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 6, 2024

Approved for the Committee
(Mathematics and Computer Science)

Zajj Daugherty	Adam Groce

Table of 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 ([6, 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*, 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.

Definition 1.2.1. The complexity class P is

Definition 1.2.2. The complexity class NP is

Definition 1.2.3. The complexity class PSPACE is

Definition 1.2.4. The complexity class NPSPACE is

Definition 1.2.5. A language is PSPACE-complete if

Definition 1.2.6. A language is NPSPACE-complete if

1.3 Polynomials

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 1.3.1 ([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 1.3.1.1. 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 1.3.2 ([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 1.3.3 ([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 1.3.4 ([4, §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 1.3.5 ([4, §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*

$$\hat{f}(x) = \sum_{\beta \in H^m} \delta_\beta(x) f(\beta), \tag{1.3.1}$$

where δ is the polynomial

$$\delta_x(y) = \prod_{i=1}^{m} \left(\sum_{\omega \in H} \left(\prod_{\gamma \in H \setminus \{\omega\}} \frac{(x_i - \gamma)(y_i - \gamma)}{(\omega - \gamma)^2} \right) \right). \tag{1.3.2}$$

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 $x, y \in H^m$,

$$\delta_y(x) = \begin{cases} 1 & x = y \\ 0 & x \neq y. \end{cases}$$

This can be shown through some straightforward but tedious algebra which we have omitted here. The equivalence above is the reason we have chosen our notation here to be reminiscent of the Kronecker delta function.

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 1.3.6 ([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*

$$\hat{f}(x) = \sum_{\beta \in \{0,1\}^m} \delta_\beta(x) f(\beta) \tag{1.3.3}$$

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

$$\delta_x(y) = \left(\prod_{i : x_i = 1} y_i \right) \left(\prod_{i : x_i = 0} (1 - y_i) \right). \tag{1.3.4}$$

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

As we can see, the form of δ in Equation (1.3.4) is much more manageable than the form in Equation (1.3.2), and it is perhaps more immediately apparent here why δ has the property it does.

The form of δ_x defined in Equation (1.3.4) has further use to us than just being simpler.

Theorem 1.3.7 ([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 $\mathbb{F}^n \to \mathbb{F}$.*

This is a particularly useful basis because it allows us to reason about multilinear polynomials in terms of their outcomes on the Boolean cube.

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

$$A = \bigcup_{m \in \mathbb{N}} \{x \in \{0,1\}^m \mid A_m(x) = 1\}. \tag{2.0.1}$$

An oracle is not particularly interesting mathematical object on its own; its utility comes from when it interacts with a Turing machine.

Definition 2.0.2. A *Turing machine with an oracle* is

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. □

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 algebrizing, 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 algebrize*.

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 algebrize*.

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: if we have a machine that can do more operations without an oracle, it can still do so 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 $\mathsf{P}^A \subsetneq \mathsf{NP}^A$.

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. 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 $f(A)$.

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 2.2.2 ([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 2.2.3 ([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.

Theorem 2.3.1 ([3, 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 [5] is that $\mathsf{PSPACE} = \mathsf{NPSPACE}$. Combining all these results, we get the chain

$$\mathsf{NP}^A \subseteq \mathsf{NPSPACE} = \mathsf{PSPACE} \subseteq \mathsf{P}^A \subseteq \mathsf{NP}^A. \tag{2.3.1}$$

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.

Definition 2.3.2 ([3, p. 436]). Let X be an oracle. The language $L(X)$ is the set

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

Lemma 2.3.3 ([3, 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. □

Lemma 2.3.4 ([3, Lemma 2]).

Theorem 2.3.5 ([3, 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 wish to construct B iteratively. We shall do this by taking a sequence of machines in P^A, and then constructing B such that no machine could possibly recognize $L(B)$. 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 $B(0) \leftarrow \emptyset$;
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 1: 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 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.

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 conceit 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. This ensures that no already-seen machine P_i queries any strings we might add in the next iteration, which preserves the soundness of our algorithm. Hence, 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, $\mathsf{P}^B \neq \mathsf{NP}^B$. □

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.

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 $m, \mathbb{F}$.

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

$$\tilde{D}_{\mathbb{F},c}(f) = \min_{M \in \mathcal{M}} \left(\max_{A, \tilde{A}:\mathrm{mdeg}(\tilde{A}) \leq c} T_M(\tilde{A}) \right), \tag{3.1.1}$$

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.

3.2 Algebrization of P vs. NP

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

Lemma 3.2.1. *The multilinear extension of any* PSPACE*-complete language is also in* PSPACE.

Proof. □

This result is incredibly important in Theorem 3.2.2: this tells us that PSPACE-complete languages are not made any more powerful when extended to finite fields. This allows to reuse the same analysis we made earlier in Theorem 2.3.1 mostly as-is.

Theorem 3.2.2 ([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, Babai, Fortnow, and Lund [2] have observed that the multilinear extension of any PSPACE language is also in PSPACE. Hence, reusing our argument from Theorem 2.3.1, we have

$$\mathsf{NP}^{\tilde{A}} = \mathsf{NP}^{\mathsf{PSPACE}} = \mathsf{PSPACE} = \mathsf{P}^A. \tag{3.2.1}$$

□

Theorem 3.2.3 ([1, Theorem 5.3])**.** *There exist A, $\tilde{A}$ such that* $\mathsf{NP}^A \neq \mathsf{P}^{\tilde{A}}$.

Proof. □

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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

Index