y \in G^k} Z(\beta, y). \tag{8.2}$$

Then, define $B \in M_{K^m, L^m}(\mathbb{F})$ to be the matrix whose (i,j)-entry is $\beta_{i,j}$, and define $C' = BC \in M_{K^m,\ell}(\mathbb{F})$. From that, Equation (8.2) simplifies to

$$\sum_{\alpha \in L^m} C_{m,i} \sum_{\beta \in K^m} b_{\beta,\alpha} \sum_{y \in G^k} Z(\beta, y) = \sum_{\beta \in K^m} C'_{\beta,i} \sum_{y \in G^k} Z(\beta, y). \tag{8.3}$$

Next, define $H \subseteq G$ such that $|H| = \min(d' - |G| + 2, |G|)$. Further, let

$$P_0 = \{p \in \mathbb{F}[X_{1,\ldots,m}^{\leq d}, Y_{1,\ldots,k}^{\leq |H|-1}] \mid p(q) = 0 \text{ for all } q \in S\}.$$

We can write P_0 as the kernel of the linear function $F_S \colon \mathbb{F}[X_{1,\ldots,m}^{\leq d}, Y_{1,\ldots,k}^{\leq |H|-1}] \to \mathbb{F}^S$ defined by $(F_S(p))_s = p(s)$. By the rank-nullity theorem, this means

$$\dim(P_0) \geq (d+1)^m |H|^m - |S|.$$

Let $A_0 \in M_{n,(K^m \times H^k)}(\mathbb{F})$ (for some arbitrary n) be a matrix whose rows form a basis for

$$\hat{P}_0 = \{(p(\alpha, y))_{\alpha \in K^m, y \in H^k} \mid p \in P_0\}.$$

The dimension of $\hat{P}_0$ is exactly the dimension of P_0; since two distinct polynomials of degree n can agree at no more than n points, polynomials in P have maximum degree $d^m(|H| - 1)^k$, and each vector in $\hat{P}_0$ has $|K^m \times H^k| = (d+1)^m |H|^k$ points, it follows that the map from p to its corresponding vector in $\hat{P}_0$ is a bijection and thus dimension is preserved.

For any $y_0 \in H^k$, let $A_{y_0} \in M_{n,K^m}(\mathbb{F})$ be the submatrix of A_0 consisting only of rows where $y = y_0$. Since A_0's columns form a basis, it has full rank. Further, the set of all A_{y_0}s span the row space of A_0; hence

$$n = \operatorname{rank}(A_0) \leq \sum_{y_0 \in H^k} \operatorname{rank}(A_{y_0}).$$

Hence, there exists a y_0 such that

$$\dim(A_{y_0}) \geq \frac{\operatorname{rank}(A_m)}{|H^k|} \geq \frac{(d+1)^m |H^k| - |S|}{|H^k|} = (d+1)^m - \frac{|S|}{|H^k|}.$$

Let $q \in \mathbb{F}[Y_{1,\dots,k}^{\geq |G|-1}]$ be the polynomial such that $q(y_0) = 1$ and $q(y) = 0$ for all $y \in G^k \setminus \{y_0\}$ (as per Theorem 1.3.6). Then, for all $i \in \{1, \dots, n\}$ and $j \in \{1, \dots, k\}$ it holds that

$$(A_{y_0} C')_{ij} = \sum_{\beta \in K^m} C'_{\beta,j} p_i(\beta, y_0) \tag{8.4}$$

where $p_i(\alpha, \beta)$ is the element of A_0 in row i and column (α, β). This comes from the definition of matrix multiplication and A_{y_0}. Next, the definition of q gives us

$$\sum_{\beta \in K^m} C'_{\beta,j} p_i(\beta, y_0) = \sum_{\beta \in K^m} C'_{\beta,j} \sum_{y \in G^k} q(y) p_i(\beta, y). \tag{8.5}$$

Now, since p_i is a row of A_0 and the rows of A_0 form a basis for the space of all outputs of polynomials in P_0, we can simply treat p_i as a polynomial in P_0.

Note that $qp_i \in \mathbb{F}[X_{1,\dots,k}^{\geq d}, Y_{1,\dots,k}^{\geq d'}]$ (from the definitions of q and p_i). Hence, Equation (8.3) tells us that

$$\sum_{\beta \in K^m} C'_{\beta,j} \sum_{y \in G^k} q(y) p_i(\beta, y) = \sum_{s \in S} D_{s,i}(qp_i)(s). \tag{8.6}$$

Since $s \in Q$, the definition of P_0 tells us that $p_i(s) = 0$; hence the entire sum in Equation (8.6) is equal to zero and thus $A_{y_0} C' = 0$.

Lastly, we can apply Sylvester's rank inequality:

$$\begin{aligned} \operatorname{rank}(A_{y_0}) + \operatorname{rank}(C') - (d+1)^m &\leq \operatorname{rank}(0) \\ (d+1)^m - \operatorname{rank}(C') &\geq \operatorname{rank}(A_{y_0}) \\ (d+1)^m - \operatorname{rank}(C') &\geq (d+1)^m - |S|/|H|^k \\ |S| &\geq \operatorname{rank}(C')|H|^k. \end{aligned}$$

From the definition of H, this means

$$|S| \geq \operatorname{rank}(C')(\min(d' - |G| + 2, |G|))^k,$$

as desired. □

Corollary 8.1.3 ([11, Corollary 12.2])**.** *Let $\mathbb{F}$ be a finite field, $G \subseteq \mathbb{F}$, and $d, d' \in \mathbb{N}$ with $d' \geq 2(|G| - 1)$. If $S \subseteq \mathbb{F}^{m+k}$ is such that there exist $(c_\alpha)_{\alpha \in \mathbb{F}^m}$ and $(d_\beta)_{\beta \in \mathbb{F}^{m+k}}$ such that*

1. *for all $Z \in \mathbb{F}[X_{1,\dots,m}^{\leq d}, Y_{1,\dots,k}^{\leq d}]$ it holds that*

$$\sum_{\alpha \in \mathbb{F}^m} c_\alpha \sum_{y \in G^k} Z(\alpha, y) = \sum_{q \in S} d_q Z(q), \tag{8.7}$$

and

2. *there exists $Z' \in \mathbb{F}[X_{1,\dots,m}^{\leq d}, Y_{1,\dots,k}^{\leq d}]$ such that*

$$\sum_{\alpha \in \mathbb{F}^m} c_\alpha \sum_{y \in G^k} Z'(\alpha, y) = 0, \tag{8.8}$$

then $|S| \geq |G|^k$.

Proof. □

From here, we get that the algebraic query complexity of polynomial summation is at least $|G|^k$. That is, if we query Z no more than $|G|^k$ times, then we we will recieve *no information* about the polynomial $R(X) = \sum_{\beta \in G^k} Z(X, \beta)$.

Corollary 8.1.4 ([11, Corollary 12.3])**.** *Let* $\mathbb{F}$ *be a finite field,* $G \subseteq \mathbb{F}$*, and* $d, d' \in \mathbb{N}$ *with* $d' \geq 2(|G|-1)$*. Let* Q *be a subset of* $\mathbb{F}^{m+k}$ *with* $|Q| \leq |G|^k$*, and let* Z *be uniformly random in* $\mathbb{F}[X_{1,\ldots,m}^{\leq d}, Y_{1,\ldots,k}^{\leq d'}]$*. Then, the random variables* $(\sum_{y \in G^k} Z(\alpha, y))_{\alpha \in \mathbb{F}^m}$ *and* $(Z(q))_{q \in Q}$ *are independent.*

Proof. We will leverage Theorem 1.4.3 that we proved earlier. Now, consider the vector space

$$\left\{ \left((Z(\gamma))_{\gamma \in \mathbb{F}^{m+k}}, \left(\sum_{y \in G^k} Z(\alpha, y) \right)_{\alpha \in \mathbb{F}^m} \right) \;\middle|\; Z \in \mathbb{F}[X_{1,\ldots,m}^{\leq d}, Y_{1,\ldots,k}^{\leq d'}] \right\} \tag{8.9}$$

This is a vector space over $\mathbb{F}$ with domain $\mathbb{F}^{m+k} \cup \mathbb{F}^m$. □

8.2 Algebraic commitment schemes

We gave an explanation of bit-commitment schemes in Section 4.4.1, but there are many circumstances in which commitment to a single bit is insufficient. In some cases, we will want to commit to an entire polynomial. This is the setting in which *algebraic commitment schemes* exist.

Definition 8.2.1. An *algebraic commitment scheme* is a commitment scheme such that instead of S recieving a single bit it recieves a polynomial $Q \colon \mathbb{F}^m \to \mathbb{F}$.

TODO: Mention the role of algebraic query complexity (Section 3.1) in commitment schemes

We would next like to give an example of an algebraic commitment scheme. TODO:

Theorem 8.2.2. *Algorithm 8.1 is a valid algebraic commitment scheme.*

Proof. TODO: □

8.3 The sumcheck problem

Central to our upcoming work will be a nice answer to the *sumcheck problem*, a computational problem about verifying whether or not the sum of a polynomial over some subset of a field is equal to a provided value. This will prove useful to us in arithmetizing our problems: if we can turn our boolean formulae into a question in the sumcheck format, then we can delegate to the protocol to complete our proof.

TODO: Should bring in single-element commitment scheme
Input: A polynomial $Q \in \mathbb{F}[X_{1,\dots,m}^{\leq d_Q}]$
Let $K \subseteq \mathbb{F}$ with $|K| = d + 1$;
for $\alpha \in K^m$ **do**
  sample a random $B^\alpha \in \mathbb{F}^n$ such that $\sum_{i=1}^m B_i^\alpha = Q(\alpha)$;
end
FIXME: What on earth is G? Define $B\colon K^m \times G^k \to \mathbb{F}$ by $B(\alpha, x) = B^\alpha(x)$;
TODO: d_Q vs d Define $\hat{B}\colon \mathbb{F}^m \times \mathbb{F}^k \to \mathbb{F}$ to be a low-degree extension of B;
`/* Note:` $P \in \mathbb{F}[X_{1,\dots,m}^{\leq d_Q}, Y_{1,\dots,m}^{\leq d}]$ `*/`
return $\hat{B}$;

Algorithm 8.1: An algebraic commitment scheme [11, §12]

Definition 8.3.1 ([26])**.** The *sumcheck problem* is the following problem:

> Let H be a subset of a finite field $\mathbb{F}$, let $F \in \mathbb{F}[X_{1,\dots,m}^{\leq d}]$ a polynomial over $\mathbb{F}$, and let $a \in \mathbb{F}$. Does $\sum_{x \in H^m} F(x) = a$?

For this question, we give H and a to both the prover and verifier, but only the prover gets access to F as an algebraic oracle.

8.3.1 A non-zero-knowledge sumcheck protocol

We begin with an interactive protocol for sumcheck that does not have any zero-knowledge characteristics. While this protocol itself does not preserve anything, it will form an essential building block for the zero-knowledge protocols we will construct later.

Theorem 8.3.2. *The sumcheck problem is in* IP.

Proof. We show this by implementing an interactive protocol for sumcheck in Algorithm 8.2. We need to start by showing that this algorithm will correctly answer the sumcheck question. First, if $\sum_{x \in H^m} F(x) = a$ and P is honest, then Vs check in line 2 will succeed if and only if the question is correct. Again assuming an honest P, from the definition of each F_i,

$$F_{i-1}(r_{i-1}) = \sum_{X \in H^{m-i+1}} F(r_1, \dots, r_{i-2}, r_{i-1}, X)$$

and

$$\sum_{x \in H} \sum_{x \in H^{m-i}} F(r_1, \dots, r_{i-1}, x, X) = \sum_{X \in H^{m-i+1}} F(r_1, \dots, r_{i-1}, X).$$

Hence, line 6 will always succeed with an honest P. Finally, if P has been honest in the last iteration, line 12 will always succeed since the sum in the equation in line 5 is now over a single object.

Input: A polynomial $F \in \mathbb{F}[X_{1,\dots,n}^{\leq d}]$, subset $H \subseteq \mathbb{F}$, and number a
Output: Whether $\sum_{x \in H^m} F(x) = a$

1 P: send the polynomial $F_1(x_1) = \sum_{X \in \{0,1\}^{n-1}} F(x_1, X)$;
2 V: check if $a = \sum_{x \in H} F_1(x)$;
3 V: choose random $r_1 \in \mathbb{F}$ and send to P;
4 **for** i *from* 2 *to* n **do**
5 $\quad$ P: send the polynomial

$$F_i(x_i) = \sum_{X \in H^{m-i}} F(r_1, \dots, r_{i-1}, x_i, X)$$

$\quad$ to V;
6 $\quad$ V: check $F_{i-1}(r_{i-1}) = \sum_{x \in H} F_i(x)$;
7 $\quad$ **if** $i \neq n$ **then**
8 $\quad\quad$ V: choose random $r_i \in \mathbb{F}$ and send to P;
9 $\quad$ **end**
10 **end**
11 V: choose random $r_n \in \mathbb{F}$;
12 V: check $F_n(r_n) = F(r_1, \dots, r_n)$;

Algorithm 8.2: The standard sumcheck protocol [26, Thm. 1]

If P is dishonest, then we show soundness by induction on n. If $n = 1$ then there is only one message sent. Two distinct n-variable polynomials of degree d can be equal at most d^n points, as such in the $n = 1$ case the probability of incorrectly passing the check in line 12 is at most $d/|\mathbb{F}|$.

Next, assume the $(n-1)$-variable case has soundness error at most $(n-1)d/|\mathbb{F}|$. Let

$$G_1(x_1) = \sum_{X \in H^{n-1}} F(x_1, X),$$

i.e. the "correct" value of F_1 (were P not to lie). If $F_1 \neq G_1$, then as we saw before, $F_1(r_1) \neq G_1(r_1)$ with probability $1 - d/|\mathbb{F}|$. If this is the case, then in the rest of the loop, the system is attempting to prove the claim

$$F_1(r_1) = \sum_{X \in H^{n-1}} F(r_1, X),$$

which we know to be false. $F(r_1, \cdot)$ is an $(n-1)$-variable polynomial of multidegree d; by induction this has soundness error $(r-1)d/|\mathbb{F}|$. Thus, V will reject with probability

$$1 - \mathbb{P}[F_1(r_1) \neq G_1(r_1)] - \mathbb{P}[V \text{ rejects in round } j > 1 \mid F_1(r_1) \neq G_1(r_1)]$$

which, by substituting in our definitions, gives us probability at least

$$1 - \frac{d}{|\mathbb{F}|} - \frac{d(n-1)}{\mathbb{F}} = 1 - \frac{dn}{|\mathbb{F}|}.$$

The only other step is to show that V runs in polynomial time. Since each loop iteration only requires checking a sum over H, we only need to compute the various F_is (which are themselves only polynomially larger than the original F) a total of nH times; computing F_i is also in polynomial time. Thus, V overall runs in polynomial time. □

Figure 8.1: An example computation of Algorithm 8.2

To illustrate this, in Figure 8.1 we provide an example of an iteration of the protocol when the answer is affirmative and P is honest.

8.3.2 A weakly zero-knowledge sumcheck protocol

Now that we have a non-zero-knowledge interactive protocol for the sumcheck problem, we need to turn it into a zero-knowledge protocol. While we would very much like to jump straight from Algorithm 8.2 to a perfect zero-knowledge algorithm, we need to go through one intermediate step first. We first construct Algorithm 8.3, which is only *weakly* zero-knowledge: while an outside observer to the protocol can learn *something*, it will only learn a limited amount about the polynomial in question. Only once we have this will we be able to proceed to the fully zero-knowledge algorithm.

Definition 8.3.3. A sumcheck protocol is *weak zero-knowledge* if any verifier that makes q queries to R learns at most q evaluations of F.

Theorem 8.3.4. *There exists a weakly zero-knowledge algorithm for the sumcheck protocol.*

Input: A subset $H \subseteq \mathbb{F}$ and integer a
Input: A polynomial F, as an algebraic oracle to P
Output: Whether $\sum_{x \in H^m} F(x) = a$
1 $z \leftarrow \sum_{\alpha \in H^m} R(a)$;
2 P: send z to V;
3 V: draw random $\rho \in \mathbb{F}$;
4 V: send ρ to P;
5 $Q \leftarrow \rho F + R$;
6 Both: run Algorithm 8.2 on the statement $\sum_{\alpha \in H^m} Q(\alpha) = \rho v + z$;
Algorithm 8.3: A weakly zero-knowledge sumcheck protocol [10, Construction 5.4]

Proof. We describe the weakly zero-knowledge algorithm in Algorithm 8.3. We begin by showing this algorithm implements the sumcheck protocol. Notice that this protocol is a very small wrapper around Algorithm 8.2; hence we only need to demonstrate that the modified statement will always have the same result as the original. □

8.3.3 Making the sumcheck protocol zero-knowledge

Theorem 8.3.5 ([11]). *There exists a zero-knowledge variant of Algorithm 8.3.*

Proof. We describe the algorithm in Algorithm 8.4, and then show it is zero-knowledge by constructing a simulator in Algorithm 8.6.

If P is honest in Algorithm 8.4, then both sumcheck protocols will pass if and only if (F, H, a) is valid. From the various definitions in the program, we have

$$\begin{aligned}
\sum_{\alpha \in H^m} Q(a) &= \rho_1 a + z \\
\sum_{\alpha \in H^m} \left(\rho_1 F(\alpha) + \sum_{\beta \in G^k} Z(\alpha, \beta) \right) &= \rho_1 a + \sum_{\alpha \in H^m} \sum_{\beta \in G^k} Z(\alpha, \beta) \\
\sum_{\alpha \in H^m} \rho_1 F(a) + \sum_{a \in H^m} \sum_{\beta \in G^k} Z(\alpha, \beta) &= \rho_1 a + \sum_{\alpha \in H^m} \sum_{\beta \in G^k} Z(\alpha, \beta) \\
\rho_1 \sum_{a \in H^m} F(a) &= \rho_1 a \\
\sum_{a \in H^m} F(a) &= a.
\end{aligned}$$

Since $\rho_1 \neq 0$, all of these transformations are biconditionally true; hence the sumcheck protocol in line 5 will pass if and only if (F, H, a) is valid. The modification does not affect this correctness since all it does is limit the set of elements we can randomly sample from—since we know this works for all $r_i \in \mathbb{F}$, it will also be true for all $r_i \in I$.

For the second sumcheck (in line 14), we have

$$\begin{aligned}
\sum_{\alpha \in H^m} Q'(\alpha) &= \rho w + z \\
\sum_{\alpha \in H^m} (\rho Z(c, \alpha) + A(\alpha)) &= \rho w + \sum_{\alpha \in H^m} A(\alpha) \\
\sum_{\alpha \in H^m} \rho F(\alpha) + \sum_{\alpha \in H^m} A(\alpha) &= \rho w + \sum_{\alpha \in H^m} A(\alpha) \\
\rho \sum_{\alpha \in H^m} F(\alpha) &= \rho w \\
\sum_{\alpha \in H^m} F(\alpha) &= w.
\end{aligned}$$

As before, these transformations are all biconditionally true; hence an honest P will always cause the sumcheck in line 14 to succeed if and only if (F, H, a) is valid.

If P is dishonest, things get trickier. □

Input: An instance (H, a) to both P and V
Input: A polynomial $F \in \mathbb{F}[X_{1,\dots,m}^{\leq d}]$ as an oracle to P
Output: Whether $\sum_{x \in H^m} F(x) = a$

1 P: draw uniformly random polynomials $Z \in \mathbb{F}[X_{1,\dots,m}^{\leq d}, Y_{1,\dots,m}^{\leq 2\lambda}]$ and $A \in \mathbb{F}[Y_{1,\dots,k}^{\leq 2\lambda}]$;
2 P: send the polynomial

$$O(W, X, Y) = W \cdot Z(X, Y) + (1 - W) \cdot A(Y)$$

to V;
// Note that $Z = O(1, \cdot)$ and $A = O(0, 0, \cdot)$, so V can use both Z and A later
3 P: send $z = \sum_{\alpha \in H^m} \sum_{\beta \in G^k} Z(\alpha, \beta)$ to V;
4 V: draw a random element $\rho_1 \in \mathbb{F}^\times$ and send to P;
5 Run the standard sumcheck IP (Algorithm 8.2) on the statement $\sum_{\alpha \in H^m} Q(\alpha) = \rho_1 a + z$, where

$$Q(X_1, \dots, X_m) = \rho_1 F(X_1, \dots, X_m) + \sum_{\beta \in G^k} Z(X_1, \dots, X_m, \beta).$$

We have P play the prover and V the verifier, with the following modification: For $i = 1, \dots, m$, in the ith round, V samples its random element r_i from the set I instead of from all of $\mathbb{F}$; if P ever recieves $r_i \in \mathbb{F} \setminus I$, it immediately aborts. In particular, in the mth round, P sends a polynomial

$$g_m(X_m) = \rho_1 F(c_1, \dots, c_{m-1}, X_m) + \sum_{\beta \in G^k} Z(c_1, \dots, c_{m-1}, X_m, \beta)$$

for some $c_1, \dots, c_{m-1} \in I$;
6 V: send $c_m \in I$ to P;
7 P: send $w \in \sum_{\beta \in G^k} Z(c, \beta)$ to V, where $c = (c_1, \dots, c_m)$;
8 $z' \leftarrow \sum_{\alpha \in H^m} A(\alpha)$;
9 P: send z to V;
10 V: draw random $\rho \in \mathbb{F}^\times$;
11 V: send ρ to P;
12 $Q'(x) \leftarrow \rho Z(c, x) + A$;
13 Both: run Algorithm 8.2 on the statement $\sum_{\alpha \in H^m} Q'(\alpha) = \rho w + z'$;
14 V: output the claim $F(c) = \frac{g_m(c_m) - w}{\rho_1}$;

Algorithm 8.4: Strong zero-knowledge sumcheck [11, Construction 3]

Draw $Z_s \in \mathbb{F}[X_{1,\dots,m}^{\leq d}, Y_{1,\dots,k}^{\leq 2\lambda}]$;
Run the weak-ZK sumcheck simulator;
Begin simulating $\tilde{V}$, answering its oracle queries with Z_s and the simulated A;
Send $z_s = \sum_{\alpha \in H^m} \sum_{\beta \in G^k} Z_s(\alpha, \beta)$;
Recieve $\tilde{\rho}$;
Draw $Q_s \in \mathbb{F}[X_{1,\dots,m}^{\leq d}]$ such that $\sum_{\alpha \in H^m} Q_s(\alpha) = \tilde{\rho}\alpha + z_s$;
Engage in the sumcheck protocol (Algorithm 8.2) on the claim $\sum_{\alpha \in H^m} Q_s(\alpha) = \tilde{\rho}\alpha + z_s$;
if *$\tilde{V}$ sends $c_i \notin I$ as a challenge in the above protocol* **then**
    abort;
end
Let $c \in I^m$ be the point chosen by $\tilde{V}$;
Query $F(c)$;
$w_s \leftarrow Q_s(c) - \tilde{\rho}F(c)$;
Send w_s;
Draw $Z_s' \in \mathbb{F}[X_{1,\dots,m}^{\leq d}, Y_{1,\dots,k}^{\leq 2\lambda}]$ such that
- $\sum_{\beta \in G^k} Z_s'(c, \beta) = w_s$
- $Z_s'(\gamma) = Z_s(\gamma)$ for all previous queries γ;

Use S' to simulate the sumcheck protocol for the claim $\sum_{\beta \in G^k} Z_s'(c, \beta) = w_s$;
Output the view of the simulated $\tilde{V}'$;

Algorithm 8.5: An inefficient simulator for Algorithm 8.4 [11, p. 15:33]

Algorithm 8.6: An efficient variant of Algorithm 8.5 [11, p. 15:34]

8.4 Extending the sumcheck algorithm to NEXP

Armed with our sumcheck algorithm, we are ready to take on the fight of the rest of NEXP. Like so many other theorems, we will do this with a NEXP-complete problem, and as before our problem of choice is O3SAT.

Theorem 8.4.1 ([11, Thm. 14.2])**.** *There exists a $c \in \mathbb{N}$ such that for any query-bound function $b(n)$, $d(n) \in \Omega(n^c)$, $m(n) \in O(n^c \log(b))$, and any sequence of fields $\mathbb{F}(n)$ that are field extensions of $\mathbb{F}_2$ with $|\mathbb{F}(n)| \in \Omega((n^c \log(b))^4)$,*

$$\mathsf{O3SAT} \in \mathsf{IPCP}\left[\begin{array}{rl} \textit{round complexity:} & O(n, b) \\ \textit{PCP length:} & \mathsf{poly}(2^n, b) \\ \textit{comm. complexity:} & \mathsf{poly}(n, \log(b)) \\ \textit{query complexity:} & \mathsf{poly}(n, \log(b)) \\ \textit{oracle:} & \mathbb{F}[X_{1,\dots,m}^{\leq d}] \\ \textit{soundness error:} & 1/2 \end{array}\right],$$

which is zero-knowledge with query bound b.

repeat
  P: Draw a random $Z \in \mathbb{F}[X_{1,\dots,m}^{\leq |H|+2}, Y_{1,\dots,m}^{\leq 2|H|}]$;
until $\sum_{\beta \in G^k} Z(\alpha, \beta) = A(\gamma_2(\alpha))$ *for all* $\alpha \in H^{m_2}$;
for $i \in \{1, 2, 3\}$ **do**
  P and V implement Algorithm 8.3 on the claim $\sum_{\beta \in H^k} Z(c_i', \beta) = h_i$;
end

Algorithm 8.7: A low-degree IPCP for O3SAT [11, p. 15:36]

Draw a random polynomial $Z_s \in \mathbb{F}[X_{1,\dots,m_2}^{\leq |H|+2}, Y_{1,\dots,k}^{\leq 2|H|}]$;
TODO;
Recieve $x, y \in \mathbb{F}^{3s+r}$ from $\tilde{V}$;
TODO;
for $i \in \{1, 2, 3\}$ **do**
end

Algorithm 8.8: A simulator for Algorithm 8.7 [11, p. 15.37]

Proof. We demonstrate this through an algorithm for O3SAT, which we have laid out in Algorithm 8.7. We will show this is zero-knowledge by implementing a simulator in Algorithm 8.8. □

Corollary 8.4.2. NEXP $\subseteq$ PZK-IPCP.

Proof. Since O3SAT is NEXP-complete as per Theorem 1.2.21, we can perform a polynomial reduction from any other language to O3SAT and then run Algorithm 8.7. □

8.5 Zero-knowledge MIP∗ for NEXP

Chapter 9

Zero-knowledge PCPs for #P

9.1 Constraint location

9.2 ZKPCPs for #SAT

Theorem 9.2.1 ([17, Theorem 8.1])**.** *There exists a perfect zero-knowledge PCP for* #SAT*.*

Corollary 9.2.2. #P $\subseteq$ PZK-PCP.

Chapter 10

A zero-knowledge PCP theorem

10.1 Locally-computable proofs

Definition 10.1.1 ([16, Def. 3.1])**.** Let A and A_0 be randomized Turing machines, and let $\ell : \mathbb{N} \to \mathbb{N}$. Then A is *ℓ-locally computable* from A_0 on a subset $C \subseteq \{0,1\}^*$ if there exists an oracle Turing machine f that runs in polynomial time and makes no more than ℓ queries to its oracle such that for every $x \in C$, the distribution of $A(x)$ is identical to the distribution of f with oracle $\pi_0 = A_0(x)$.

Theorem 10.1.2 ([16, Lemma 3.2])**.** *Let (P_0, V_0) be a PZK-PCP for some language L with query bound q^*, and let (P, V) be a PCP for a langauge M such that P is ℓ-locally computable from V on M. Then (P, V) is perfect zero-knowledge with query bound q^*/ℓ.*

Input: A string x and random coins r, as well as oracle access to a function π_0
Output: The interaction transcript of (P, V^*) on input x

$T \leftarrow [\,]$;
Run V^* on random coins r;
for *each query α that V^* makes* **do**
  $\beta \leftarrow f^{\pi_0}(\alpha)$;
  Push (α, β) onto T;
end
return (r, T);

Algorithm 10.1: A hybrid simulator for a locally-computable PCP [16, Construction 3.3]

Proof. We construct a simulator for (P, V) in Algorithm 10.2.

Let V^* be a malicious verifier for (P, V), and let $\pi \leftarrow P$ and $\pi_0 \leftarrow P_0$ be random variables. Since P is ℓ-locally computable with the function f, by definition we have that π is identically-distributed to $(f^{\pi_0}(\alpha))_{\alpha \in \mathrm{dom}(\pi_0)}$. Since all Algorithm 10.1 does

Input: A string x
Output: The interaction transcript of (P, V^*) on input x
1 Run $\overline{\mathrm{Sim}}_{A_{V^*}}(x)$ to obtain T_0 with random coins r;
2 Run $A_{V^*}(x, r)$ (Algorithm 10.1) using T_0 to answer its questions;
3 **return** (r, T);
Algorithm 10.2: A PZK simulator for a locally-computable PCP [16, Construction 3.4]

is compute $f^{\pi_0}(\alpha)$ for each query α, it will reproduce the same transcript as (P, V^*) would.

Next, since (P_0, V_0) is a PZK-PCP, and Algorithm 10.1 is a verifier for π_0, by definition there exists a simulator $\overline{\mathrm{Sim}}_{A_{V^*}}$ whose output is identically-distributed to $\mathrm{View}_{A_{V^*}, P_0}$, so long as A_{V^*} makes no more than q^* queries to π_0. Since P is ℓ-locally computable from P_0, it follows that Algorithm 10.1 makes no more than ℓ queries to π_0. Hence, Algorithm 10.2 makes no more than q^* queries to π_0 so long as V^* makes no more than q^*/ℓ queries to π. □

Corollary 10.1.3. *Let (P_0, V_0) be a PZK-PCPP for some language L with query bound q^* and proximity parameter δ, and let (P, V) be a PCPP for a langauge M with proximity parameter δ such that P is ℓ-locally computable from V on M. Then (P, V) is perfect zero-knowledge with query bound q^*/ℓ.*

Proof. □

10.2 Zero-knowledge proof composition

Theorem 10.2.1 ([16, Theorem 3.7])**.** *The construction in Theorem 5.3.1 is perfect zero-knowledge with query bound q^*/q_{out} if V_{out} is perfect zero-knowledge with query bound q^*.*

Input: A string $r \in \{0,1\}^{r_{\mathrm{out}}}$
Output: The function π_r
1 $I_{\mathrm{out}} \leftarrow Q_{\mathrm{out}}(x, r)$;
2 Compile D_{out} on input r into a circuit $C_{\mathrm{out}} : \{0,1\}^n \times \{0,1\}^{\ell_{\mathrm{out}}} \to \{0,1\}$;
3 Run $P_{\mathrm{in}}(C_{\mathrm{out}}, \pi_{\mathrm{out}}|_{I_{\mathrm{out}}})$ to get π_r;
4 **return** π_r;
Algorithm 10.3: An algorithm for π_r from π_0

Proof. All that is needed to prove this theorem is to show that Algorithm 5.1 preserves zero-knowledge, since we have already showed that it satisfies the conditions in Theorem 5.3.1. We do this by showing P_{comp} is q_{out}-locally computable from P_{out}.

We need to show that for any input $x \in L$, the distributions of $P(x)$ and $f^{\pi_0}(x)$ are identically distributed. Consider the function

$$f(O, r) = \begin{cases} \pi_0(r) & O = b_0 \\ \text{Algorithm 10.3} & O = b_r. \end{cases} \tag{10.1}$$

If $O = b_0$, then we make no more than one query to π_0. If $O = b_r$, then Algorithm 10.3 makes no more than $|I_{\text{out}}|$ queries to π_{out} (since that is the size of the domain of the restricted function); regardless of r we have that $|I_{\text{out}}| \leq q_{\text{out}}$ since I_{out} is a set of queries and q_{out} is the maximum number of queries that V_{out} makes. Hence f makes no more than q_{out} queries.

Lastly, so long as V_{out} runs in polynomial time, so must D_{out} and Q_{out}; compiling into a circuit is also known to take polynomial time. Lastly, the problem statement tells us P_{in} is guaranteed to run in polynomial time; it follows that f runs in polynomial time. Lastly, since Algorithm 10.3 uses its input r as randomness to all the algorithms it calls, it follows that f itself is deterministic (since all randomness comes from the choice of r). Hence, P_{comp} is q_{out}-locally computable from P_{out}.

Since P_{comp} is q_{out}-locally computable from P_{out}, by Theorem 10.1.2 we have that P_{comp} is perfect zero-knowledge with query bound q^*/q_{out}. □

10.3 Zero-knowledge alphabet reduction

Theorem 10.3.1 ([9, Lemma 2.13])**.** *Let L be a language with a PZK-PCP over the language $\{0,1\}^a$ such that*

$$L \in \mathsf{PZK\text{-}PCP}\begin{bmatrix} \textit{rand. complexity:} & r \\ \textit{query complexity:} & q \\ \textit{query bound:} & q^* \\ \textit{soundness error:} & \varepsilon \\ \textit{RS error:} & s \\ \textit{robustness param:} & \rho \end{bmatrix}.$$

Then L has a PZK-PCP over the language $\{0,1\}$ such that

$$L \in \mathsf{PZK\text{-}PCP}\begin{bmatrix} \textit{rand. complexity:} & r \\ \textit{query complexity:} & O(aq) \\ \textit{query bound:} & q^* \\ \textit{soundness error:} & \varepsilon \\ \textit{RS error:} & s \\ \textit{robustness param:} & \Omega(\rho) \end{bmatrix}.$$

Proof. We proved the non-zero-knowledge version of this as Theorem 5.4.1, so all that remains is to prove that Algorithm 5.2 preserves zero-knowledge.

First, we show P is 1-locally computable from P_a. Consider the function f (with oracle π_a) defined by

$$\begin{aligned} f^{\pi_a} &: \{b_\pi, b_\tau\} \times \operatorname{dom}(\pi_a) \times [n] \to \{0,1\} \\ f^{\pi_a}(O, \alpha, i) &\mapsto \begin{cases} \pi_a(\alpha) & O = b_\pi \\ \mathsf{ECC}(\pi_a(\alpha))_i & O = b_\tau. \end{cases} \end{aligned} \tag{10.2}$$

We show the distribution of P is the same as the distribution of f with oracle from P_a for all x.

The domain for our function is three values: O, a marker for either π_a or τ; α, the value we query; and i, the bit of the error-correcting code we wish to query. This is equivalent to a way to access the output of P from Algorithm 5.2; since it returns an ordered pair of two functions (π_a, τ), whenever we query it we need first to choose which of the two functions to query, the value we are querying them on, and then since we can only look at one bit, if we are querying τ we need to also determine which specific bit we are asking for. Hence, for any (O, α, i), the definition of f means that $\pi(O, \alpha, i) = f^{\pi_0}(O, \alpha, i)$. Hence the two are identically distributed and thus P is 1-locally computable from P_a.

Since P is 1-locally computable from P_a, it follows from Theorem 10.1.2 that P is perfect zero-knowledge with the same query bound as P_a. □

Corollary 10.3.2.

$$\mathsf{PZK\text{-}PCP}\left[\begin{array}{rl} \textit{rand. complexity:} & r \\ \textit{query complexity:} & q \\ \textit{query bound:} & q^* \\ \textit{soundness error:} & \varepsilon \\ \textit{RS error:} & s \\ \textit{robustness param:} & \Omega(1) \end{array}\right] \subseteq \mathsf{PZK\text{-}PCP}\left[\begin{array}{rl} \textit{rand. complexity:} & r + \log(n) \\ \textit{query complexity:} & q \\ \textit{query bound:} & q^*/q \\ \textit{soundness error:} & \varepsilon \\ \textit{RS error:} & s \\ \textit{robustness param:} & \Omega(1) \end{array}\right]. \tag{10.3}$$

Proof. □

10.4 Robust total-degree test

Theorem 10.4.1 ([27, Prop. 5.7])**.**

Algorithm 10.4: A robust low-degree test [27, Prop. 5.7]

10.5 PCPPs for polynomial summation

Definition 10.5.1 ([16, Def. 4.1])**.** The language Sum is the set of all ordered pairs $((\mathbb{F}, 1^m, 1^d, H, \gamma), F)$, where

- $\mathbb{F}$ is a finite field,
- $m, d \in \mathbb{N}$,
- $H \subseteq \mathbb{F}$,
- $\gamma \in F$, and

- $F \in \mathbb{F}^{\leq d}[X_{1,\ldots,m}]$ with

$$\sum_{b \in H^m} F(b) = \gamma.$$

Definition 10.5.2. For a fixed finite field $\mathbb{F}$, $m, d \in \mathbb{N}$, $H \subseteq \mathbb{F}$, and $\gamma \in \mathbb{F}$, the language $\mathsf{Sum}[\mathbb{F}, m, d, H, \gamma]$ is the subset of Sum where the co-named parameters in Definition 10.5.1 are equal to their fixed values.

For simplicity, we will often say $F \in \mathsf{Sum}[\mathbb{F}, m, d, H, \gamma]$ in cases where we mean $((\mathbb{F}, m, d, H, \gamma), F) \in \mathsf{Sum}[\mathbb{F}, m, d, H, \gamma]$ to reduce redundancy.

Theorem 10.5.3. *Let $\delta > 0$, $\mathbb{F}$ be a finite field, $H \subseteq \mathbb{F}$, $\gamma \in \mathbb{F}$, and $m, d \in \mathbb{N}$ such that $\frac{md}{|\mathbb{F}|} < \delta$ and $d > |H| + 1$. Then there exists a PCP of proximity for $\mathsf{Sum}[\mathbb{F}, m, d, H, \gamma]$ over the alphabet $\mathbb{F}^{m+1}$ with proximity parameter δ and robustness parameter $\rho = \Omega(\delta)$. Further, the verifier makes $O(|m|)$ queries to F and π and the proof length is $O(|\mathbb{F}|^m)$.*

Proof. We construct such an algorithm as Algorithm 10.5. To show this is a PCPP, we will first show that it will always accept when given a valid input and proof; following that we will show it will correctly reject for all inputs not near any polynomial in the language.

□

Theorem 10.5.4 ([16, Lemma 5.1])**.** *Let $\delta > 0$, $\mathbb{F}$ be a finite field, $H \subseteq \mathbb{F}$, $\gamma \in \mathbb{F}$, and $m, d \in \mathbb{N}$ such that $\frac{md}{|\mathbb{F}|} < \delta$ and $d > |H| + 1$. Then there exists a perfect zero-knowledge PCP of proximity for $\mathsf{Sum}[\mathbb{F}, m, d, H, \gamma]$ over the alphabet $\mathbb{F}^{m+1}$ with proximity parameter δ and robustness parameter $\rho = \Omega(\delta)$. Further, the verifier makes $O(|m|)$ queries to F and π and the proof length is $O(|\mathbb{F}|^m)$.*

Proof. We construct such an algorithm as Algorithm 10.6. First, we will show that this is a PCP of proximity by showing correctness in both acceptance and rejection (with high probability), and then we will demonstrate zero knowledge.

Let $F \in \mathsf{Sum}[\mathbb{F}, m, d, H, \gamma]$, and let (π_Σ, π_P) be the honest proof. In this case, as we showed in Theorem 10.5.3, the simulation of Algorithm 10.5 we do on line 11 will always succeed. To show that the equation in line 11 is true, note that from the definition of π_P, $(\pi_P(\alpha))_1 = Q(x)$ and $(\pi_P(\alpha))_{i+1} = T_i(\alpha)$ for all $i \geq 1$. Doing these substitutions, the right hand side of the equation is exactly $F(\alpha)$ plus the definition of R from line 8. Since F is defined in the problem statement to be a polynomial of degree d, the check in line 12 will always succeed. Further, Q and each T_i are defined to be multidegree-d m-variable polynomials and thus their degree will be less than md. Hence, the check in line 16 will pass. As such, when given valid inputs Algorithm 10.6 will always succeed.

□

10.6 A zero-knowledge PCP for NP and NEXP

Theorem 10.6.1 ([16, Theorem 6.3]). *For any query bound $q^*(n) \leq 2^{\mathsf{poly}(n)}$,*

$$\mathsf{NEXP} \subseteq \mathsf{PZK\text{-}PCP}_{\Sigma(n)}\left[\begin{array}{rl} \textit{rand. complexity:} & \log(n) + \log(q^*(n)) \\ \textit{query complexity:} & \mathsf{poly}(\log(n) + \log(q^*(n))) \\ \textit{query bound:} & q^*(n) \\ \textit{soundness error:} & \varepsilon \\ \textit{RS error:} & s \\ \textit{robustness param:} & \Omega(1) \end{array}\right].$$

where $|\Sigma(n)| = \mathsf{poly}(n, q)$.

Proof. □

Theorem 10.6.2 ([16, Theorem 6.3]). *For any query bound $q^*(n) \leq 2^{\mathsf{poly}(n)}$,*

$$\mathsf{NEXP} \subseteq \mathsf{PZK\text{-}PCP}_{\Sigma(n)}\left[\begin{array}{rl} \textit{rand. complexity:} & \mathsf{poly}(n) + \log(q^*(n)) \\ \textit{query complexity:} & \mathsf{poly}(n) \\ \textit{query bound:} & q^*(n) \\ \textit{soundness error:} & \varepsilon \\ \textit{RS error:} & s \\ \textit{robustness param:} & \Omega(1) \end{array}\right].$$

where $\Sigma(n)$ is any alphabet with $|\Sigma(n)| \in \mathsf{poly}(n, q)$.

Proof. We construct a PZK-PCP for O3SAT, a NEXP-complete language, in Algorithm 10.9. □

10.6.1 Reducing query complexity to constant

We are now finally able to present our analogues to the classical PCP theorem.

Theorem 10.6.3 ([16, Theorem 2]). $\mathsf{NP} \subseteq \mathsf{PZK\text{-}PCP}[\log(n), 1]$.

Proof. To show this, we will take the PCP for NP that we showed in Theorem 10.6.1, which has relatively weak bounds and an arbitrary alphabet, and transform it into one with the exact parameters we seek. We do this first through an alphabet reduction (as per Theorem 5.4.1) and then through proof-composing it with the algorithm for CktVal with Corollary 10.3.2, we can get a constant query-complexity PCP. Since several of these class inclusions involve modifying a small number of parameters in a relatively complex class, we will be highlighting the changed parameters in each inclusion statement.

/* Proof */
for $i \in \{1, \ldots, m-1\}$ **do**
  $g_i(X) \leftarrow \sum_{b \in H^{m-i}} F(X, b)$;
end
Define $\pi \colon \mathbb{F}^{m-1} \to \mathbb{F}^{m-1}$ by

$$\pi(c_1, \ldots, c_{m-2}, \alpha) = (g_1(\alpha), g_2(c_1, \alpha), \ldots, g_{m-1}(c_1, \ldots, c_{m-2}, \alpha))$$

for each $(c_1, \ldots, c_{m-2}, \alpha) \in \mathbb{F}^{m-1}$;
return π;
/* Verifier */
Sample $c \in \mathbb{F}^{m-1}$ at random;
for $\alpha \in \mathbb{F}$ **do**
  Query $\pi(c_1, \ldots, c_{m-2}, \alpha)$;
  Query $F(c_1, \ldots, c_{m-1}, \alpha)$;
end
for $i \in \{1, \ldots, m-1\}$ **do**
  if $g_i \notin \mathbb{F}[X_{1,\ldots,i}^{\leq d}]$ **then**
    reject;
  end
end
Check $\sum_{b \in H} g_1(b) = \gamma$;
for $i \in \{1, \ldots, m-2\}$ **do**
  Check

$$\sum_{b \in H} g_{i+1}(c_1, \ldots, c_i, b) = g_i(c_1, \ldots, c_i);$$

end
Check

$$\sum_{b \in H} F(c, b) = g_{m-1}(c);$$

Run Algorithm 10.4 on F, with proximity parameter $\delta_R = \min(\delta, 1/5)$;
Accept if and only if the prior test passes;

Algorithm 10.5: A robust PCPP for Sum [16, Construction 4.3]

/* Proof */

Sample $Q \in \mathbb{F}[X_{1,\dots,m}^{\leq d}]$ uniformly;
for $i \in \{1, \dots, m\}$ **do**
  Sample $T_i \in \mathbb{F}[X_{1,\dots,i-1}^{\leq d}, X_i^{\leq d-|H|}, X_{i+1,\dots,m}^{\leq d}]$ uniformly;
end
Define $\pi_P(x) = (Q(x), T_1(x), \dots, T_m(x))$;
Define $Z_H(X) = \prod_{a \in H}(X - a)$;
Define $Q_{\text{rev}}(X) = Q(X_{\text{rev}})$;
Define $R(X) = Q(X) - Q_{\text{rev}}(X) + \sum_{i=1}^{m} Z_H(X_i)T_i(X)$;
Compute the proof π_Σ for Algorithm 10.5 with explicit input $(\mathbb{F}, m, d, H, \gamma, \delta)$ and implicit input $F + R$;
return (π_Σ, π_P);

/* Verifier */

Emulate Algorithm 10.5 on input $F + R$ and proof π_Σ. To query $F + R$ at some $\alpha \in \mathbb{F}^m$, query $F(\alpha)$, $\pi_P(\alpha)$, and $\pi_P(\alpha_{\text{rev}})$, then compute

$$(F + R)(\alpha) = F(\alpha) + (\pi_P(\alpha))_1 - (\pi_P(\alpha_{\text{rev}}))_1 + \sum_{i=1}^{m} Z_H(\alpha_i)(\pi_P(\alpha))_{i+1};$$

Perform Algorithm 10.4 on F with proximity parameter $\delta_{RM} = \min(d, 1/5)$;
if *Algorithm 10.4 fails* **then**
  reject;
end
Perform Algorithm 10.4 on π_P with proximity parameter $\varepsilon_P = \delta_R/8$ and degree parameter $d_P = md$;
if *Algorithm 10.4 fails* **then**
  reject;
end
accept;

Algorithm 10.6: A zero-knowledge robust PCPP for Sum [16, Construction 5.2]

Algorithm 10.7: A PZK-PCP for NP [16, Theorem 6.3]

Algorithm 10.8: A simulator for Algorithm 10.7

/* Proof */

Let $A\colon \{0,1\}^n \to \{0,1\}$ be a satisfying assignment for B;

Choose $\hat{C} \in \mathbb{F}[X_{m_2+k}^{\leq 2(|H|-1)}]$ randomly such that

$$\sum_{c \in H^k} \hat{C}(b,c) = A(\gamma_2, b)$$

for all $b \in H^{m_2}$;

for $\tau \in \mathbb{F}^{m_1+3m_2+3}$ **do**

  Let π_τ be a PZK-PCPP for the claim

$$\sum_{\substack{z \in H^{m_1} \\ b_1,b_2,b_3 \in H^{m_2}}} \sum_{a_1,a_2,a_3 \in \{0,1\}} \sum_{c_1,c_2,c_3 \in H^k} L_{H_{m_1+3m_2} \times \{0,1\}^3, (z,b,a)}(\tau) h_{\hat{C}}(\tau, c_1, c_2, c_3) = 0;$$

end

return $(\pi_C, (\pi_\tau)_{\tau \in \mathbb{F}^{m_1+3m_2+3}})$;

/* Verifier */

Algorithm 10.9: A PZK-PCP for O3SAT [16, Construction 6.4]

Input: A query α to an oracle O, either π_C or π_τ

Output: The result of the query

if *The query request is to* π_C **then**

else

  if *This is the first query to* π_τ **then**

  end

end

Algorithm 10.10: A simulator for Algorithm 10.9 [16, Construction 6.7]

Let $q^*(n) \leq 2^{\mathsf{poly}(n)}$ be arbitrary. This will be the query complexity of our final PCP after we do all the class inclusions. As per Theorem 10.6.1, we know that

$$\mathsf{NP} \subseteq \mathsf{PZK\text{-}PCP}_{\Sigma(n)} \left[\begin{array}{rl} \text{rand. complexity:} & \log(n) + \log(\tilde{q}(n)) \\ \text{query complexity:} & \mathsf{poly}(\log(n) + \log(\tilde{q}(n))) \\ \text{query bound:} & \tilde{q}(n) \\ \text{soundness error:} & \varepsilon \\ \text{RS error:} & s \\ \text{robustness param:} & \Omega(1) \end{array} \right]. \tag{10.4}$$

for any $\tilde{q} \leq 2^{\mathsf{poly}(n)}$ and where $|\Sigma(n)| \in \mathsf{poly}(n, \tilde{q})$. Theorem 10.6.1 only guarantees this inclusion for alphabets of $\{0,1\}^a$, however a PCP over $\Sigma(n)$ is equivalent to a PCP over $\{0,1\}^{\log_2(|\Sigma(n)|)}$ by a simple relabeling of alphabet items, so this inclusion still holds.

Next, we perform an alphabet reduction: by Theorem 10.3.1,

$$\begin{aligned}\mathsf{PZK\text{-}PCP}_{\Sigma(n)}&\left[\begin{array}{rl}\text{rand. complexity:}&\log(n)+\log(\tilde{q}(n))\\ \text{query complexity:}&\mathsf{poly}(\log(n)+\log(\tilde{q}(n)))\\ \text{query bound:}&\tilde{q}(n)\\ \text{soundness error:}&\varepsilon\\ \text{RS error:}&s\\ \text{robustness param:}&\Omega(1)\end{array}\right]\\ \subseteq\mathsf{PZK\text{-}PCP}_{\{0,1\}}&\left[\begin{array}{rl}\text{rand. complexity:}&\log(n)+\log(\tilde{q}(n))\\ \text{query complexity:}&\mathsf{poly}(\log(n)+\log(\tilde{q}(n)))\\ \text{query bound:}&\tilde{q}(n)\\ \text{soundness error:}&\varepsilon\\ \text{RS error:}&s\\ \text{robustness param:}&\Omega(1)\end{array}\right].\end{aligned} \tag{10.5}$$

Next, we perform proof composition. For any language $L \in \mathsf{NP}$, let $Q(n) \in \mathsf{poly}(\log(n) + \log(\tilde{q}))$ be the query complexity of the PZK-PCP within the parameters of the right-hand side of Equation (10.5) that recognizes L. Since $\tilde{q}$ was arbitrary, we can define it to be whatever we like; hence let $\tilde{q}(n)$ be a polynomial in q^* and n large enough that for all $n \in \mathbb{N}$, $\tilde{q}(n)/Q(n) \geq q^*(n)$. By Corollary 10.3.2, we have that

$$\begin{aligned}\mathsf{PZK\text{-}PCP}_{\{0,1\}}&\left[\begin{array}{rl}\text{rand. complexity:}&\log(n)+\log(\tilde{q}(n))\\ \text{query complexity:}&\mathsf{poly}(\log(n)+\log(\tilde{q}(n)))\\ \text{query bound:}&\tilde{q}(n)\\ \text{soundness error:}&\varepsilon\\ \text{RS error:}&s\\ \text{robustness param:}&\Omega(1)\end{array}\right]\\ \subseteq\mathsf{PZK\text{-}PCP}_{\{0,1\}}&\left[\begin{array}{rl}\text{rand. complexity:}&\log(n)+\log(q^*(n))\\ \text{query complexity:}&1\\ \text{query bound:}&q^*(n)\\ \text{soundness error:}&\varepsilon\end{array}\right].\end{aligned} \tag{10.6}$$

At the beginning of the proof, we said q^* was arbitrary; hence we can set $q^* \in O(1)$. Thus, we get

$$\mathsf{PZK\text{-}PCP}_{\{0,1\}}\left[\begin{array}{rl}\text{rand. complexity:}&\log(n)+\log(q^*(n))\\ \text{query complexity:}&1\\ \text{query bound:}&q^*(n)\\ \text{soundness error:}&\varepsilon\end{array}\right]\subseteq\mathsf{PZK\text{-}PCP}[\log(n),1]. \tag{10.7}$$

By combining the inclusions in Equations (10.4) to (10.7), we get that $\mathsf{NP} \subseteq \mathsf{PZK\text{-}PCP}[\log(n), 1]$, as desired. □

Theorem 10.6.4 ([16, Theorem 7.1])**.** $\mathsf{NEXP} \subseteq \mathsf{PZK\text{-}PCP}[\mathsf{poly}(n), 1]$.

Proof. Broadly speaking, this proof will proceed in the same style as the proof for Theorem 10.6.3.

Let $q^* \leq 2^{\mathsf{poly}(n)}$ be arbitrary. This will be the query complexity of our final PCP after we do all the class inclusions. As per Theorem 10.6.2, we know that

$$\mathsf{NEXP}\subseteq\mathsf{PZK\text{-}PCP}_{\Sigma(n)}\left[\begin{array}{rl}\text{rand. complexity:}&\mathsf{poly}(n)+\log(\tilde{q}(n))\\ \text{query complexity:}&\mathsf{poly}(n)\\ \text{query bound:}&\tilde{q}(n)\\ \text{soundness error:}&\varepsilon\\ \text{RS error:}&s\\ \text{robustness param:}&\Omega(1)\end{array}\right]. \tag{10.8}$$

for any $\tilde{q} \leq 2^{\mathsf{poly}(n)}$ and where $|\Sigma(n)| \in \mathsf{poly}(n, \tilde{q})$. As before, Theorem 10.6.1 only guarantees this inclusion for alphabets of $\{0,1\}^a$, however a PCP over $\Sigma(n)$ is equivalent to a PCP over $\{0,1\}^{\log_2(|\Sigma(n)|)}$ by a simple relabeling of alphabet items, so this inclusion still holds.

Next, we perform an alphabet reduction: by Theorem 10.3.1,

$$\begin{aligned}\mathsf{PZK\text{-}PCP}_{\Sigma(n)}&\left[\begin{array}{rl}\text{rand. complexity:}&\mathsf{poly}(n)+\log(\tilde{q}(n))\\ \text{query complexity:}&\mathsf{poly}(n)\\ \text{query bound:}&\tilde{q}(n)\\ \text{soundness error:}&\varepsilon\\ \text{RS error:}&s\\ \text{robustness param:}&\Omega(1)\end{array}\right]\\ \subseteq \mathsf{PZK\text{-}PCP}_{\{0,1\}}&\left[\begin{array}{rl}\text{rand. complexity:}&\mathsf{poly}(n)+\log(\tilde{q}(n))\\ \text{query complexity:}&\mathsf{poly}(n,\log(\tilde{q}(n)))\\ \text{query bound:}&\tilde{q}(n)\\ \text{soundness error:}&\varepsilon\\ \text{RS error:}&s\\ \text{robustness param:}&\Omega(1)\end{array}\right].\end{aligned} \tag{10.9}$$

Next, we perform proof composition. For any language $L \in \mathsf{NP}$, let $Q(n) \in \mathsf{poly}(n)$ be the query complexity of the PZK-PCP within the parameters of the right-hand side of Equation (10.5) that recognizes L. Since $\tilde{q}$ was arbitrary, we can define it to be whatever we like; hence let $\tilde{q}(n) = q^*(n) \cdot Q(n)$ for all $n \in \mathbb{N}$. By Corollary 10.3.2, we have that

$$\begin{aligned}\mathsf{PZK\text{-}PCP}_{\{0,1\}}&\left[\begin{array}{rl}\text{rand. complexity:}&\mathsf{poly}(n)\\ \text{query complexity:}&\mathsf{poly}(n,\log(\tilde{q}(n)))\\ \text{query bound:}&\tilde{q}(n)\\ \text{soundness error:}&\varepsilon\\ \text{RS error:}&s\\ \text{robustness param:}&\Omega(1)\end{array}\right]\\ \subseteq \mathsf{PZK\text{-}PCP}_{\{0,1\}}&\left[\begin{array}{rl}\text{rand. complexity:}&\mathsf{poly}(n)+\log(q^*(n))\\ \text{query complexity:}&1\\ \text{query bound:}&q^*(n)\\ \text{soundness error:}&\varepsilon\end{array}\right].\end{aligned} \tag{10.10}$$

At the beginning of the proof, we said q^* was arbitrary; hence we can set $q^* \in O(1)$. Thus, we get

$$\mathsf{PZK\text{-}PCP}_{\{0,1\}}\left[\begin{array}{rl}\text{rand. complexity:}&\mathsf{poly}(n)+\log(q^*(n))\\ \text{query complexity:}&1\\ \text{query bound:}&q^*(n)\\ \text{soundness error:}&\varepsilon\end{array}\right] \subseteq \mathsf{PZK\text{-}PCP}[\mathsf{poly}(n), 1]. \tag{10.11}$$

By combining the inclusions in Equations (10.8) to (10.11), we get that $\mathsf{NEXP} \subseteq \mathsf{PZK\text{-}PCP}[\mathsf{poly}(n), 1]$, as desired. □

Appendix A

More on extension polynomials

In this appendix, we will work through some of the algebra we mentioned but did not go into detail about in Section 1.3.

A.1 A proof of Equation (1.6)

Our goal is to demonstrate the following:

$$[x = y] = \prod_{i=1}^{m} \left(\sum_{\omega \in H} \left(\prod_{\gamma \in H \setminus \{\omega\}} \frac{(x_i - \gamma)(y_i - \gamma)}{(\omega - \gamma)^2} \right) \right) \tag{A.1}$$

for all $x, y \in H^n$. We will do this in two cases: one where $x = y$ and one where $x \neq y$.

First, assume $x \neq y$ (so we want to show $\delta_y(x) = 0$). In this case, there exists at least one i where $x_i \neq y_i$. For this i, for each ω there exists some $\gamma \in H \setminus \{\omega\}$ such that either $x_i = \gamma$ or $y_i = \gamma$.[1] As such, it follows that either $(x_i - \gamma) = 0$ or $(\gamma_i - \gamma) = 0$. Hence, for this i the sum will be entirely over zero terms (since there will be at least one zero term in the product for each ω). As such, this means that the ith term of our outermost product is 0, and hence the entire product is 0, as desired.

When $x = y$ (and so we want to show $\delta_y(x) = 1$), the above equation simplifies to

$$[x = y] = \prod_{i=1}^{m} \left(\sum_{\omega \in H} \left(\prod_{\gamma \in H \setminus \{\omega\}} \frac{(x_i - \gamma)^2}{(\omega - \gamma)^2} \right) \right) \tag{A.2}$$

Whenever $\omega \neq x_i$, the innermost product becomes 0 since there will be a term where $\gamma = x_i$. Hence, we can simplify this further to

$$[x = y] = \prod_{i=1}^{m} \left(\prod_{\gamma \in H \setminus \{\omega\}} \frac{(x_i - \gamma)^2}{(x_i - \gamma)^2} \right). \tag{A.3}$$

Since $\gamma \neq x_i$, we can simplify the fraction to 1; since we have two nested products it follows that the equation as a whole simplifies to 1.

[1] This piece fails in the case where $x_i = y_i$, since if $\omega = x_i = y_i$ neither of the terms will ever be zero.

A.2 Algebra behind Equation (1.8)

Our goal is to show that the equation in Equation (1.5) simplifies to that of Equation (1.8) when $H = \{0,1\}^n$.

As a refresher, our starting equation has the form

$$\delta_y(x) = \prod_{i=1}^{m}\left(\sum_{\omega\in H}\left(\prod_{\gamma\in H\setminus\{\omega\}}\frac{(x_i-\gamma)(y_i-\gamma)}{(\omega-\gamma)^2}\right)\right). \tag{A.4}$$

We start by manually substituting the outer sum:

$$\delta_y(x) = \prod_{i=1}^{m}\left(\left(\prod_{\gamma\in\{0,1\}\setminus\{0\}}\frac{(x_i-\gamma)(y_i-\gamma)}{(\omega-\gamma)^2}\right)+\left(\prod_{\gamma\in\{0,1\}\setminus\{1\}}\frac{(x_i-\gamma)(y_i-\gamma)}{(\omega-\gamma)^2}\right)\right). \tag{A.5}$$

Next, notice that the inner products are actually each over one term, so we can manually substitute there:

$$\delta_y(x) = \prod_{i=1}^{m}\left(\frac{(x_i-1)(y_i-1)}{(0-1)^2}+\frac{(x_i-0)(y_i-0)}{(1-0)^2}\right). \tag{A.6}$$

Next, we simplify, taking note that the denominator of both fractions is 1:

$$\delta_y(x) = \prod_{i=1}^{m}((x_i-1)(y_i-1)+x_iy_i). \tag{A.7}$$

From here, we take advantage of the fact that $y\in\{0,1\}^n$; here we split our product into two smaller products: one where $y_i = 0$ and one where $y_i = 1$.

$$\delta_y(x) = \left(\prod_{i:y_i=0}(x_i-1)(0-1)+0x_i\right)\left(\prod_{i:y_i=1}(x_i-1)(1-1)+x_i1\right). \tag{A.8}$$

Finally, we simplify, bringing us to Equation (1.8).

$$\delta_y(x) = \left(\prod_{i:y_i=0}(1-x_i)\right)\left(\prod_{i:y_i=1}x_i\right). \tag{A.9}$$

Appendix B

More on Lemma 3.2.3

Definition B.0.1. An *alternating Turing machine* is

Proof of Lemma 3.2.3 as written in [5]. Let L be a PSPACE-robust language. Let $g_n(x_1, \ldots, x_n)$ be the multilinear extension of the characteristic function of $L_n = L \cap \{0,1\}^n$. Clearly, $L \in \mathsf{P}^g$, where $g = \{g_n \mid n \geq 0\}$. We will describe an alternating polynomial-time Turing machine with access to L computing g. First guess the value $z = g_n(x_1, \ldots, x_n)$. Then existentially guess the linear function $h_1(y) = g(y, x_2, \ldots, x_n)$ and verify that $h_1(x_1) = z$. Then universally choose $t_1 \in \{0,1\}$ and existentially guess the linear function $h_2(y) = g(t_1, y, x_3, \ldots, x_n)$. Keep repeating this process until we have specified $t_1, \ldots, t_n$ and then verify $t_1, \ldots, t_n \in L$. Since a PSPACE machine can simulate an alternating polynomial-time Turing machine, if L is PSPACE-robust then g is Turing-reducible to L. □

Bibliography

[1] Scott Aaronson and Avi Wigderson. "Algebrization: A New Barrier in Complexity Theory". In: *ACM Trans. Comput. Theory* 1.1 (Feb. 2009). ISSN: 1942-3454. DOI: 10.1145/1490270.1490272.

[2] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. 1st. USA: Cambridge University Press, 2009. ISBN: 978-0-521-42426-4. DOI: 10.5555/1540612.

[3] Sanjeev Arora and Shmuel Safra. "Probabilistic checking of proofs: a new characterization of NP". In: *J. ACM* 45.1 (Jan. 1998), pp. 70–122. ISSN: 0004-5411. DOI: 10.1145/273865.273901. URL: https://doi.org/10.1145/273865.273901.

[4] Sanjeev Arora et al. "Proof verification and the hardness of approximation problems". In: *J. ACM* 45.3 (May 1998), pp. 501–555. ISSN: 0004-5411. DOI: 10.1145/278298.278306. URL: https://doi.org/10.1145/278298.278306.

[5] L. Babai, L. Fortnow, and C. Lund. "Nondeterministic exponential time has two-prover interactive protocols". In: *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*. 1990, 16–25 vol.1. DOI: 10.1109/FSCS.1990.89520.

[6] László Babai and Lance Fortnow. "Arithmetization: A new method in structural complexity theory". In: *computational complexity* 1.1 (Mar. 1991), pp. 41–66. ISSN: 1420-8954. DOI: 10.1007/BF01200057. URL: https://link.springer.com/article/10.1007/BF01200057.

[7] Theodore Baker, John Gill, and Robert Solovay. "Relativizations of the $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ Question". In: *SIAM Journal on Computing* 4.4 (1975), pp. 431–442. DOI: 10.1137/0204037. eprint: https://doi.org/10.1137/0204037. URL: https://doi.org/10.1137/0204037.

[8] Michael Ben-Or et al. "Multi-prover interactive proofs: how to remove intractability assumptions". In: *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*. STOC '88. Chicago, Illinois, USA: Association for Computing Machinery, 1988, pp. 113–131. ISBN: 0897912640. DOI: 10.1145/62212.62223. URL: https://doi.org/10.1145/62212.62223.

[9] Eli Ben-Sasson et al. "Robust PCPs of proximity, shorter PCPs and applications to coding". In: *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*. STOC '04. Chicago, IL, USA: Association for Computing Machinery, 2004, pp. 1–10. ISBN: 1581138520. DOI: 10.1145/1007352.1007361. URL: https://doi.org/10.1145/1007352.1007361.

[10] Eli Ben-Sasson et al. "Zero Knowledge Protocols from Succinct Constraint Detection". In: *Theory of Cryptography*. Ed. by Yael Kalai and Leonid Reyzin. Cham: Springer International Publishing, 2017, pp. 172–206. ISBN: 978-3-319-70503-3.

[11] Alessandro Chiesa et al. "Spatial Isolation Implies Zero Knowledge Even in a Quantum World". In: *J. ACM* 69.2 (Jan. 2022). ISSN: 0004-5411. DOI: 10.1145/3511100. URL: https://doi.org/10.1145/3511100.

[12] Stephen A. Cook. "A hierarchy for nondeterministic time complexity". In: *Journal of Computer and System Sciences* 7.4 (1973), pp. 343–353. ISSN: 0022-0000. DOI: 10.1016/S0022-0000(73)80028-5. URL: https://www.sciencedirect.com/science/article/pii/S0022000073800285.

[13] Stephen A. Cook. "The complexity of theorem-proving procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. ISBN: 9781450374644. DOI: 10.1145/800157.805047. URL: https://doi.org/10.1145/800157.805047.

[14] Oded Goldreich. *Foundations of Cryptography*. Vol. 1. 1st ed. Cambridge University Press, 2001. 372 pp. ISBN: 978-0-511-54689-1. DOI: 10.1017/CBO9780511546891.

[15] Oded Goldreich and Liav Teichner. "Super-Perfect Zero-Knowledge Proofs". In: *Computational Complexity and Property Testing: On the Interplay Between Randomness and Computation*. Ed. by Oded Goldreich. Cham: Springer International Publishing, 2020, pp. 119–140. ISBN: 978-3-030-43662-9. DOI: 10.1007/978-3-030-43662-9_8. URL: https://doi.org/10.1007/978-3-030-43662-9_8.

[16] Tom Gur, Jack O'Connor, and Nicholas Spooner. *A Zero-Knowledge PCP Theorem*. 2024. arXiv: 2411.07972 [cs.CC]. URL: https://arxiv.org/abs/2411.07972.

[17] Tom Gur, Jack O'Connor, and Nicholas Spooner. "Perfect Zero-Knowledge PCPs for #P". In: *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*. STOC 2024. Vancouver, BC, Canada: Association for Computing Machinery, 2024, pp. 1724–1730. ISBN: 9798400703836. DOI: 10.1145/3618260.3649698. URL: https://doi.org/10.1145/3618260.3649698.

[18] R. W. Hamming. "Error detecting and error correcting codes". In: *The Bell System Technical Journal* 29.2 (1950), pp. 147–160. DOI: 10.1002/j.1538-7305.1950.tb00463.x.

[19] Johan Håstad. "Some optimal inapproximability results". In: *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*. STOC '97. El Paso, Texas, USA: Association for Computing Machinery, 1997, pp. 1–10. ISBN: 0897918886. DOI: 10.1145/258533.258536. URL: https://doi.org/10.1145/258533.258536.

[20] Kenneth E. Iverson. *A Programming Language*. 1st ed. John Wiley and Sons, Inc., 1962. 286 pp. ISBN: 978-0471430148. URL: https://dl.acm.org/doi/10.5555/1098666.

[21] Zhengfeng Ji et al. "MIP* = RE". In: *Commun. ACM* 64.11 (Oct. 2021), pp. 131–138. ISSN: 0001-0782. DOI: 10.1145/3485628. URL: https://dl.acm.org/doi/10.1145/3485628.

[22] Zhengfeng Ji et al. "Quantum soundness of the classical low individual degree test". In: *CoRR* abs/2009.12982 (2020). arXiv: 2009.12982. URL: https://arxiv.org/abs/2009.12982.

[23] Ali Juma et al. "The Black-Box Query Complexity of Polynomial Summation". In: *Comput. Complex.* 18.1 (Apr. 2009), pp. 59–79. ISSN: 1016-3328. DOI: 10.1007/s00037-009-0263-7. URL: https://doi.org/10.1007/s00037-009-0263-7.

[24] Yael Tauman Kalai and Ran Raz. "Interactive PCP". In: *Proceedings of the 35th International Colloquium on Automata, Languages and Programming, Part II*. ICALP '08. Reykjavik, Iceland: Springer-Verlag, 2008, pp. 536–547. ISBN: 9783540705826. DOI: 10.1007/978-3-540-70583-3_44. URL: https://doi.org/10.1007/978-3-540-70583-3_44.

[25] Donald E. Knuth. "Two Notes on Notation". In: *The American Mathematical Monthly* 99.5 (1992), pp. 403–422. ISSN: 00029890, 19300972. URL: http://www.jstor.org/stable/2325085 (visited on 11/19/2024).

[26] Carsten Lund et al. "Algebraic methods for interactive proof systems". In: *J. ACM* 39.4 (Oct. 1992), pp. 859–868. ISSN: 0004-5411. DOI: 10.1145/146585.146605. URL: https://doi.org/10.1145/146585.146605.

[27] Orr Paradise. "Smooth and Strong PCPs". In: *Computational Complexity* 30.1 (Jan. 2021). ISSN: 1420-8954. DOI: 10.1007/s00037-020-00199-3. URL: https://link.springer.com/article/10.1007/s00037-020-00199-3.

[28] Walter Rudin. *Principles of Mathematical Analysis*. 3rd ed. McGraw-Hill, 1976. 342 pp. ISBN: 978-0-07-085613-4.

[29] Walter J. Savitch. "Relationships between nondeterministic and deterministic tape complexities". In: *Journal of Computer and System Sciences* 4.2 (1970), pp. 177–192. ISSN: 0022-0000. DOI: 10.1016/S0022-0000(70)80006-X. URL: https://www.sciencedirect.com/science/article/pii/S002200007080006X.

[30] Adi Shamir. "IP = PSPACE". In: *J. ACM* 39.4 (Oct. 1992), pp. 869–877. ISSN: 0004-5411. DOI: 10.1145/146585.146609. URL: https://doi.org/10.1145/146585.146609.

[31] Michael Sipser. *Introduction to the Theory of Computation*. 1st ed. International Thomson Publishing, 1996. 396 pp. ISBN: 978-0-534-94728-6. DOI: 10.1145/230514.571645.

[32] L. J. Stockmeyer and A. R. Meyer. "Word problems requiring exponential time". In: *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*. STOC '73. Austin, Texas, USA: Association for Computing Machinery, 1973, pp. 1–9. ISBN: 9781450374309. DOI: 10.1145/800125.804029. URL: https://doi.org/10.1145/800125.804029.

[33] A. M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of The London Mathematical Society* 42.1 (1936), pp. 230–265. DOI: 10.1112/PLMS/S2-42.1.230.

Index