Thesis Draft: Algebrization

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

November 25, 2024

Approved for the Committee
(Mathematics and Computer Science)

Zajj Daugherty	Adam Groce

Contents

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.

Definition 1.1.1 ([12, Def. 3.1])**.** A *Turing machine* 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. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ 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* [14], 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.

1.2 Complexity classes

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.

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 1.2.1 ([2, Def. 1.19])**.** Let $f : \mathbb{N} \to \mathbb{N}$ 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 1.2.2 ([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.

Example 1.2.2.1. 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 1.2.3 ([12, 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 1.2.3.1. 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 1.2.4 ([12, Def. 7.19])**.** NP *is exactly the class of all languages verifiable by a* P*-time Turing machine.*

Example 1.2.4.1. The language SAT we defined in Example 1.2.3.1 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.

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 1.2.5 ([2, Def. 4.1])**.** Let $f : \mathbb{N} \to \mathbb{N}$ 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 1.2.6 ([2, Def. 4.1])**.** Let $f : \mathbb{N} \to \mathbb{N}$ 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 1.2.7 ([2, Def. 4.5])**.** The complexity class PSPACE is the class

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

Definition 1.2.8 ([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 1.2.9 (Savitch's theorem; [11])**.** PSPACE = NPSPACE.

Upon seeing this, one might ask why it is that we believe $\mathsf{P} \neq \mathsf{NP}$ 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 1.2.9.1. 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 1.2.10 ([12, Def. 7.29])**.** A language A is *polynomial-time reducible* to a language B if a polynomial-time computable function $f : \Sigma^* \to \Sigma^*$ 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: An algorithm to reduce A to B

Definition 1.2.11 ([12, Def. 7.34])**.** A language L is NP-complete if $L \in \mathsf{NP}$ and every $A \in \mathsf{NP}$ 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 $\mathsf{P} = \mathsf{NP}$ if and only if *any* NP-complete language is in P.

Example 1.2.11.1. A famous result of Cook [6], 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 1.2.3.1 is NP-complete.

Just as we have NP-completeness for time complexity, we also have notions of completeness for space complexity. Since $\mathsf{PSPACE} = \mathsf{NPSPACE}$, instead of calling the class NPSPACE-complete, we call it PSPACE-complete.

Definition 1.2.12 ([12, 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 1.2.12.1. A result of Stockmeyer and Meyer [13] is that the language we defined in Example 1.2.9.1 is PSPACE-complete.

1.3 Polynomials

Definition 1.3.1 ([9])**.** Let P be a mathematical statement. The function $[P]$ is the the function

$$[P] = \begin{cases} 1 & P \text{ is true} \\ 0 & \text{otherwise.} \end{cases} \tag{1.1}$$

This is called the *Iverson bracket*, after its inventor Kenneth Iverson, who originally included it in the programming language APL (although originally using parentheses) [7, 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 $\mathbb{F}[x_{1,\ldots,m}]$.

Definition 1.3.2 ([1, p. 8])**.** The *multidegree* of a multivariate polynomial p, written $\mathrm{mdeg}(d)$, 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 $x_1^2 x_2 + x_2^2$. The multidegree of this polynomial is 2, while its degree is 3.

We denote by $\mathbb{F}[x_{1,\ldots,m}^{\leq d}]$ the subset of $\mathbb{F}[x_{1,\ldots,m}]$ 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.

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.

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 $f : H^m \to \mathbb{F}$ be a function. An *extension polynomial* of f is any polynomial $f' \in \mathbb{F}[x_{1,\ldots,m}]$ such that $f(h) = f'(h)$ for all $h \in H$.

It turns out that this polynomial needs only to be of a surprisingly low multidegree. Since polynomials of lower degree are generally easier to compute, we would like to have some measure of what a "small" polynomial actually is in this context.

Definition 1.3.5 ([5, §5.1])**.** Let $\mathbb{F}$ be a finite field, $H \subseteq \mathbb{F}$, $m \in \mathbb{N}$ a number, and $f : H^m \to \mathbb{F}$ be a function. A *low-degree extension* $\hat{f}$ 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 ([5, §5.1])**.** *Let $\mathbb{F}$ be a finite field, $H \subseteq \mathbb{F}$, $m \in \mathbb{N}$ a number, and $f : H^m \to \mathbb{F}$. Then a low-degree extension $\hat{f}$ of f is the function*

$$\hat{f}(x) = \sum_{\beta \in H^m} \delta_\beta(x) f(\beta), \tag{1.2}$$

where δ is the polynomial

$$\delta_x(y) = \prod_{i=1}^{m} \left(\sum_{\omega \in H} \left(\prod_{\gamma \in H \setminus \{\omega\}} \frac{(x_i - \gamma)(y_i - \gamma)}{(\omega - \gamma)^2} \right) \right). \tag{1.3}$$

Proof. First, we must show $\hat{f}$ has multidegree $|H| - 1$. First, note that $\hat{f}$ is a linear combination of some $\delta_x$es; hence asking about the multidegree of $\hat{f}$ 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 $\hat{f}(x)$ agrees with $f(x)$ on H, we first should look at $\delta_\beta(x)$. In particular, for all $x, y \in H^m$,

$$\delta_y(x) = [x = y] = \delta_{xy}. \tag{1.4}$$

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 (1.3) 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 $\hat{f}(x)$ is the term where $\beta = x$; thus $\hat{f}(x) = f(x)$. Hence, $\hat{f}$ 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 $f : \{0,1\}^m \to \mathbb{F}$. Then*

$$\hat{f}(x) = \sum_{\beta \in \{0,1\}^m} \delta_\beta(x) f(\beta) \tag{1.5}$$

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

$$\delta_y(x) = \left(\prod_{i : y_i = 1} x_i \right) \left(\prod_{i : y_i = 0} (1 - x_i) \right). \tag{1.6}$$

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 (1.6) is much more manageable than the form in Equation (1.3), 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 (1.6) 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.[1]

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

$$\sum_{y \in \{0,1\}^n} a_y \delta_y = 0 \tag{1.7}$$

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 $p : \mathbb{F}^n \to \mathbb{F}$ 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. □

[1] 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.

Lemma 1.3.10 ([1, Lemma 4.5]). *Let $\mathcal{F}$ be a collection of fields. Let $f : \{0,1\}^n \to \{0,1\}$ be a Boolean function, and for every $\mathbb{F} \in \mathcal{F}$, let $p_{\mathbb{F}} : \mathbb{F}^n \to \mathbb{F}$ 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 $f' : \{0,1\}^n \to \{0,1\}$ that agree with f on B, there exist multiquadratic polynomials $p'_{\mathbb{F}} : \mathbb{F}_n \to \mathbb{F}$ (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 $y \in \mathcal{Y}_{\mathbb{F}}$.*

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

$$p'_{\mathbb{F}}(x) = p_{\mathbb{F}}(x) + \sum_{z\in G} (f'(z) - f(z)) u_{\mathbb{F},z}(x). \tag{1.8}$$

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 (1.8) 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. □

Lemma 1.3.11 ([8, Lemma 7]). *Let $m(x_1, \ldots, x_n)$ be a multilinear monomial. Over a field of characteristic other than 2, we have*

$$\sum_{b\in\{-1,1\}} m(b) = 0. \tag{1.9}$$

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

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

□

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 $A_m : \{0,1\}^m \to \{0,1\}$, 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.

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

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

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

$$\begin{aligned} f_n : \{0,1\}^n &\to \{0,1\} \\ a_1 a_2 \cdots a_n &\mapsto a_n. \end{aligned} \tag{2.3}$$

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

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.

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.

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.

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. □

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 $A : \{0,1\}^n \to \{0,1\}$ 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 $f : \{0,1\}^N \to \{0,1\}$ 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 $A : \{0,1\}^n \to \{0,1\}$ 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 $f : \{0,1\}^N \to \{0,1\}$ 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 $f : \{0,1\}^N \to \{0,1\}$ 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 ([4, 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 [11] (which we briefly discussed as Theorem 1.2.9) 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 ([4, 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 ([4, 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 ([4, 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 give the following algorithm to construct B:

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: An algorithm for constructing B

Now that we have presented the algorithm, let us demonstrate its 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

[1] This argument style is named after *Cantor's diagonal argument*, which was originally used to prove that the real numbers are uncountable [10, Thm. 2.14].

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 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, $\mathsf{P}^B \neq \mathsf{NP}^B$. □

2.4 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 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] 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.

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.

Definition 3.0.1 ([1, Def. 2.2])**.** Let $A_m : \{0,1\}^m \to \{0,1\}$ be a Boolean function and let $\mathbb{F}$ be a finite field. Then an *extension* of A_m over $\mathbb{F}$ is a polynomial $\tilde{A}_{m,\mathbb{F}} : \mathbb{F}^m \to \mathbb{F}$ 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 $\tilde{A}_{m,\mathbb{F}} : \mathbb{F}^m \to \mathbb{F}$, 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:

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

An extension of that function is the polynomial

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

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

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

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 $f : \{0,1\}^N \to \{0,1\}$ 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 $A : \{0,1\}^n \to \{0,1\}$ and every finite field extension $\tilde{A} : \mathbb{F}^n \to \mathbb{F}$ 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. □

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

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

Consider the function

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

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

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

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

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

Since $p(b) \geq 0$ for all $b \in \{0,1\}^n$, it follows that $p(1/2, \ldots, 1/2)$ 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 ([3, 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 ([3, 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 [3] 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

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

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

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

$z' \leftarrow 0$;
for $k \in \{0,1\}^n$ **do**
  Simulate L on input k;
  if $k \in L$ **then**
    // Compute $d_k(x)$
    $d \leftarrow 1$;
    for i *from* 1 *to* n **do**
      if $k_i = 1$ **then**
        $d \leftarrow d \cdot x_i$;
      else
        $d \leftarrow d \cdot (1 - x_i)$;
      end
    end
    $z' \leftarrow z' + d$;
  end
end
return *whether* $z = z'$;

Algorithm 3: Determiner for $\tilde{L}$

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.

First, we demonstrate that the above algorithm 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 is z'. Since $A(x_1, \ldots, x_n) \in \{0, 1\}$, each term in the sum in Equation (1.5) 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 is in PSPACE^L, and thus is in P^L.

Next, we show that Algorithm 3 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 (1.6). 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 (1.5) 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 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, reusing our argument from Theorem 2.3.1, we have

$$\mathsf{NP}^{\tilde{A}} = \mathsf{NP}^{\mathsf{PSPACE}} = \mathsf{PSPACE} = \mathsf{P}^A. \tag{3.12}$$

□

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 $L(B) \in \mathsf{NP}^A$. We shall construct $\tilde{A}$ using the following algorithm:

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 $p_i(n) < 2^n$;
  $T_j \leftarrow \bigcup_{j<i} S_j$;
  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 4: An algorithm for constructing $\tilde{A}$

As before, we will start by demonstrating soundness and then move on to why the constructed oracle provides the separation we seek. □

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 (1.4)

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.

A.2 Algebra behind Equation (1.6)

Our goal is to show that the equation in Equation (1.3) simplifies to that of Equation (1.6) when $H = \{0, 1\}^n$.

[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.

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 (1.6).

$$\delta_y(x) = \left(\prod_{i:y_i=0}(1-x_i)\right)\left(\prod_{i:y_i=1}x_i\right). \tag{A.9}$$

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 [3]. 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] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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).

[10] Walter Rudin. *Principles of Mathematical Analysis*. 3rd ed. McGraw-Hill, 1976. 342 pp. ISBN: 978-0-07-085613-4.

[11] 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.

[12] 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.

[13] 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.

[14] 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