Claremont Colleges
Scholarship @ Claremont

HMC Senior Theses

HMC Student Scholarship

2021

The Complexity of Symmetry

Matthew LeMay

Recommended Citation
LeMay, Matthew, "The Complexity of Symmetry" (2021). *HMC Senior Theses*. 246.
https://scholarship.claremont.edu/hmc_theses/246

The Complexity of Symmetry,
or,
Bad News for All of You Planning on Computing Invariant Polynomials with Small-Sized Arithmetic Circuits Later Today

Matthew LeMay

Mohamed Omar, Advisor

Nicholas Pippenger, Reader

HARVEY MUDD COLLEGE

Department of Mathematics

May, 2021

Abstract

One of the main goals of theoretical computer science is to prove limits on how efficiently certain Boolean functions can be computed. The study of the algebraic complexity of polynomials provides an indirect approach to exploring these questions, which may prove fruitful since much is known about polynomials already from the field of algebra. This paper explores current research in establishing lower bounds on invariant rings and polynomial families. It explains the construction by Garg et al. (2019) of an invariant ring for whom a succinct encoding would imply $\mathsf{NP} \subseteq \mathsf{P}$/poly. It then states the partial derivative complexity theorem of Baur and Strassen (1983) and its implications for elementary symmetric function complexity, and proposes potential implications for other classes of functions.

Contents

Abstract **iii**

Acknowledgments **ix**

1 Introduction **1**

1.1 Why polynomials? 2

1.2 Invariant rings 4

1.3 Symmetric polynomials 5

2 Background **7**

2.1 Language complexity 7

2.2 Algebraic complexity 11

2.3 Invariant rings 18

3 Conditional Lower Bounds for Invariant Rings **25**

3.1 NP $\not\subseteq$ P/poly, probably 25

3.2 3-way matching is NP-complete 27

3.3 The group action 27

3.4 If we had a succinct encoding, we could solve 3-matching . . 29

3.5 Conclusion 30

4 Polynomial Lower Bounds **31**

4.1 Elementary symmetric polynomials 32

4.2 Proof overview 33

4.3 Partial derivatives preserve lower bounds 34

4.4 The lower bound 38

5 Can We Go Further? **41**

5.1 Simple extensions 41

5.2 Which polynomials have symmetric partial derivatives? . . . 43
5.3 What next? 48

6 Conclusion 51
6.1 What we have learned 51
6.2 Future work 52

Bibliography 55

List of Figures

1.1 Two ways to express the Boolean function $f(x, y) = x \wedge y \vee \neg y$, and their equivalents for the polynomial $p(x, y) = xy + y + 1$. 2

2.1 An algebraic circuit computing the polynomial $1 + \alpha(x^2 + y^2) + \beta x^2 y^2 + \alpha\beta(x^3 y - xy^3)$. As α and β run over all values in $\mathbb{C}$, this produces a generating set for the invariant ring $\mathbb{C}[x, y]^{\mathbb{Z}/4\mathbb{Z}}$, making it a succinct encoding for that invariant ring. 23

Acknowledgments

Thank you to my advisor, Prof. Omar, for helping me find this topic, and supporting me as I slowly struggled through it. Thank you to my second reader, Nick Pippenger, whose very detailed comments have helped ensure the rigor of my writing. Thanks also to Zoë Bell for her peer review feedback, and to my cat Elise, whose support was absolutely indispensible.

Chapter 1

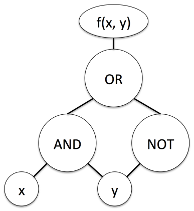
Introduction

Computers are called “computers,” so one might ask, what do they compute? And of course, there are many possible answers, but essentially all of them are encapsulated by one answer: Boolean functions. The semiconductors in a computer allow arbitrary information to be stored in the form of bits (usually conceived as “0s and 1s,” though they could just as well be labelled “cats and hamsters”). Any manipulation of this information is then a process of converting bits into other bits, and any transformation of bits into bits is a Boolean function. There are many questions about computation one might want answered, including questions with important applications to human existence, like “Can a computer simulate the structure and behavior of enzymes efficiently enough to automate the process of curing diseases?”, and all such questions are fundamentally questions about Boolean functions.

Given the central importance of Boolean functions, it may surprise you to learn that this thesis is not actually about Boolean functions at all. It is about polynomials. You are probably now sadly shaking your head and saying, “How typical for a callow young mathematician to waste valuable time mucking about with polynomials, when Boolean functions are sitting right there, practically begging to be studied.” But wait! I can explain; hear me out. Studying polynomials may in fact be more beneficial for understanding Boolean functions than directly studying Boolean functions themselves. The argument for this very counterintuitive statement is essentially two facts: (1) polynomials and Boolean functions are very similar entities, and (2) it is easier to prove facts about polynomials than Boolean functions. Before I drag you with me down into the depths of the polynomial world, I’d like to convince you of these two facts.

x	y	f(x,y)
false	false	true
false	true	false
true	false	true
true	true	true

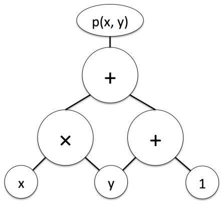
a. A truth table for f.



b. A circuit computing f.

x	y	p(x,y)
0	0	1
0	1	0
1	0	1
1	1	1

c. p as a function over $\mathbb{Z}_2$.



d. An arithmetic circuit computing p.

Figure 1.1 Two ways to express the Boolean function $f(x, y) = x \wedge y \vee \neg y$, and their equivalents for the polynomial $p(x, y) = xy + y + 1$.

1.1 Why polynomials?

First off, what have polynomials to do with Boolean functions? The two are on their surface very different. A Boolean function is something that takes parameters that are either true or false, and returns either true or false, such as $f(x, y) = (x \text{ and } y)$ or (not y), which using standard logical symbols we would represent as $f(x, y) = x \wedge y \vee \neg y$. Boolean functions are fully defined by their outputs, which we can represent by a table of values, as in Figure 1.1a. Two functions with the same table of values are considered the same function. For example, the Boolean function $f'(x, y) = \neg(((\neg x) \vee (\neg y)) \wedge y)$ is precisely the same function as f because it shares the same table of values, even though we have defined it with a different formula.

Contrast this with polynomials, such as $p(x, y) = xy + y + 1$. p is not defined by its outputs, but rather by the coefficients on its terms. Considered over $\mathbb{Z}_2$ (the integers mod 2), p specifies a function from $\mathbb{Z}_2 \times \mathbb{Z}_2$ to $\mathbb{Z}_2$, as shown in Figure 1.1c. But even though $p'(x, y) = x^2y + y^2 + 1$ specifies the same function over $\mathbb{Z}_2$ (since $z^2 = z$ for all $z \in \mathbb{Z}_2$), p' and p are still distinct polynomials. This is because polynomials are defined specifically by their coefficients on each monomial: p has a coefficient of 1 on the monomials $xy = x^1y^1$, $y = x^0y^1$, and $1 = x^0y^0$, and a coefficient of 0 on every other monomial. p' is a distinct polynomial from p since its coefficient on xy

is 0 instead of 1, and its coefficient on x^2y is 1 instead of 0. However, $p''(x, y) = 1 + y(1 + x)$ is the same polynomial as p, since expanding out the expression for p'' using the distributive property gives $p''(x, y) = 1 + y + xy$, which has precisely the same coefficient set as p. So the important takeaway here is: Polynomials are coefficient sets and not functions, although they do define functions.

Comparing Figures 1.1a and 1.1c, you may notice that a relabelling of true to 1 and false to 0 renders the two tables identical. Stated another way, as long as true is called 1 and false is called 0, the function specified by the polynomial p over $\mathbb{Z}_2$ is precisely the Boolean function f. And this is the key connection between Boolean functions and polynomials (as well as the reason that the "cats/hamsters" labelling for bits did not catch on) – every Boolean function is specified by a polynomial over $\mathbb{Z}_2$. This is a consequence of the fact that all Boolean functions are computed by Boolean circuits consisting of AND, OR, and NOT gates, such as the one for f in Figure 1.1b. Since all of three of these gates have corresponding polynomials over $\mathbb{Z}_2$ ($\mathrm{NOT}(x) = 1+x$, $\mathrm{AND}(x, y) = xy$, $\mathrm{OR}(x, y) = x+y+xy$), all circuits built from them can be converted into *arithmetic circuits* with addition and multiplication gates that compute polynomials, such as the one computing p in Figure 1.1d. So every Boolean function is really "a polynomial at heart."

This then leads us to the second fact, that polynomials are easier to study than Boolean functions. More precisely, it is at least as difficult to prove that a Boolean function is hard to compute, as to prove that a polynomial is hard to compute. To see why, suppose a complexity theorist spends a long time proving that you can't compute a certain kind of Boolean function with a circuit smaller than a certain size – a *circuit lower bound* on those Boolean functions. Then they have also proven an arithmetic circuit lower bound on the polynomials corresponding to those Boolean functions, because if one could compute the polynomials efficiently, one could easily compute the Boolean functions by just plugging values into the polynomials. So studying Boolean functions is like lifting double weight – any lower bound results have to hold for both Boolean functions and polynomials.

On the other hand, the reverse does not hold. That is, if you can prove arithmetic circuit lower bounds on a family of polynomials, then that doesn't necessarily imply Boolean circuit lower bounds, because there is no way to translate Boolean functions back into polynomials. For example, the polynomial $p'(x, y) = x^2y + y^2 + 1$ corresponds to the Boolean function $f(x, y) = x \wedge y \vee \neg y$. But if you could compute f extremely efficiently, then this wouldn't help you compute p' since there is no way to translate f back into p'. There are many other polynomials that correspond to f, such as $p(x, y) = xy + y + 1$. When we consider a polynomial as a function, we lose information about which coefficients were associated with which monomials, and that information can't always be recovered from the function alone. This means that it might be very hard to compute a polynomial even if its Boolean function is easy to compute. So in studying polynomials, it may be possible to prove lower bounds that one couldn't have found equivalents of in the Boolean realm.

Now, you may be thinking, what is then the point of studying polynomials if polynomial lower bounds don't directly imply Boolean function lower bounds? Aren't Boolean functions what we really care about? And this is a reasonable question. But the direct study of Boolean function lower bounds is extremely difficult and very few concrete results have been found after decades of study. Polynomial results may not directly imply Boolean results, but they may be a stepping stone on the way, and they may inspire new strategies of analysis. Besides which, there is an entire field of mathematics called "algebra" which predates computer science by centuries, and is full to the brim of results about polynomials. This provides a wealth of knowledge about them which may make it easier to understand their inherent complexity.

This thesis is concerned with using tools available from algebra to understand the inherent complexity of polynomials. We are looking for provable limits on how efficiently we can compute algebraic objects such as families of polynomials. We will focus on two types of algebraic objects: invariant rings and symmetric polynomials. We will first discuss conditional results that show that there exist invariant rings which cannot be efficiently represented, assuming commonly believed conjectures in complexity theory hold. We will then discuss unconditional results that show explicit lower bounds on certain families of symmetric polynomials.

1.2 Invariant rings

We will define invariant rings in more detail in Chapter 2, but informally, they are sets of polynomials which are unchanged by a particular collection of transformations, called a group action. For example, one transformation in a group action might be to transform polynomials in the variables x, y, z by replacing x with y, y with $2z$, and z with $\frac{1}{2}x$. This would turn a polynomial into another polynomial, but certain polynomials, like $f(x, y, z) = xyz$, would be unchanged. The set of all polynomials unchanged by a group action is called an invariant ring. So you can think of invariant rings as being polynomials that are defined by some kind of inherent symmetry.

An invariant ring is called a ring because this is the algebraic name for a set of objects in which the objects can be added or multiplied and the result will still be in the set (along with some other rules that enforce "expected behavior" of the operations). Invariant rings usually contain infinitely many polynomials, which may put some doubt in your mind that it makes sense to talk about them as computable objects. And, indeed, a computer cannot possibly hold infinitely many polynomials in memory. However, it was proved by David Hilbert that for a large class of "well-behaved" group actions, their invariant rings will have a finite set of generators (Hilbert (1893)). This means that one could write down a finite list of polynomials such that any polynomial in the invariant ring could be found by adding and multiplying the polynomials on the list in some way. This provides some hope that, even though the ring may be infinite, one could state precisely what the invariant ring is, without requiring infinite space.

To quantify the complexity of an invariant ring, we want to know how efficiently the list of generators can be expressed. This is often formalized as the size of the optimal *encoding*, where an encoding is an arithmetic circuit whose set of possible outputs forms a generating set for the invariant ring. If we want to compute a given invariant ring, we might hope for a small encoding. However, work by Garg et al. (2019) gives an explicit invariant ring for whom a brief (polynomial-size) encoding would imply an unlikely result in complexity theory ($\mathsf{NP} \subseteq \mathsf{P}/\text{poly}$). This then is an invariant ring which likely has a high level of "inherent complexity," since it probably cannot be efficiently expressed. We show the construction of this invariant ring and prove its implications in Chapter 3.

1.3 Symmetric polynomials

The *symmetric group* on n elements, S_n, is the algebraic structure containing all possible permutations (or "reorderings") of those elements. S_n has a natural group action on n-variate polynomials, in which a permutation in S_n permutes the variables of the polynomial. A symmetric polynomial is any polynomial in the invariant ring of this action. Some natural symmetric polynomials we might think of would include the sum of all the variables and the product of all the variables. In fact, these are the 1st and n-th *elementary symmetric polynomials*, where the d-th symmetric polynomial is the sum of all possible products of d distinct variables. It can be shown that the elementary symmetric polynomials are a generating set for the full ring of symmetric polynomials.

Elementary symmetric polynomials have a special place in the current landscape of algebraic complexity theory, since they are some of the only polynomials for which nontrivial lower bounds have been shown. It was shown by Baur and Strassen (1983) that for $d \leq \frac{n}{2}$, the d-th n-variate elementary symmetric polynomial requires an arithmetic circuit size of at least $\Omega(n \log d)$. We will discuss precisely what this means and how it is shown in Chapter 4.

The proof is based on the interesting result that the circuit complexity of the partial derivatives of a function is directly related to its own circuit complexity. It is possible that this strategy may generalize, and that other polynomial lower bounds can be obtained by analyzing the partial derivatives of function. We explore this possibility in Chapter 5.

This thesis is intended as an introduction to the study of algebraic lower bounds, to make the big questions and recent results in this area accessible to a more general mathematical audience. I hope that it will help you, the reader, understand and appreciate this field, and perhaps even inspire your own inquiries. Let's get into it!

Chapter 2

Background

The goal of this chapter is to formally define concepts like "arithmetic circuits," "invariant rings," and "succinct encodings." To give a clear overall picture of the field of algebraic complexity theory, we will also discuss one of its biggest open questions – the question of whether VP is equal to VNP. This is the algebraic analog of the more famous P vs. NP question in "standard" or language-based complexity theory, so to motivate our discussion, we will begin with some background in language complexity. We will then see how algebraic complexity is defined in analogy with language complexity, and finally introduce the concept of invariant rings. The rest of the paper will not directly involve the VP vs. VNP question, so readers who are for some reason not interested in the context of the overall field of algebraic complexity may wish to skim lightly over section 2.2.

2.1 Language complexity

First, a brief summary of "standard" (non-algebraic) complexity theory. This concerns the complexity of *languages*, which are sets of finite-length binary strings. For example, one such language would be the set of all possible binary strings, and another would be the empty set. For a slightly more interesting example, consider the language L_{1alt} of all strings consisting of a 1 followed by 0 or more copies of the string 01. (Using the terminology of regular expressions, which we won't define here, this is the language defined by 1(01)*). The strings in L_{1alt} are 1, 101, 10101, 1010101, and so forth.

For a substantially more interesting language, assume a binary encoding system that allows us to (a) uniquely describe a graph G as a binary string

$\langle G \rangle$, (b) represent any integer in the set $\mathbb{Z}$ of all integers, and (c) encode a divider symbol ",". Recall that graph G is said to be k-colorable if each of its vertices can be assigned one of k colors in such a way that no two vertices of the same color are connected by an edge. Then we can define a language:

$$L_{kcolor} = \left\{ \langle G \rangle, k \mid \text{ G is a graph, } k \in \mathbb{Z} \text{ and } k > 0\text{, and G is } k\text{-colorable } \right\}.$$

We say that a language is *computable* if given a binary string, a Turing machine can determine whether the string is in the language. A Turing machine is an abstraction of computers with a finite number of states, an unlimited memory "tape," and a "head" which observes one tape cell at a time. At each step of computation, it acts based on which state it is in and what character it is reading from the memory tape. It can act by changing its state, writing a character on the memory tape, and/or moving its head left or right on the tape. It finishes a computation by entering the "accepting" state or the "rejecting" state. A string s is in the language computing by a Turing machine M if and only if, when M's memory tape begins with s written on it, M's computation ends in the accepting state. We say that M *computes* the set of strings it accepts. This is based on the process a human being might go through when solving a problem with pencil and paper, and in general, we usually expect this model of computation to be capable of computing anything a human being could in principle figure out on paper (given sufficient quantities of time and paper).

L_{1alt} can be computed by a Turing machine that simply moves through the input string once, checking that the string begins with a 1 and the characters alternate between 0 and 1. Thus L_{1alt} is computable. L_{kcolor} is also computable, because there exists a Turing machine that can determine whether any string is in L_{kcolor} in finite time. At a high level, this machine first verifies the string is well-formatted (that it consists of a valid graph G and a positive integer k), then lists all possible k-colorings of the vertices of G, and checks if each is a valid k-coloring.

This fairly simple algorithm is sufficient to show that we can check whether strings are in L_{kcolor} in finite time. However, for a graph with N vertices, there are k^N possible k-colorings of the vertices to check, an exponential number of colorings. For graphs with on the order of thousands of vertices, a computer might spend the better part of a human lifetime running this algorithm. It is therefore safe to say that even though this algorithm will always solve the problem eventually, most people interested in the results of the computation would be dissatisfied with it. [1]

[1] Perhaps this is an unfortunate consequence of the rapid pace of modern life, and the

Complexity theory provides us with a way to distinguish between the different kinds of "finite time" (a few seconds versus a few millenia), by stratifying computable languages by how efficiently they can be computed. A common measure of "efficient" is "polynomial time": A language L is computable in polynomial time if there exists some Turing machine m that computes L, such that for every input of length n, M uses time no more than $p(n)$, where p is a polynomial. The class of all languages that are polynomial-time computable is called P.

It is unknown whether L_{kcolor} is in P. L_{kcolor} is however known to be in a class called NP. NP consists of all problems solvable in polynomial time by a what is called a "non-deterministic Turing machine." This is a Turing machine which is allowed to make guesses on the way to its result. It accepts an input if there is any sequence of guesses that leads it to accept, and it rejects an input if every sequence of guesses leads it to reject. It runs in polynomial time if its running time is polynomially bounded for every possible sequence of guesses.

To get an intuitive understanding of NP, imagine the following scenario: You are a student in a middle school class. The teacher has written a particular statement on the blackboard, and has asked that, after precisely one minute for consideration, everyone who believes the statement to be true should raise their hand at the same time. Whether you raise your hand will then count as your answer, and you will be graded on correctness. Unfortunately, due to the wealth of non-academic concerns that characterize your life as a middle school student, you don't know how to figure out the answer. You want to get a good grade, but above all you want to avoid the embarassment of raising your hand when the statement is false, as this would call the attention of the class directly to you and your mistake. In these circumstances one might reasonably resign oneself to just not raising one's hand no matter what, but you have a glimmer of hope for getting the right answer, in the form of your intelligent friend, Lyra. Lyra is sitting right next to you, close enough that they can whisper to you without anyone noticing. But you know quite well that Lyra can sometimes have a cruel

crisis of the ever-shortening attention span. In a less hasty and impatient culture, we might be contented to simply wait this time out. The Ents of Fangorn Forest from J.R.R. Tolkien's *The Two Towers* are an extremely patient people, who don't mind taking the time necessary to really appreciate moments as they pass by. Even in their language of Old Entish, shares an Ent named Treebeard, one does not say anything "unless it is worth taking a long time to say, and to listen to" (Tolkien, 1954: p. 66). At no point in *The Two Towers* do the Ents ever mention developing computational complexity theory, and this can hardly be a coincidence.

sense of humor, so if they simply whisper "it's true," this won't be enough to convince you to risk raising your hand. Instead, Lyra will whisper to you a hint with which you will try to convince yourself that the statement is true. Note that both the hint and the checking process must be time-efficient, since you only have a minute to answer. After the minute is up, you will raise your hand only if their hint has fully convinced you that the statement is true.

If you replace "one minute" with "polynomial time in the length of the input," then problems in NP are precisely those languages L such that the statement on the board is in L if and only if there is some hint that Lyra can give you that will convince you to write "true."

To use the example of L_{kcolor}, imagine that the teacher has drawn a graph G on the blackboard, and next to it written the statement "G is 5-colorable." Then if the graph really is 5-colorable, Lyra could whisper a valid 5-coloring to you (as in "Vertex 1 red, vertex 2 blue, vertex 3 red, ..."), and you could use your minute to verify that no edge connects same-colored vertices. If the coloring checks out, you know for sure that G is 5-colorable, and can then feel safe and secure in raising your hand. If the graph is not 5-colorable, then no matter what 5-coloring Lyra whispers to you, you will notice a particular edge connecting same-colored vertices somewhere in the graph, and you won't risk raising your hand. Note that if you fail to verify the statement based on the hint, you can't be sure whether the statement is really false, or if the statement is true and it was just a bad hint. But if you do verify the statement, you can be absolutely sure that the statement is true.

What does this scenario have to do with Turing machines that can guess? Well, imagine that the teacher has discovered your illicit answer-swapping with Lyra, and has moved you to opposite sides of the classroom, so you can no longer whisper to each other. Are you now forced to give up on the prospect of getting the answer right if the statement happens to be true? Not quite! You can simulate having your friend Lyra next to you by simply taking a guess as to a hint they might give. The odds may not be very good that you will hit upon a helpful hint by chance, but nonetheless, if the statement is true, there will exist some guess you can make that will match up with what Lyra would have told you. This is the essence of what it means for a non-deterministic Turing machine to guess.

Every problem in P is also in NP because the non-deterministic Turing machine could simply not make use of its guessing ability (i.e. the problem is so easy you don't even need Lyra's help). It is unknown, however, whether P is strictly contained in NP, or whether the two classes are equal. It is generally suspected among complexity theorists that $\mathsf{P} \neq \mathsf{NP}$, but proving this remains an open problem.

It turns out that $L_{kcolor} \in \mathsf{P}$ if and only if $\mathsf{P} = \mathsf{NP}$. This is because of the somewhat magical fact that given any problem in NP, there is a polynomial-time procedure for transforming instances of that problem into instances of L_{kcolor}. If it were possible to solve L_{kcolor} in polynomial time, then we could solve any problem in NP in polynomial time by converting it into an instance of L_{kcolor} (that is, a graph G and integer k) with the same yes/no answer as the original problem, and then find the original answer with our L_{kcolor} algorithm. In the language of complexity theory, we say that any language in NP is *polynomial-time reducible* to L_{kcolor}. We call a language in NP to which all other languages are polynomial-time reducible NP-complete under polynomial-time reductions. NP-complete languages can be thought of as "the hardest languages in NP."[2]

We can discuss completeness in P also, but polynomial-time reductions are less useful in this context, since any problem $A \in \mathsf{P}$ can be trivially polynomial-time reduced to another problem B by simply solving problem A in polynomial time and then supplying a B instance that gives the same answer. It is more helpful in this context to discuss *log-space reductions*, in which the reduction must be doable by a Turing machine that can read the input, but otherwise only has access to space of size logarithmic in the size of the input. For example, the following problem, the Circuit Value Problem, is P-complete under log-space reductions (Ladner (1975)): Given a description of a Boolean circuit and its inputs, does it output True?

We introduce these ideas of P, NP, reduction, and completeness for languages because they provide the framework for analyzing the complexity of other structures – in particular, polynomials.

2.2 Algebraic complexity

In this section we will show how the ideas of complexity theory introduced in the previous section have analogs in the study of polynomial complexity. The majority of this section is based on an excellent survey paper by Mahajan (2014).

[2] Note that we have specifically defined L_{kcolor} to take an arbitary k as input. The 2-coloring problem, the special case in which k is always 2, is not believed to be NP-complete, as it is known to be in P. But in fact the 3-coloring problem is NP-complete, so as long as the $k \geq 3$ case is allowed, k-coloring is NP-complete.

Let (f_n) be a polynomial family, that is, a sequence of polynomials $f_1, f_2, f_3, \ldots$, where each f_n is a degree $d(n)$ polynomial in $s(n)$ variables. We will call (f_n) *tractable* if $d(n) \leq \text{poly}(n)$ and $s(n) \leq \text{poly}(n)$. The notion of tractability encodes the fact that we want the complexity of our polynomials to increase at a reasonable rate. If our polynomial family were defined by $f_n(x) = x^{2^{2^n}}$, so that the degree of f_n grows doubly exponentially in n, then it wouldn't be particularly surprising to find that f_n requires a lot of computation relative to size of the index n. To avoid situations like this, we will henceforth assume our polynomial families to be tractable.

One way we could measure the complexity of (f_n) is by asking for the lengths of formulas that compute f_n. What is a formula? Given a field k (think of the complex numbers, $k = \mathbb{C}$) and indeterminates $x_1, \ldots, x_n$, we can define a formula recursively as follows:

1. If $c \in k$, then "c" is a formula computing the constant polynomial c, whose size is the number of bits needed to encode c.

2. If x_i is an indeterminate, then "x_i" is a formula computing the polynomial x_i, of size 0.

3. If F_1 and F_2 are formulas computing polynomials f_1 and f_2, then $F_1 + F_2$ and $F_1 \times F_2$ are formulas, each of size $1 + \text{size}(F_1) + \text{size}(F_2)$, computing respectively $f_1 + f_2$ and $f_1 f_2$.

An example of a formula would be something like

$$\text{“}(x_1 + 2) \times ((2 \times x_2) + (-1))\text{”},$$

which computes the polynomial

$$(x_1 + 2)(2x_2 - 1) = 2x_1x_2 - x_1 + 4x_2 - 2.$$

Notice that we can think of a formula as a binary tree, with cases (1) and (2) above representing leaves, and case (3) representing a tree with an operation at its root, and F_1 and F_2 being its left and right subtrees.

We can define the following complexity class of polynomials based on formulas:

Definition 1 (VF)**.** VF is the class consisting of all tractable polynomial families (f_n) that can be computed by a sequence of formulas (F_n) with $\text{size}(F_n) \leq \text{poly}(n)$.

Notice that formulas are a slightly artificial way of representing the computation of polynomials: to compute the result of the formula

$$\text{“}(((x_1 + x_2) \times x_1) + 3) \times (((x_1 + x_2) \times x_1) + 1)\text{”},$$

we are obligated to compute the subformula $(x_1 + x_2) \times x_1$ twice, whereas if we were human beings doing the computation, we could recognize that the two were the same and reuse the same result. Formulas do not allow for the result of a subformula to be used in multiple places in an expression. However, formulas are a special case of another model for computing polynomials where this is allowed:

Definition 2 (Algebraic circuit)**.** An *algebraic circuit* (equivalently, *arithmetic circuit*) over a field k is a directed acyclic graph, which we interpret in analogy with Boolean circuits. Its leaves are either labelled as *inputs* $x_1, \dots, x_n$ which take on values from k, or labelled as constant values from k. Its internal nodes are called *gates* and labelled with one of the operations $\{+, \times\}$. We restrict all gates to have at most two inputs, and we require that there is only one root or *output* gate. Each gate can be considered as a polynomial function of its children, and the circuit is said to *compute* the polynomial function corresponding to its output gate. (Arora and Barak (2009))

We will frequently refer to the *size* of an algebraic circuit, by this we simply mean the number of gates in the circuit, plus the number of bits required to specify the constant value leaves (often excluding the constants 0 and 1). We write $\text{size}(C)$ to mean the size of a circuit C, and if f is a polynomial, we write $S(f)$ to mean the size of the smallest algebraic circuit computing f.

A formula is simply an algebraic circuit whose underlying directed acyclic graph is a tree. Since formulas are a special case of algebraic circuits, VF is contained in another complexity class based on algebraic circuits:

Definition 3 (VP)**.** VP is the class consisting of all tractable polynomial families (f_n) that can be computed by a sequence of algebraic circuits (C_n) with $\text{size}(C_n) \le \text{poly}(n)$.

The V in VF and VP stands for Leslie Valiant, who first defined these classes. VF stands for “Valiant’s Formulas,” while VP is so named because it is “Valiant’s version of P”, the algebraic analog of the class of polynomial-time computable languages.

A good example of a polynomial family in VP is the family of determinant polynomials, (Det_n), where Det_n is the following polynomial in n^2 indeterminates:

$$\mathrm{Det}_n(x_{11}, x_{12}, \ldots, x_{nn}) = \begin{vmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nn} \end{vmatrix} = \sum_{\sigma \in S_n} \left(\mathrm{sgn}(\sigma) \cdot \prod_{i=1}^{n} x_{i\sigma(i)} \right).$$

When we write $\mathrm{sgn}(\sigma)$, we mean 1 is σ is composed of an even number of transpositions, and -1 if it is composed of an odd number of transpositions. (We can define this quantity because every permutation can be decomposed into a unique number of two-element transpositions.)

(Det_n) is known to be in VP. This result is particularly interesting since circuits in VP cannot use division or conditional statements (i.e. something equivalent to "if X then Y else Z"). A circuit with division and conditionals could compute the determinant in polynomial size using something similar to the Gaussian elimination strategy (much as a standard computer program would). It has been shown that one can eliminate the need for division and conditionals: Csanky (1975) shows that (Det_n) can be computed using polynomial-size algebraic circuits with division gates, and Berkowitz (1984) shows it can be computed in polynomial size with only addition and multiplication gates. Thus $(\mathrm{Det}_n) \in \mathsf{VP}$. It is unknown whether (Det_n) is in VF.

Since we have an algebraic analog for P, it is natural to ask whether there is an algebraic analog for NP. Such an analog is defined as follows:

Definition 4 (VNP)**.** VNP is the class consisting of all tractable polynomial families (f_n) such that there is some $(g_n) \in \mathsf{VP}$ such that:

(i) If f_n has $s(n)$ variables $x_1, \ldots, x_{s(n)}$, then g_n has $s(n) + m(n)$ variables $x_1, \ldots, x_{s(n)}, y_1, \ldots, y_{m(n)}$.

(ii) For all $n \in \mathbb{N}$,

$$f_n(x_1, \ldots, x_{s(n)}) = \sum_{(y_1,\ldots,y_{m(n)}) \in \{0,1\}^{m(n)}} g_n(x_1, \ldots, x_{s(n)}, y_1, \ldots, y_{m(n)}).$$

It may not be immediately obvious that the above definition gives an analog for non-determinism. However, we can frame non-deterministic computation in Turing machines as the following process:

1. A sequence of $m(n) \leq \text{poly}(n)$ coin flips is made, producing a guess string $y \in \{0,1\}^{m(n)}$.
2. For each of the $2^{m(n)}$ possible values of y, a computational path is run, leading to an answer of True or False.
3. The logical OR of all computational paths is taken.

The process for computing f_n in Definition 4 is very similar to this process, with $(y_1, ..., y_{m(n)})$ playing the role of the guess string, evaluating g_n for each guess string playing the role of the separate computation paths, and the sum of the results playing the role of the logical OR.

Just as (Det_n) can be thought of as a "canonical" polynomial family for VP, so can (Perm_n), the permanent family, be thought of as the canonical family for VNP. The polynomial Perm_n is the permanent of the $n \times n$ symbolic matrix:

$$\text{Perm}_n(x_{11}, x_{12}, ..., x_{nn}) = \text{Perm}\left(\begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nn} \end{bmatrix}\right) = \sum_{\sigma \in S_n} \left(\prod_{i=1}^{n} x_{i\sigma(i)}\right),$$

or simply the determinant but without the signs of the permutations.

Let us examine in detail why Perm_n is in VNP. Notice that we can encode permutations on n elements as $n \times n$ matrices; for example, consider the permutation on $\{1,2,3,4,5\}$ given by $\sigma = (13)(254)$. This notation means that σ exchanges 3 and 1, and cycles 2 to 5, 5 to 4, and 4 to 2. That is, σ acts as follows:

$$\begin{matrix} 1 & 2 & 3 & 4 & 5 \\ \downarrow_\sigma & \downarrow_\sigma & \downarrow_\sigma & \downarrow_\sigma & \downarrow_\sigma \\ 3 & 5 & 1 & 2 & 4 \end{matrix}$$

We can encode σ as the following matrix Y^σ:

$$Y^\sigma = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix},$$

which we have defined by letting $Y^\sigma_{i\sigma(i)} = 1$ for all $i \in \{1,2,3,4,5\}$, and letting all other entries equal 0. All permutations on n items can be expressed by the same method as $n \times n$ matrices with exactly one 1 is every row and column, and 0's everywhere else.

We can define the monomials of Perm_n based on this matrix representation. The monomial in Perm_5 corresponding to σ defined above is

$$\prod_{i=1}^{5} x_{i\sigma(i)} = x_{13}x_{25}x_{31}x_{42}x_{54}.$$

An algebraic circuit cannot find this monomial by "plugging i into σ and then choosing the entry $x_{i\sigma(i)}$" because it cannot perform this kind of conditional behavior. However, given the matrix Y^σ, it could find $x_{i\sigma(i)}$ by the following expression:

$$\sum_{j=1}^{5} Y_{ij}^{\sigma} x_{ij}.$$

Since the only entry of row i in Y^σ that is 1 is the one in column $\sigma(i)$, this expression will yield simply $x_{i\sigma(i)}$. In this way, we could express Perm_n as follows:

$$\mathrm{Perm}_n(x_{11}, x_{12}, \ldots, x_{nn}) = \sum_{Y \text{ is an } n \times n \text{ permutation matrix}} \left[\prod_{i=1}^{n} \left(\sum_{j=1}^{n} Y_{ij} x_{ij} \right) \right].$$

This is beginning to look like an expression for a polynomial family in VNP, since it involves a sum of all possible polynomials produced by "guessing" a permutation matrix Y. However, we cannot sum over specifically the set of permutation matrices Y; the definition of VNP only allows us to specify a sequence length and sum over *all* binary strings of that length. What we can do instead is sum over all $n \times n$ matrices composed of 0's and 1's, and multiply the terms by 0 if the matrix is not a permutation matrix. To do this, we need a polynomial $h_n(Y)$ that serves as an indicator function for whether Y is a permutation matrix. This can be defined as follows:

$$h_n(Y) = \left(\prod_{i=1}^{n} \sum_{j=1}^{n} Y_{ij} \right) \cdot \left(\prod_{i=1}^{n} \prod_{\substack{j_1, j_2 \in [n] \\ j_1 < j_2}} \left(1 - Y_{ij_1} Y_{ij_2}\right) \right) \cdot \left(\prod_{j=1}^{n} \prod_{\substack{i_1, i_2 \in [n] \\ i_1 < i_2}} \left(1 - Y_{i_1 j} Y_{i_2 j}\right) \right).$$

The above is a product of three factors, each being either 1 or 0 (assuming the entries of Y are all 1 or 0). The first factor is nonzero if and only if Y has at least one 1 in each row, the second factor is 1 if and only if Y has has at most one 1 in each row, and the third factor is 1 if and only if Y has at most one 1 in each column. We do not need a factor that is 1 if and only if Y has at least one 1 in each column, since it would be redundant – if some column is only zeroes, then either there is some row with only zeroes, or there is a column with more than one 1. Among the three factors that make up $h_n(Y)$, the first can be computed within n^2 gates (since there are n factors to be multiplied, each of which is a sum of n constants). Similarly, the second and third factors can each be computed within a constant multiple of n^3 gates, since they involve three index variables ranging from 1 to n. Thus $h_n(Y)$ can be computed by an algebraic circuit with no more than a constant multiple of n^3 gates, which is polynomial in n.

Armed with $h_n(Y)$, we can now express Perm_n as:

$$\mathrm{Perm}_n(x_{11}, x_{12}, \dots, x_{nn}) = \sum_{Y \in \{0,1\}^{n^2}} \left[h_n(Y) \cdot \prod_{i=1}^{n} \left(\sum_{j=1}^{n} Y_{ij} x_{ij} \right) \right].$$

This is now a form consistent with the definition of VNP! To see this, let g_n be a polynomial in $2n^2$ variables defined by

$$g_n(x_{11}, x_{12}, \dots, x_{nn}, Y) = h_n(Y) \cdot \prod_{i=1}^{n} \left(\sum_{j=1}^{n} Y_{ij} x_{ij} \right).$$

Note that g_n is a tractable polynomial family and can be computed with polynomial-size circuits in n, so $(g_n) \in \mathsf{VP}$. Then Perm_n can be expressed as

$$\mathrm{Perm}_n(x_{11}, x_{12}, \dots, x_{nn}) = \sum_{Y \in \{0,1\}^{n^2}} g_n(x_{11}, \dots, x_{nn}, Y),$$

so $\mathrm{Perm}_n \in \mathsf{VNP}$.

Thus far we have defined VP and VNP, and shown that $\mathrm{Det}_n \in \mathsf{VP}$ and $\mathrm{Perm}_n \in \mathsf{VNP}$. It is clear that $\mathsf{VP} \subseteq \mathsf{VNP}$, since the definition of VP is just the definition of VNP in the special case where $m(n) = 0$. Just like their analogs P and NP, however, it is an open question whether $\mathsf{VP} = \mathsf{VNP}$. The determinant and permanent are often thought of as key to answering this question. This is because of an analog of reductions for polynomial families called projections. We will not give a formal definition, but the essence of a projection is that a polynomial family (f_n) is at least as hard to compute as a family (g_n) if for all n, g_n can be expressed by plugging some combination of variables and constants into f_m for some m. If m is polynomial in n, this is called a p-projection, and if m is quasi-polynomial in n (meaning m is bounded above by $2^{\log^c n}$ for some c), this is called a qp-projection.

Leslie Valiant showed that every polynomial family in VNP is reducible by p-projections to Perm_n, and every polynomial family in VP is reducible by qp-projections to Det_n (Valiant (1979)). So Perm_n is a "canonical complete problem" for VNP is a nice and satisfying way, and Det_n is *almost* that for VP. The VNP-completeness of Perm_n implies that the question of whether $\mathsf{VP} = \mathsf{VNP}$ is equivalent to whether Perm_n is in VP. This provides much of the motivation for the study of arithmetic circuit lower bounds – we want to eventually be able to prove that Perm_n cannot be represented with polynomial-size circuits, which would then show that $\mathsf{VP} \neq \mathsf{VNP}$. This would be an important result for fields in which the permanent plays an important role, such as graph theory, knot theory, and statistical mechanics (Wigderson (2019)), and its proof might also provide insight into showing that $\mathsf{P} \neq \mathsf{NP}$. In the remainder of the paper, we will explore known strategies for lower-bounding arithmetic circuits which may eventually provide insight into Perm_n.

2.3 Invariant rings

Readers who have been reading since the introduction (the long-time fans) will know that the primary concern of this paper isn't just the complexity of any old polynomials, but particularly those which are defined by their invariance under some kind of transformation. These polynomials are the elements of *invariant rings*, and this section will clarify what exactly an invariant ring is.

To begin with, we need to define the transformations under which the polynomials are invariant. These will take the form of group actions. Note that we will not formally define the notions of group, ring, field and vector space; any good algebra textbook can provide these definitions for those interested. For readers who want to know what they are but aren't inclined to go through the effort of looking them up, here is a lightning summary: Groups, rings, and fields are all sets of elements on which operations can be defined that follow specific rules; these rules ensure the operations behave in the "nice" ways one would expect. In a group there is a single operation (usually called addition or multiplication). In a ring there are two operations (both addition and multiplication). A field is a ring in which multiplication is commutative, and one can divide by all elements except 0 (multiplication is invertible). A vector space is a set of elements together with a field called its scalar field (it is said to be a vector space *over* its scalar field), and its operations are (1) addition and (2) multiplication by elements of the scalar field. (For a visualization of vector spaces, imagine arrows in three-dimensional space which can be added tip-to-tail, or stretched by constant factors.)

A homomorphism of a group $(G, \times)$ is a function φ from G to another group $(H, \cdot)$ such that for $g_1, g_2 \in G$, $\varphi(g_1 \times g_2) = \varphi(g_1) \cdot \varphi(g_2)$. One can think of homomorphisms as preserving the structure defined by the operations of the respective groups. A similar definition holds for rings and fields, where homomorphisms preserve the structure of both addition and multiplication. On a vector space V, homomorphisms must preserve the structure of both addition and scalar multiplication, and are also called linear transformations. An isomporhism is an invertible homomorphism, and an automorphism is an isomorphism from an algebraic structure to itself. The automorphisms of a group, ring, etc. form a group under composition, which is denoted by $\circ$.

Definition 5 (Group action)**.** Let G be a group, and V a vector space. An *action* of G on V is an association of each element $g \in G$ with an automorphism $\pi(g) : V \longrightarrow V$, such that for all $g_1, g_2 \in G$, $\pi(g_1 g_2) = \pi(g_1) \circ \pi(g_2)$. That is, π is a homomorphism from G to the group of automorphisms of V. We will write $g \cdot v$ to mean $\pi(g)[v]$.

We will assume for our purposes in this paper that we are dealing with a finite-dimensional vector space V of dimension n over a field k. You can use some linear algebra to show that this vector space can be taken without loss of generality to be k^n, or the set of n-tuples $(x_1, x_2, ..., x_n)$ with every $x_i \in k$. Automorphisms of V are then equivalent to invertible $n \times n$ matrices with coefficients in k. Since we can naturally think of points in V as settings of n variables, it is natural to speak of n-variate polynomials over V. Define $k[V] = k[x_1, ..., x_n]$ as the set of polynomials over the indeterminates $x_1, ..., x_n$, with coefficients in k. If $f \in k[V]$, then f can be naturally interpreted as a function $f : V \longrightarrow k$ by simply substituting the coordinates of a vector v in for $x_1, ..., x_n$. Then if G is a group acting on V, and $g \in G$, we write $g \cdot f$ to mean the polynomial such that $[g \cdot f](v) = f(g^{-1} \cdot v)$[3].

[3] You may wonder why algebraists add in the irritating-looking inverse, rather than simply defining $[g \cdot f](v) = f(g \cdot v)$. The choice is somewhat arbitrary, as as either definition would

For example, let $V = \mathbb{C}^2$, and let $G = \mathbb{Z}/4\mathbb{Z}$. We can define an action of G on V by associating with the element $\overline{1} \in G$ the linear transformation encoded by the matrix $\pi(\overline{1}) = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$, so that

$$\overline{1} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -y \\ x \end{bmatrix}.$$

Since π must be a homomorphism, and $\overline{1}$ generates G, this fully determines the group action:

$$\pi(\overline{1}) = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}, \quad \pi(\overline{2}) = \pi(\overline{1} + \overline{1}) = \pi(\overline{1}) \cdot \pi(\overline{1}) = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix},$$

$$\pi(\overline{3}) = \pi(\overline{1}+\overline{2}) = \pi(\overline{1}) \cdot \pi(\overline{2}) = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}, \quad \pi(\overline{0}) = \pi(\overline{1}+\overline{3}) = \pi(\overline{1}) \cdot \pi(\overline{3}) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Intuitively, we can think of this group action of G on V as rotation, with the element $\overline{1}$ rotating vectors counterclockwise by 90 degrees, the element $\overline{2}$ rotating vectors by 180 degrees, and so on.

To see the group acting on a polynomial, let $f(x, y) = 2x + 3y$. Then $[\overline{1} \cdot f](x, y) = f(y, -x) = 2y - 3x$. We can see that the group action transforms f into a new polynomial distinct from f. However, for any group action, there is a special class of polynomials that are unchanged by the action:

Definition 6 (Invariant polynomial)**.** Let V be a finite-dimensional vector space over a field k, and G a group acting on V. If $f \in k[V]$ is a polynomial over V, we say that f is invariant under G iff for all $g \in G$, $g \cdot f = f$.

Note that if two polynomials f_1 and f_2 are invariant under G, then $f_1 + f_2$ and $f_1 f_2$ are also invariant polynomials under G. Therefore the invariant polynomials in $k[V]$ under G form a ring. We call this the *invariant ring* of G, and denote it by $k[V]^G$. Theorems due to David Hilbert, Emmy Noether, and Hermann Weyl guarantee that when G is a "well-behaved" group (including the general linear group of matrices, any continuous subgroup, or any finite subgroup), the invariant ring of G is finitely generated (Hilbert (1893), Noether (1915), Weyl (1946)).

work, but to get a sense of why this way was chosen, imagine that the vectors v are points on the real line, and g is the transformation that shifts v right by 1 (to $v + 1$). Then we want g to shift the graph of the function f right by 1, which means that the output of $g \cdot f$ at v should be the same as the output of f at $v - 1$.

Continuing with our group action example, we may ask, what is the invariant ring of G within the polynomial ring $\mathbb{C}[V] = \mathbb{C}[x, y]$? This should consist of all polynomials in x and y of some degree D, represented by

$$f(x, y) = \sum_{\substack{i,j\in\mathbb{Z}^+\cup\{0\}\\ 0\leq i+j\leq D}} a_{ij}x^i y^j,$$

which are invariant under the action of $G = \mathbb{Z}/4\mathbb{Z}$. A polynomial will be invariant under all of G if and only if it is invariant under the generator $\overline{1}$, since $\overline{1}$ generates G. Note that

$$\overline{1}\cdot f(x, y) = f\left(\overline{1}^{-1}\cdot \begin{bmatrix}x\\ y\end{bmatrix}\right) = f\left(\overline{3}\cdot \begin{bmatrix}x\\ y\end{bmatrix}\right) = f\left(\begin{bmatrix}0 & 1\\ -1 & 0\end{bmatrix}\begin{bmatrix}x\\ y\end{bmatrix}\right) = f(y, -x),$$

so f is invariant if and only if $f(x, y)$ and $f(y, -x)$ are equal polynomials. In other words,

$$\sum_{\substack{i,j\in\mathbb{Z}^+\cup\{0\}\\ 0\leq i+j\leq D}} a_{ij}x^i y^j = \sum_{\substack{i,j\in\mathbb{Z}^+\cup\{0\}\\ 0\leq i+j\leq D}} a_{ij}y^i(-x)^j = \sum_{\substack{i,j\in\mathbb{Z}^+\cup\{0\}\\ 0\leq i+j\leq D}} (-1)^j a_{ij}x^j y^i,$$

where equality means that the coefficients of each $x^i y^j$ term are equal on both sides. For j even, we must have $a_{ij} = a_{ji}$, and for j odd, we must have $a_{ij} = -a_{ji}$. This implies that $a_{ij} = 0$ when i and j have opposite parity, $a_{ij} = -a_{ji}$ when i and j are both odd, and $a_{ij} = a_{ji}$ when i and j are both even. Therefore f can be expressed as

$$f(x, y) = \sum_{\substack{i,j\text{ even}\\ i<j}} \left(a_{ij}\cdot\left(x^i y^j + x^j y^i\right)\right) + \sum_{i\text{ even}} a_{ii}x^i y^i + \sum_{\substack{i,j\text{ odd}\\ i<j}} \left(a_{ij}\cdot\left(x^i y^j - x^j y^i\right)\right).$$

The set of possible polynomials produced by the first two sums is precisely the set of symmetric polynomials in x^2 and y^2, which are generated by the elementary symmetric polynomials $\{1, x^2 + y^2, x^2y^2\}$. It can be shown that polynomials produced by the third sum are all generated by $x^3y - xy^3$, x^2y^2, and $x^2 + y^2$. Therefore a full list of generators for the invariant ring $\mathbb{C}[x, y]^{\mathbb{Z}/4\mathbb{Z}}$ is $\{1, x^2 + y^2, x^2y^2, x^3y - yx^3\}$.

Listing the generators of an invariant ring allows us to fully specify the ring using only finitely many polynomials. However, as has been seen before in the case of finite time, complexity theorists are rarely satisfied with "finite." We want to specify invariant rings not just finitely, but as efficiently as possible. There are some invariant rings for which there may be no set of generators smaller than, say, exponential in the dimension of the vector space, but we would love a way to specify such invariant rings in less than exponential space. This can be done if the invariant ring in question has a succinct encoding.

Definition 7 (Succinct encoding of an invariant ring)**.** Let G be a group acting on an m-dimensional vector space V over a field k, with invariant ring $k[V]^G$. We say that an algebraic circuit $C(x_1, ..., x_m, y_1, ..., y_r)$ is a *succinct encoding* of $F[V]^G$ if the polynomials formed by different valuations of the y-variables, $\{C(x_1, ..., x_m, \alpha_1, ..., \alpha_r)\}_{\alpha_1,...,\alpha_r \in k}$, are a generating set for $k[V]^G$. Garg et al. (2019)

Suppose we want to find a succinct encoding of the invariant ring $\mathbb{C}[x, y]^{\mathbb{Z}/4\mathbb{Z}}$ we found above. It would have to take the form of an algebraic circuit $C(x, y, \alpha, \beta)$ such that all possible circuit outputs are invariant polynomials, and all the generators we found above, $\{1, x^2 + y^2, x^2y^2, x^3y - yx^3\}$, can be produced by as outputs (or sums and products of outputs) by different settings of α and β. Such a circuit is shown in Figure 2.1.

Just we can use as the size of smallest algebraic circuit computing f, $S(f)$, as a way to measure the inherent complexity of a polynomial f, we can use the size of the smallest succinct encoding to measure the inherent complexity of an invariant ring $k[V]^G$. One might expect that the inherent complexity of the invariant ring depends mainly on the properties of the group G, or on the complexity of the group action of G on V. However in the next chapter, we will see the construction of an invariant ring which under standard complexity theory assumptions has high inherent complexity, even though the group G and its associated group action are easy to describe.

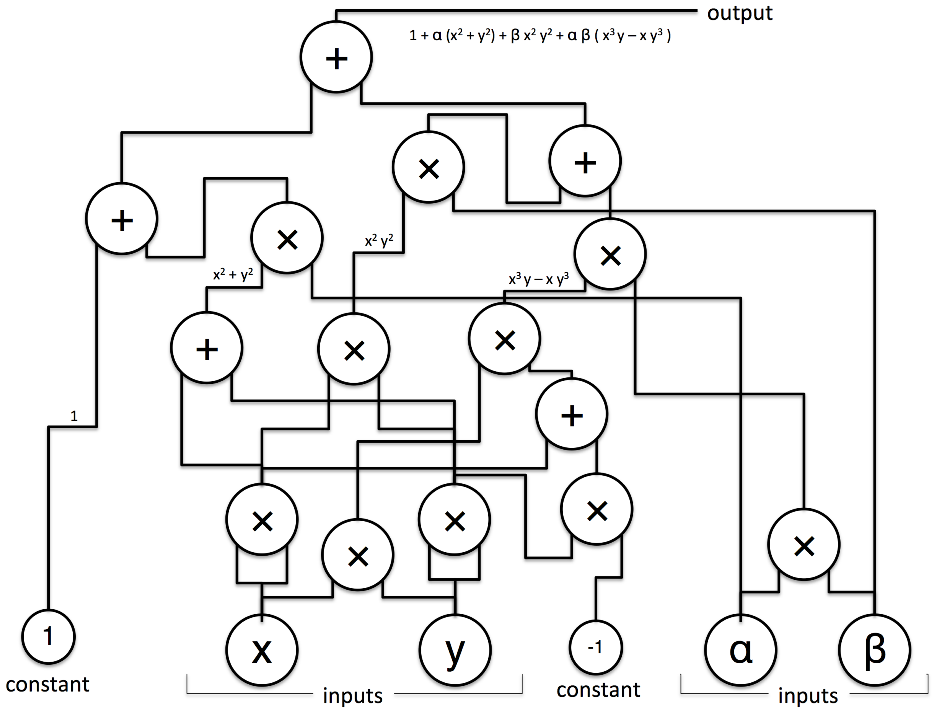


Figure 2.1 An algebraic circuit computing the polynomial $1 + \alpha(x^2 + y^2) + \beta x^2 y^2 + \alpha\beta(x^3 y - x y^3)$. As α and β run over all values in $\mathbb{C}$, this produces a generating set for the invariant ring $\mathbb{C}[x, y]^{\mathbb{Z}/4\mathbb{Z}}$, making it a succinct encoding for that invariant ring.

Chapter 3

Conditional Lower Bounds for Invariant Rings

This chapter describes work in invariant theory by Garg et al. (2019), which provides arithmetic circuit lower bounds for the succinct encoding of a particular invariant ring, under the standard complexity theory assumption that $\mathsf{NP} \not\subseteq \mathsf{P}/\text{poly}$. The authors construct a particular group action which is "simple" in that it can be easily described in polynomial time, but whose succinct encoding is "complex" meaning that if it were polynomially sized, it would give a polynomial size circuit for all problems in NP. After reading this chapter, if I have done my job correctly, you will understand precisely what group action this is, and why a succinct encoding for its invariant ring would have counterintuitive implications.

3.1 $\mathsf{NP} \not\subseteq \mathsf{P}/$poly, probably

First, what does $\mathsf{NP} \subseteq \mathsf{P}/\text{poly}$ mean? We have described the complexity class NP in Chapter 2. What is P/poly? It is the class of languages of binary strings that can be computed by a sequence (C_n) of polynomial-size Boolean circuits, with each circuit C_m being used to check inputs of size m. (We say a Boolean circuit *computes* a language if its output bit is 1 precisely when its input bits form a string in the language.) P/poly can be thought of as the "Boolean circuit equivalent" of the Turing machine-based class P. P describes languages computable in polynomial time by Turing machines, while P/poly describes languages computable in polynomial size by Boolean circuits.

In fact, it is known that $\mathsf{P} \subseteq \mathsf{P}/\text{poly}$. The proof of this result involves forming a circuit that simulates the behavior of a Turing machine at every point on its tape, and since the number of tape cells in a polynomial-time computation is polynomial, a polynomial-size circuit suffices to simulate a polynomial-time Turing machine. However, the opposite inclusion does not hold. This is because a Turing machine must have the same transition rules regardless of input length, but in a circuit sequence (C_n) there is a different circuit for each input length. This gives P/poly to compute many languages which are known to not be in P, since some of them may not even be computable.

For example, if one chooses some binary representation of Turing machines, then the Halting Language L_{halt} is defined as the language of strings $\langle M \rangle, w$, where $\langle M \rangle$ is a description of Turing machine M, w is an input to M, and M eventually halts on the string w. L_{halt} is known to not be computable by Turing machines in finite time. Now define L to be a related language, the unary language consisting of strings of n 0s, where n's binary representation $\langle n \rangle$ is a member of L_{halt}. L is not computable by Turing machines at all, and certainly not in polynomial time. However, $L \in \mathsf{P}$/poly, since one could construct a circuit sequence (C_n) where if C_n always outputs false for $\langle n \rangle \notin L_{halt}$, and C_n outputs true when all inputs are 0 for $\langle n \rangle \in L_{halt}$.

In essence, P/poly is more powerful than P since there are infinitely many circuits in (C_n), and so circuit sequences get an "extra piece of information" (the structure of the particular circuit C_n) for each input length. This idea leads to an equivalent formulation of P/poly as the class of problems that can be solved in polynomial time by *advised* Turing machines – Turing machines which on all inputs of length n, receive an "advice" string s_n of length polynomial in n. The advice levels the playing field between Turing machines and circuit sequences by giving Turing machines an extra piece of information for each input length as well, and this turns out to be enough to make the two computational models equivalent.

It is possible, given results that are currently proven, that $\mathsf{NP} \subseteq \mathsf{P}$/poly, even if $\mathsf{P} \neq \mathsf{NP}$. This would mean that all problems in NP, problems like the graph k-coloring problem, can be solved by polynomial-size circuits. However, after decades of study, NP-complete problems have resisted attempts to solve them with polynomial-size circuits, just as stubbornly as they have resisted polynomial-time algorithms, so it is often assumed that $\mathsf{NP} \not\subseteq \mathsf{P}$/poly.

3.2 3-way matching is NP-complete

The strategy of Garg et al. (2019) is to cleverly to turn a succinct encoding for their group action into a Boolean circuit sequence that solves an NP-complete problem. The NP-complete problem in question is the 3-way matching problem, which we will now describe. A k-uniform hypergraph is a graph whose edges are sets of k vertices – for example, an undirected graph is a 2-uniform hypergraph. A *tripartite* 3-uniform hypergraph is a 3-uniform hypergraph whose vertices can be partitioned into three sets X, Y, and Z, such that every edge in the hypergraph has exactly one vertex from each of X, Y, and Z. The 3-way matching problem asks, given a tripartite 3-uniform hypergraph G, does there exist a set of edges such that every vertex is included in one of the edges, and no two edges share vertices?

The 3-matching problem can be shown to be NP-complete by a reduction from the 3-satisfiability problem, a known NP-complete problem. Since it is NP-complete, if we had some sequence C_n of polynomially-sized circuits that accepted 3-matching, then given any language $L \in \mathsf{NP}$, we could construct circuits of polynomial size that first reduce L to 3-matching, and then use the appropriate circuit C_n to find the solution. Therefore all problems in NP would be solvable by polynomially sized circuits, meaning $\mathsf{NP} \subseteq \mathsf{P}/\text{poly}$.

3.3 The group action

We will now describe the particular group action for which a succinct encoding would yield a circuit sequence solving 3-matching. We start with the relevant group G. Let $\mathrm{ST}_n(\mathbb{C})$ represent the group of diagonal matrices with determinant 1; for $\mathbf{a} \in \mathrm{ST}_n(\mathbb{C})$, we will denote by a_i the i-th entry along the diagonal of $\mathbf{a}$. Then $G = \mathrm{ST}_n(\mathbb{C}) \times \mathrm{ST}_n(\mathbb{C}) \times \mathrm{ST}_n(\mathbb{C})$ is the group consisting of triples of determinant-1 diagonal matrices. The vector space G acts on will be $V = \mathbb{C}^n \otimes \mathbb{C}^n \otimes \mathbb{C}^n$, which for our purposes simply means the space of three-dimensional "boxes" of complex numbers with dimensions $n \times n \times n$. G will act on $u \in V$ by $(\mathbf{a}, \mathbf{b}, \mathbf{c}) \cdot u := v$ such that $v_{ijk} = a_i b_j c_k u_{ijk}$. We will represent polynomials over V as complex polynomials in the n^3 variables x_{ijk} for $1 \leq i, j, k \leq n$. We will refer to the "symbolic 3d box" of these variables as $\mathbf{x}$, so that a polynomial $f \in \mathbb{C}[V]$ can be written as $f(\mathbf{x})$.

Let us now understand the connection between this choice of group action, and the 3-matching problem. The action of G on V has invariant monomials which correspond to possible matchings of a tripartite

3-uniform hypergraph with disjoint vertex sets $\{\alpha_1, ..., \alpha_n\}$, $\{\beta_1, ..., \beta_n\}$, and $\{\gamma_1, ..., \gamma_n\}$. By a "possible matching" we mean a collection of triples $(\alpha_1, \beta_{\sigma(1)}, \gamma_{\tau(1)}), (\alpha_2, \beta_{\sigma(2)}, \gamma_{\tau(2)}), ..., (\alpha_n, \beta_{\sigma(n)}, \gamma_{\tau(n)})$ such that σ and τ are permutations of the integers from 1 to n; this matching is uniquely determined by σ and τ, so we will refer to it as $M_{\sigma\tau}$. $M_{\sigma\tau}$ is a true matching in the hypergraph if and only if $(\alpha_i, \beta_{\sigma(i)}, \gamma_{\tau(i)})$ is an edge in the hypergraph for all $i \in [n]$.

We associate with $M_{\sigma\tau}$ the degree-n monomial $\prod_{i=1}^n x_{i\sigma(i)\tau(i)}$. An element $(\mathbf{a}, \mathbf{b}, \mathbf{c}) \in G$ acts on a single-variable polynomial $f(\mathbf{x}) = x_{ijk}$ by $\left[(\mathbf{a}, \mathbf{b}, \mathbf{c}) \cdot f\right](\mathbf{x}) = f\left((\mathbf{a}, \mathbf{b}, \mathbf{c})^{-1} \cdot \mathbf{x}\right) = \frac{1}{a_i b_j c_k} \cdot x_{ijk}$. Every monomial $\prod_{i=1}^n x_{i\sigma(i)\tau(i)}$ is invariant under the group action, since

$$(\mathbf{a}, \mathbf{b}, \mathbf{c}) \cdot \prod_{i=1}^n x_{i\sigma(i)\tau(i)} = \prod_{i=1}^n \frac{x_{i\sigma(i)\tau(i)}}{a_i b_{\sigma(i)} c_{\tau(i)}}$$

$$= \left(\prod_{i=1}^n \frac{1}{a_i}\right)\left(\prod_{i=1}^n \frac{1}{b_{\sigma(i)}}\right)\left(\prod_{i=1}^n \frac{1}{c_{\tau(i)}}\right)\prod_{i=1}^n x_{ijk} = \prod_{i=1}^n x_{ijk}.$$

Here we have used the fact that $\prod_{i=1}^n \frac{1}{a_i} = 1$ because the a_i are the diagonal entries of matrix $\mathbf{a}$, which must multiply to $\det(\mathbf{a}) = 1$. Likewise $\prod_{i=1}^n \frac{1}{b_{\sigma(i)}} = 1$ and $\prod_{i=1}^n \frac{1}{c_{\tau(i)}} = 1$ since σ and τ are permutations, so $b_{\sigma(i)}$ and $c_{\tau(i)}$ as i ranges from 1 to n are exactly the diagonal entries of $\mathbf{b}$ and $\mathbf{c}$.

No other degree-n monomial in $\mathbf{x}$, nor any polynomial in $\mathbf{x}$ of degree less than n, is invariant under the action. For if g is such a monomial $g(\mathbf{x}) = \prod_{m=1}^r x_{i_m j_m k_m}$, $1 \le r \le n$, then the sets $\{i_m : 1 \le m \le r\}$, $\{j_m : 1 \le m \le r\}$, $\{k_m : 1 \le m \le r\}$ could not all contain all the integers from 1 to n (otherwise g would correspond to a matching). Suppose without loss of generality that $\{i_m : 1 \le m \le r\} \ne [n]$, and that $1 \notin \{i_m : 1 \le m \le r\}$. Then the action of $(\mathbf{a}, \mathbf{b}, \mathbf{c})$, with $\mathbf{a} = (2^{n-1}, \frac{1}{2}, \frac{1}{2}, ..., \frac{1}{2})$ and $\mathbf{b} = \mathbf{c} = (1, 1, 1, ..., 1)$, on g gives $(\mathbf{a}, \mathbf{b}, \mathbf{c}) \cdot g(\mathbf{x}) = \frac{1}{2^r} g(\mathbf{x}) \ne g(\mathbf{x})$, so g is not invariant.

It can be shown that given an algebraic circuit C computing a polynomial, there exists another algebraic circuit C_n which computes only the terms in that polynomial of degree less than n, with the size of C_n polynomial in n and the size of C.

To sum up this section, we (1) defined a group action of G on V, (2) showed that its invariant monomials of degree $\le n$ correspond precisely to the set of possible matchings of a tripartite hypergraph, and (3) mentioned that we could efficiently pull these monomials of degree $\le n$ out of a circuit computing a larger invariant polynomial. Let us now see how this leads us to a Boolean circuit that solves 3-matching.

3.4 If we had a succinct encoding, we could solve 3-matching

Recall that succinct encodings are circuits computing a set of generators for invariant rings. Assume that there is a succinct encoding $C(\mathbf{x}, y)$ for the invariant ring under the action of G on V (with the input y a set of variables whose different settings produce the different generators). Then there must also be a circuit $C_n(\mathbf{x}, y)$ that encodes a generating set for the invariant polynomials of degree $\leq n$. We will exploit this circuit C_n to check whether a given tripartite hypergraph H contains a matching.

Given a tripartite hypergraph H with vertices $\{\alpha_1, ..., \alpha_n\}$, $\{\beta_1, ..., \beta_n\}$, and $\{\gamma_1, ..., \gamma_n\}$, construct an element $\mathbf{v} \in V$ by letting $v_{ijk} = 1$ if the edge $(\alpha_i, \beta_j, \gamma_k)$ is in H, and $v_{ijk} = 0$ otherwise. Then note that for a possible matching $M_{\sigma\tau}$, the corresponding monomial

$$m_{\sigma\tau}(\mathbf{x}) = \prod_{i=1}^{n} x_{i\sigma(i)\tau(i)}$$

vanishes on H precisely when $M_{\sigma\tau}$ is a valid matching in H. For if it is a valid matching, then $v_{i\sigma(i)\tau(i)} = 1$ for all $i \in [n]$ and the monomial does not vanish, and if it is not a valid matching, then there is some $i \in [n]$ for which $v_{i\sigma(i)\tau(i)} = 0$, and the monomial does vanish. Therefore H has a valid matching if and only if there is some nonzero monomial in the output of $C_n(\mathbf{v}, y)$. Equivalently, H has a valid matching if and only if $C_n(\mathbf{v}, y)$ is a non-zero polynomial in y.

What we have described is therefore a reduction of the 3-way matching problem to the Polynomial Identity Testing problem, which asks, given an algebraic circuit, whether that circuit computes the identically zero polynomial. The goal is to show that 3-way matching is in P/poly, which is the class of problems that can be solved by polynomial-sized Boolean circuit sequences. Recall that P/poly is the class of problems that can be solved in polynomial time with polynomial advice. It also happens that P/poly is equal to BPP/poly, which means that if we relax the constraints on advised Turing machines and allow them to use randomness and make errors with some bounded probability, we can solve precisely the same set of problems.

Therefore if we can use our reduction to construct a randomized, bounded-error, polynomial-time algorithm by which an advised Turing machine can solve the 3-way matching problem, this will show that $\mathsf{NP} \subseteq \mathsf{P}/\text{poly}$, under the original assumption that our group action had a succinct encoding. We will use the fact that that there exists a randomized, bounded-error, polynomial-time algorithm for solving Polynomial Identity Testing (Zippel (1979) and Schwartz (1980)).

The randomized algorithm is as follows: Let the advice string for input of a tripartite graph H with $3n$ vertices be an encoding of the circuit $C_n(\mathbf{x}, y)$, which we showed to exist if the group action had a succinct encoding. Then convert H into a vector space element $\mathbf{v} \in V$ as described above, and evaluate the circuit $C_n(\mathbf{v}, y)$ as a circuit in the y variables. Use the existing randomized algorithm for Polynomial Identity Testing to check whether $C_n(\mathbf{v}, y)$ is the zero polynomial. If it is not, then H has a matching, so return "yes," and otherwise, return "no."

This algorithm is correct by our arguments above, and proves that if the group action has a succinct encoding, then 3-way matching is in P/poly, and thus $\mathsf{NP} \subseteq \mathsf{P}/\text{poly}$ since 3-way matching is NP-complete.

3.5 Conclusion

We have described an invariant ring which, if $\mathsf{NP}/\mathsf{P}/\text{poly}$, has a high degree of "inherent complexity." Garg et al. (2019) also give an example of an invariant ring for which a succinct encoding would imply $\mathsf{VP} = \mathsf{VNP}$. Their work suggests that succinct encodings for reasonably simple group actions can have many counterintuitive results for complexity theory. The general takeaway is that the situation of a group acting on a vector space provides a malicious complexity theorist, who is seeking to construct an invariant ring that is hard to encode, with a great deal of leeway to encode hard problems inside of the group action. This means that general efficient algorithms for finding succinct encodings for invariant rings are unlikely to exist. However, if the results are ever able to move from conditional to unconditional, they may provide a starting point for proving lower bounds on polynomials arising from invariant rings, which has proved a difficult subject. In Chapter 4, we will explore one of the few known results in this area, that being the lower bounds on elementary symmetric polynomials.

Chapter 4

Polynomial Lower Bounds

This chapter explores the known lower bound on elementary symmetric polynomials, proved by Baur and Strassen (1983). Their proof is in two parts, one relying on the partial derivative operator preserving properties of arithmetic circuit complexity, and the other relying on lower bounds from algebraic geometry obtained earlier by Strassen (1973a). We will prove the former part in detail, and give a high-level overview of the latter. Our explanation also draws extensively from chapter 12 of the survey text *Mathematics and Computation* (Wigderson (2019)).

Given a polynomial f, recall that we write $S(f)$ to denote the size of the smallest arithmetic circuit computing f. In this thesis thus far we have been speaking loosely about it being "hard to prove lower bounds," however, what we really mean is that it is hard to prove *nontrivial* lower bounds, or lower bounds which exploit the properties of the specific polynomial in question and aren't universally applicable to all functions. We do in fact know a basic ("trivial") circuit lower bound on any polynomial f, which is the following:

$$S(f) \geq \log_2(\deg f).$$

Here $\deg(f)$ represents the largest sum of exponents among the terms of f; for example, if $f(x, y) = x^2y^3 + x^4$ then $\deg(f) = 5$. This result can also be expressed asymptotically, using "big-omega" notation:

$$S(f) \geq \Omega(\log(\deg f)),$$

which means that for $\deg f$ above some threshold integer N, there exists some constant C such that $S(f) \geq C \cdot \log(\deg f)$.

The reason that this lower bound is known is that to obtain a term of the form x^d, it is necessary to multiply together d copies of the variable x. It can be shown that the fastest possible way to do this is to multiply x by itself to get x^2, then multiply x^2 by itself to get x^4, and so on until an exponent on the order of d is reached, for approximately $\log_2 d$ multiplications. This logic still applies for multi-variable terms of total degree d. So this lower bound simply reflects the nature of multiplication, rather than any special properties of a particular polynomial in question. Bauer and Strassen's achievement is to establish a lower bound on elemenary symmetric polynomials that is greater than simply the logarithm of their degrees.

4.1 Elementary symmetric polynomials

To begin with, what is an elementary symmetric polynomial? Assume a set of n objects, $[n] = \{1, 2, 3, ..., n\}$. A *permutation* of $[n]$ is an bijective function $\sigma : [n] \to [n]$, which can be thought of as simply a reordering of the objects. The set of all permutations of $[n]$ form a group S_n under function composition, called the *symmetric group* on n elements. Fix a field k and let S_n act on the vector space $V = k^n$ by permuting the coefficients of a given vector in the natural way (i.e. for $\sigma \in S_n$, the i-th coefficient of $v \in k^n$ is the $\sigma(i)$-th coefficient of $\sigma \cdot v$).

We are interested in the invariant ring of this action, $k[V]^{S_n}$, which consists of all n-variate polynomials which are unchanged by permuting their coefficients. Polynomials in $k[V]^{S_n}$ are called *symmetric polynomials*. It can be shown that a natural generating set for the non-constant symmetric polynomials is given by:

$$\begin{aligned}
e_1^n(x_1, x_2, ..., x_n) &= x_1 + x_2 + \cdots + x_n \\
e_2^n(x_1, x_2, ..., x_n) &= x_1x_2 + x_1x_3 + \cdots + x_{n-1}x_n \\
&\vdots \\
e_d^n(x_1, x_2, ..., x_n) &= \sum_{1 \le j_1 < j_2 < \cdots < j_d \le n} x_{j_1}x_{j_2} \cdots x_{j_d} \\
&\vdots \\
e_n^n(x_1, x_2, ..., x_n) &= x_1x_2 \cdots x_n
\end{aligned}$$

These are called the *elementary symmetric polynomials* (the e is for elementary) in n variables. Bauer and Strassen prove the following, which is one of the only known arithmetic circuit lower bound beyond the naïve logarithmic lower bounds:

Theorem 1. *If e^n_d is the d-th elementary symmetric polynomial in n variables, then*

$$S(e^n_d) \geq \Omega(n \log r),$$

where $r = \min(d, n-d)$.

Note that this improves the naïve lower bound $S(e^n_d) \geq \Omega(\log d)$ by a factor of n.

4.2 Proof overview

We will start with an overview of how Theorem 1 is proved. We will assume the case where $d \leq \frac{n}{2}$, so $r = d$; the result then follows for $d > \frac{n}{2}$ because

$$e^n_d = e^n_{n-d}\left(\frac{1}{x_1}, \ldots, \frac{1}{x_n}\right) \cdot x_1 \cdot x_2 \cdot \cdots \cdot x_n.$$

This is computable with a circuit of size $O(S(e^n_{n-d}) + n)$.[1]

As mentioned above, the proof arises from combining two theorems. Before we can state the theorems, we need to define two key pieces of notation.

For $f \in k[x_1, \ldots, x_n]$, ∇f is the *gradient* of f, or the tuple $(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n})$ consisting of all partial derivatives of f. Below we will speak of $S(\nabla f)$, and you may wonder what it means to apply S to a tuple of polynomials. By $S(f_1, f_2, \ldots, f_m)$, we mean the size of the smallest arithmetic circuit with m outputs such that the first output is f_1, the second is f_2, and so on.

Finally, the *degree* of a tuple of polynomials $f_1, f_2, \ldots, f_m$, denoted by $\deg(f_1, f_2, \ldots, f_m)$, is a notion from algebraic geometry that generalizes the degree of a single polynomial. We will not be able to define it here without giving a full course's worth of background in algebraic geometry, but it is computed by examining the surface in $2n$-dimensional space defined by the system $z_1 = f_1(x_1, \ldots, x_n), z_2 = f_2(x_1, \ldots, x_n), \ldots, z_n = f_n(x_1, \ldots, x_n)$.

[1] The reciprocals can be used to construct a circuit because Bauer and Strassen allow divisions in their circuits, which is a wrinkle we will not discuss here, for the sake of brevity.

Theorem 2. *For any polynomial f,*

$$S(\nabla f) \leq 5 \cdot S(f).$$

Theorem 3. *For any tuple of polynomials $f_1, \dots, f_m$,*

$$S(f_1, \dots, f_m) \geq \Omega(\log \deg(f_1, f_2, \dots, f_m)).$$

Theorem 2 asserts that there is no polynomial f that is more than a constant factor easier to compute than its partial derivatives. Theorem 3 (which was proven in Strassen (1973a)) generalizes the trivial lower bound $S(f) \geq \Omega(\log \deg f)$ discussed above to tuples of polynomials, using the generalized degree that can apply to such tuples.

Bauer and Strassen prove Theorem 1 by computing the partial derivatives of e_d^n, and then showing that the generalized degree of this set of polynomials is d^n. This then shows that

$$S(e_d^n) \geq \Omega\left(S\left(\frac{\partial e_d^n}{\partial x_1}, \dots, \frac{\partial e_d^n}{\partial x_n}\right)\right) \geq \Omega\left(\log \deg\left(\frac{\partial e_d^n}{\partial x_1}, \dots, \frac{\partial e_d^n}{\partial x_n}\right)\right)$$
$$= \Omega\left(\log(d^n)\right) = \Omega(n \log d),$$

with Theorem 2 providing the first inequality and Theorem 3 providing the second.

4.3 Partial derivatives preserve lower bounds

In this section we will prove Theorem 2. Our proof is based on the proof given by Morgenstern (Morgenstern (1985)).

Assume we have some polynomial f over a field k and n variables $x_1, \dots, x_n$. We will simplify our induction by proving the slightly stronger statement that for any polynomial f whose smallest circuit A has size $S(f)$, there is a circuit B computing ∇f that not only has $\text{size}(B) \leq 5 \cdot \text{size}(A)$, but also uses all of the same constants that A uses. Note that we will not consider the constants 0 or 1 to have any bit complexity.

Our proof will proceed by induction on $S(f)$. When $S(f) = 0$, f requires no operations to compute. Thus f is either the constant 0 or 1, or a single variable x_i. If f is constant, than all its partial derivatives are zero, so $S(\nabla f) = 0 = S(f)$. If f is x_i, then it has a single partial derivative which is 1, and the rest are zero, so again $S(\nabla f) = 0 = S(f)$.

If $S(f) > 0$ but f can be computed by a circuit with no additions or multiplications, then f consists of some constant c, and $S(f) = \lceil \log_2 c \rceil$. Then there must be a circuit for ∇f that has a constant gate for c which it ignores, and returns 0 or for all partial derivatives. This circuit shows that $S(\nabla f) \leq S(f) \leq 5 \cdot S(f)$.

Now we consider the case where $S(f) > 0$ and f is non-constant, and we assume by way of induction that the theorem holds for polynomials f' with $S(f') < S(f)$. Let A be a circuit of size $S(f)$ computing f, with N operations. Let $g_1, g_2, ..., g_N$ be the functions computed by the multiplication and addition gates of A, in an ordering such that every gate occurs after its inputs.

We can imagine the computation of A as occurring by first listing the variables $x_1, ..., x_n$, then the constants $c_1, ..., c_m$ used in the computation, then listing each g_i as a sum or product of two items previously listed. For example, a circuit computing the polynomial $f(x, y) = 2xy + y$ might look like:

$$x_1 = x, \quad x_2 = y, \quad c_1 = 2, \quad g_1 = x_1 \times c_1 = 2x,$$

$$g_2 = x_2 \times g_1 = 2xy, \quad g_3 = x_2 + g_2 = 2xy + y = f.$$

In this case, assuming this is the smallest circuit computing f (which it is), we would have 3 gates and a bit complexity of $\log_2(2) = 1$, so $S(f) = 4$.

Now, consider the polynomial f' on $x_1, ..., x_n, x_{n+1}$ that satisfies

$$f'(x_1, ..., x_n, g_1(x_1, ..., x_n)) = f(x_1, ..., x_n).$$

f' can be computed by a circuit A' using $N - 1$ gates and the same set of constants, by using the same computation strategy as in A, but treating g_1 as an indeterminate. Thus $S(f') \leq \text{size}(A') - 1 = S(f) - 1$.

Let us see how this works for our example polynomial $f(x, y) = 2xy + y$. In our example circuit for f above, we would define $f'(x, y, z) = yz + y$, so that $f'(x, y, g_1(x, y)) = f'(x, y, 2x) = 2xy + y = f(x, y)$. Then f' can be computed by the following circuit A':

$$x_1 = x, \quad x_2 = y, \quad x_3 = z, \quad c_1 = 2,$$

$$g_1' = x_2 \times x_3 = yz, \quad g_2' = x_2 + g_1' = yz + y = f',$$

which uses $3 - 1 = 2$ gates, and no additional constants. Thus the size of A' is 3, which is strictly smaller than A, the optimal circuit for f, meaning that $S(f') \leq S(f) - 1$.

Since $S(f') < S(f)$, we can apply the inductive hypothesis to get $S(\nabla f') \leq 5 \cdot S(f')$. So we can assume we have a circuit B' of size at most $5 \cdot S(f')$ that computes $\nabla f' = \left(\frac{\partial f'}{\partial x_1}, \frac{\partial f'}{\partial x_2}, \ldots, \frac{\partial f'}{\partial x_n}, \frac{\partial f'}{\partial x_{n+1}}\right)$, and also uses all the same constants that A' and A use. By transforming B' while adding no more than 5 to its size, we will turn it into a circuit B computing $\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right)$ in size at most $5 \cdot S(f') + 5 \leq 5 \cdot S(f)$.

To see how, note that since we have

$$f'(x_1, \ldots, x_n, g_1(x_1, \ldots, x_n)) = f(x_1, \ldots, x_n),$$

the chain rule gives

$$\frac{\partial f}{\partial x_i} = \frac{\partial f'}{\partial x_i} + \frac{\partial f'}{\partial x_{n+1}} \cdot \frac{\partial g_1}{\partial x_i}.$$

Therefore the partial derivatives of f are expressible in terms of the partials derivatives of f' and g_1. In B', we already have the partial derivatives of f' in terms of $x_1, \ldots, x_n, x_{n+1}$, and if we add the orginal first gate g_1 back in, this will give us x_{n+1} in terms of the other variables, and thus give access to all the partials of f' in terms of $x_1, \ldots, x_n$.

Thus by adding 1 gate g_1, all that is left is to compute the partials of g_1, and carry out the addition and multiplication that characterizes the chain rule. We must be able to do this by adding no more than $5 - 1 = 4$ to the size complexity of the circuit. Fortunately, since g_1's children are either variables or constants, its derivatives will be easy to compute and plug into the chain rule. For all the x_i which are not inputs to the gate g_1, $\frac{\partial g_1}{\partial x_i} = 0$, and we simply have $\frac{\partial f}{\partial x_i} = \frac{\partial f'}{\partial x_i}$, so the circuit B' already computes $\frac{\partial f}{\partial x_i}$. Therefore, we only need to add gates onto B' to compute the derivatives for variables involved in gate g_1.

The next step of the proof is best handled in cases, depending on the nature of g_1:

Case 1: $g_1 = x_i + c_j$

In this case, $\frac{\partial g_1}{\partial x_i} = 1$, so

$$\frac{\partial f}{\partial x_i} = \frac{\partial f'}{\partial x_i} + \frac{\partial f'}{\partial x_{n+1}},$$

which can be computed by appending a single addition gate onto B', for a complexity increase of 1.

Case 2: $g_1 = x_i + x_j$, $i \neq j$

In this case, $\frac{\partial g_1}{\partial x_i} = \frac{\partial g_1}{\partial x_j} = 1$, so

$$\frac{\partial f}{\partial x_i} = \frac{\partial f'}{\partial x_i} + \frac{\partial f'}{\partial x_{n+1}}$$

and

$$\frac{\partial f}{\partial x_j} = \frac{\partial f'}{\partial x_j} + \frac{\partial f'}{\partial x_{n+1}}.$$

We can compute the two by appending two addition gates onto B', for a complexity increase of 2.

Case 3: $g_1 = x_i + x_i$

In this case, $\frac{\partial g_1}{\partial x_i} = 2$, so

$$\frac{\partial f}{\partial x_i} = \frac{\partial f'}{\partial x_i} + 2\frac{\partial f'}{\partial x_{n+1}}.$$

We can compute the two by appending the constant 2 (which adds a bit complexity of 2), a multiplication gate, and an addition gates onto B', for a complexity increase of 4.

Case 4: $g_1 = x_i \times c_j$

In this case, $\frac{\partial g_1}{\partial x_i} = c_j$, so

$$\frac{\partial f}{\partial x_i} = \frac{\partial f'}{\partial x_i} + c_j\frac{\partial f'}{\partial x_{n+1}}.$$

Since we assume that B' uses all the same constants as A (here we see why that wrinkle is necessary), c_j must already be a constant in B', meaning that $\frac{\partial f}{\partial x_i}$ can be computed by appending multiplication gate and an addition gate onto B', for a complexity increase of 2.

Case 5: $g_1 = x_i \times x_j$, $i \neq j$ In this case, $\frac{\partial g_1}{\partial x_i} = x_j$ and $\frac{\partial g_1}{\partial x_j} = x_i$, so

$$\frac{\partial f}{\partial x_i} = \frac{\partial f'}{\partial x_i} + x_j\frac{\partial f'}{\partial x_{n+1}}$$

and

$$\frac{\partial f}{\partial x_j} = \frac{\partial f'}{\partial x_j} + x_i\frac{\partial f'}{\partial x_{n+1}}.$$

We can compute these two partial derivatives by with two additional gates each (an addition and a multiplication) added onto B', for a complexity increase of 4.

Case 6: $g_1 = x_i \times x_i$ In this case, $\frac{\partial g_1}{\partial x_i} = 2x_i$, so

$$\begin{aligned}\frac{\partial f}{\partial x_i} &= \frac{\partial f'}{\partial x_i} + 2x_i \frac{\partial f'}{\partial x_{n+1}} \\ &= \frac{\partial f'}{\partial x_i} + (x_i + x_i)\frac{\partial f'}{\partial x_{n+1}}.\end{aligned}$$

We can compute this with three additional gates (two additions and one multiplication) added onto B', for a complexity increase of 3.

We can see that regardless of what g_1 is, we need add at most 5 gates to B' (1 to compute g_1, and ≤ 4 to compute the chain rule) in order to create a circuit B that computes ∇f. Thus

$$S(\nabla f) \leq \text{size}(B') + 5 = S(\nabla f') + 5 \leq 5 \cdot S(f') + 5 \leq 5(S(f) - 1) + 5 = 5 \cdot S(f).$$

This concludes our proof.

4.4 The lower bound

Now consider Theorem 2 applied to the elementary symmetric polynomial e^n_d. Note that

$$\frac{\partial e^n_d}{\partial x_i} = \frac{\partial}{\partial x_i} \sum_{1 \leq j_1 < j_2 < \cdots < j_d \leq n} x_{j_1} x_{j_2} \cdots x_{j_d}$$

$$= \sum_{\substack{1 \leq j_1 < j_2 < \cdots < j_{d-1} \leq n \\ i \notin \{j_1, \ldots, j_{d-1}\}}} x_{j_1} x_{j_2} \cdots x_{j_{d-1}} = e^{n-1}_{d-1}(x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n).$$

Denote this last polynomial, e^{n-1}_{d-1} applied to $x_1, \ldots, x_n$ with x_i removed, by $e^{n-1}_{d-1}(\hat{i})$. Theorem 2 then gives that

$$S(e^n_d) \geq \Omega\left(S(\nabla e^n_d)\right) = \Omega\left(S(e^{n-1}_{d-1}(\hat{1}), e^{n-1}_{d-1}(\hat{2}), \ldots, e^{n-1}_{d-1}(\hat{n}))\right).$$

We will not give an explanation of how Theorem 3 is proved, since this would require an in-depth dive into algebraic geometry. However, Bauer and Strassen rely on this result, and prove that

$$\deg\left(e^{n-1}_{d-1}(\hat{1}), e^{n-1}_{d-1}(\hat{2}), \ldots, e^{n-1}_{d-1}(\hat{n})\right) = (d-1)^n = \Omega(d^n).$$

Therefore

$$S(e_d^n) \geq \Omega(\log d^n) = \Omega(n \log d).$$

Theorem 2 seems to be a powerful tool, in that it can be used to obtain lower bounds on one polynomial based on other lower bounds (namely those on its partial derivatives). It is a tool for translating lower bounds between polynomials that are related by the partial derivative operator. In the next chapter, we will explore the question of whether Theorem 2 can be further applied to translate Bauer and Strassen's lower bounds into other polynomials besides elementary symmetric polynomials.

Chapter 5

Can We Go Further?

This chapter will discuss the potential implications of Bauer and Strassen's lower bound discussed in Chapter 4, and possible strategies for generalizing it beyond just elementary symmetric polynomials. This chapter reflects exploratory work by the author, and though it is light on new results, it will propose original questions and ideas for solving them. It is my hope that the ideas here may provide an avenue for future research in arithmetic circuit lower bounds.

5.1 Simple extensions

As mentioned in Chapter 4, Bauer and Strassen's lower bound for elementary symmetric polynomials (Theorem 1) is one of the only known non-trivial arithmetic circuit lower bounds. Given that all symmetric polynomials are expressible in terms of these elementary symmetric polynomials, we can intuitively think of this result as a quantification of the intrinsic complexity of symmetric polynomials.

But there are many polynomials whose intrinsic complexity we are interested in that are not symmetric (such as, notably, the determinant and the permanent). In many cases these polynomials, while not symmetric in the sense of being invariant with respect to every permutation of their variables, do have some type of inherent symmetry. For example, the determinant generates the invariant ring of a simple action involving matrix multiplication (for a description, we refer the reader to Garg et al. (2019)).

Since Theorem 1 tells us that objects with a particular kind of symmetry are complex, it is natural to wonder what it might imply for objects with other kinds of symmetry. Can we somehow extend Theorem 1 to find lower bounds on polynomials which are not symmetric polynomials, but nonetheless have a high degree of symmetry?

The short answer is yes, although not all extensions of their result are very interesting. For example, for integers n that are a perfect square $n = m^2$, for $1 \leq d \leq m$, define

$$\tau_d^n(x_1, \ldots, x_n) = e_d^m(x_1 + \cdots + x_m, x_{m+1} + \cdots + x_{2m}, \ldots, x_{n-m+1} + \cdots + x_n).$$

For example,

$$\begin{aligned} \tau_3^9(x_1, \ldots, x_9) &= e_3^3(x_1 + x_2 + x_3, x_4 + x_5 + x_6, x_7 + x_8 + x_9) \\ &= (x_1 + x_2 + x_3)(x_4 + x_5 + x_6)(x_7 + x_8 + x_9). \end{aligned}$$

Note that in general τ_d^n is not a symmetric polynomial – exchanging x_1 and x_4 in the above expression would yield a polynomial with different coefficients. It is however invariant with respect to permutations that act only on the m variables $\{x_{mi+1}, x_{mi+2} \ldots, x_{m(i+1)}\}$, for each $i = 0, 1, \ldots, m-1$. It is therefore in the invariant ring of a *subgroup* of the full permutation group S_n (specifically the subgroup generated by the aforementioned permutations).

It can be straightforwardly shown that Theorem 1 implies the following for τ_d^n:

Theorem 4. *If τ_d^n is as defined above, and $r = \min(d, \sqrt{n} - d)$, then*

$$S(\tau_d^n) \geq \Omega(\sqrt{n} \log r).$$

Proof. We will demonstrate this in two ways. The first way is by analogy with how Theorem 1 was proved. We find the gradient of τ_d^n:

$$\nabla \tau_d^n = \left(\frac{\partial \tau_d^n}{\partial x_1}, \ldots, \frac{\partial \tau_d^n}{\partial x_n} \right) = \left(e_{d-1}^{m-1}(\hat{1}), \ldots, e_{d-1}^{m-1}(\hat{m}) \right),$$

where we are now using $e_{d-1}^{m-1}(\hat{i})$ to denote e_{d-1}^{m-1} applied to the list $x_1 + \cdots + x_m, \ldots, x_{n-m+1} + \cdots + x_n$ with $x_{m(i-1)+1} + \cdots + x_{mi}$ removed. Now the partial derivative theorem (Theorem 2) guarantees that a lower bound on computing this tuple of polynomials is also a lower bound on computing τ_d^n (to within a constant factor). This tuple is identical to the tuple ∇e_d^m, with the exception that single variables have been replaced with sums of m variables, with no two sums sharing terms. The same logic that Bauer and Strassen used to show that ∇e_d^m has degree r^m can now be applied to show that $\nabla \tau_d^n$ has degree r^m. Therefore by Theorem 3,

$$S(\tau_d^n) \geq \Omega\left(S(\nabla \tau_d^n)\right) \geq \Omega\left(\log r^m\right) = \Omega(m \log r) = \Omega(\sqrt{n} \log r).$$

Our second proof method is simpler. Note that if we set $x_i = 0$ for all i that are not multiples of m, then

$$\tau_d^n(x_1, x_2, ..., x_n) = e_d^m(x_1 + \cdots + x_m, x_{m+1} + \cdots + x_{2m}, ..., x_{n-m+1} + \cdots + x_n)$$

$$= e_d^m(0 + \cdots + 0 + x_m, 0 + \cdots + 0 + x_{2m}, ..., 0 + \cdots + 0 + x_n) = e_d^m(x_m, x_{2m}, ..., x_n).$$

If we had a circuit computing τ_d^n in less than $\Omega(m \log r)$ gates, then we could compute e_d^m in less than $\Omega(m \log r)$ as well by simply replacing those variable inputs with constant 0s. This is impossible by Theorem 1.

□

The simplicity (or, as some might say, the triviality) of our second proof method suggests that the τ_d^n family is not all that interesting of an extension of Theorem 1. It is just $e_d^{\sqrt{n}}$ with each variable replaced by a sum of $\sqrt{n}$ variables, and as such it behaves just like $e_d^{\sqrt{n}}$.

How can we find a more interesting extension? The idea we will explore in the remainder of this chapter is to look for a class of polynomials which are not symmetric, but whose *partial derivatives* are symmetric. The rationale behind this idea is that Theorem 1 intuitively tells us that symmetric polynomials are inherently complex, and Theorem 2 tells us that functions are as complex as their partial derivatives. So if we could find a polynomial whose partial derivatives are symmetric, we might be able to prove it shares the inherent complexity of those derivatives. It turns out that much of the difficulty of following this line of reasoning comes is hidden by that little word "find."

5.2 Which polynomials have symmetric partial derivatives?

For all you fans of formal notation out there, let's formally define exactly what the title of this section is asking. For convenience we'll use the notation f_{x_i} to represent $\frac{\partial f}{\partial x_i}$, and $[n]$ to represent $\{1, 2, ..., n\}$.

Fix an integer n and a field k. Let S_n be the symmetric group on n elements, and let it act on the vector space k^n by permuting coefficients in the natural way. Then we want to characterize the set of n-variate polynomials $f \in k[x_1, ..., x_n]$ such that for every x_i, f_{x_i} is symmetric. That is, the set

$$D_{n,k} = \{f \in k[x_1, ..., x_n] \mid f_{x_i} \in k[x_1, ..., x_n]^{S_n} \text{ for all } i \in [n]\}.$$

We will hereafter omit the field k and write simply D_n.

One interesting feature of D_n is that unlike $k[x_1, ..., x_n]^{S_n}$, or any other invariant ring, D_n is not a ring in general, because it is not necessarily closed under multiplication. Symmetric polynomials form a ring because if we have $f, g \in k[x_1, ..., x_n]^{S_n}$, then both $f + g$ and fg are also symmetric. This reflects the fact that both addition and multiplication *commute* with the operation of permuting the variables. If one permutes the variables of f and g and then adds/multiplies them, the result is the same as if one first adds/multiplies them, then permutes the variables of the result. Therefore for any permutation σ,

$$\sigma \cdot (f + g) = (\sigma \cdot f) + (\sigma \cdot g) = f + g,$$

and

$$\sigma \cdot (fg) = (\sigma \cdot f)(\sigma \cdot g) = fg,$$

so $f + g$ and fg are in $k[x_1, ..., x_n]^{S_n}$. In contrast, the operation that first takes a partial derivative with respect to x_i, and then permutes the variables, commutes with addition but not multiplication. Formally, if $f, g \in D_n$, then for all $i \in [n]$,

$$\sigma \cdot (f + g)_{x_i} = \sigma \cdot (f_{x_i} + g_{x_i}) = (\sigma \cdot f_{x_i}) + (\sigma \cdot g_{x_i}) = f_{x_i} + g_{x_i} = (f + g)_{x_i},$$

so $f + g \in D_n$. However, for $i \in [n]$,

$$\sigma \cdot (fg)_{x_i} = \sigma \cdot (f_{x_i}g + fg_{x_i}) = (\sigma \cdot f_{x_i}g) + (\sigma \cdot fg_{x_i}) = f_{x_i}(\sigma \cdot g) + (\sigma \cdot f)g_{x_i},$$

which is not in general equal to $(fg)_{x_i}$ since f and g are not necessarily symmetric. Thus fg may not be in D_n.

D_n is not a ring, but it is a vector space over k, since it is closed under both addition, and multiplication by constants in k. Therefore instead of trying to describe D_n with a generating set, we will look for a basis, or a set such that every polynomial in D_n can be expressed uniquely as a linear combination of set elements.

To understand D_n better, let us start with small particular values of n. D_1 is simply the set of all univariate polynomials, since all univariate polynomials (and their derivatives) are symmetric with respect to the single permutation of the variable x (the identity permutation). A basis for D_1 is given by $\{1, x, x^2, x^3, ...\}$. Note that since all univariate polynomials are trivially symmetric, there are no univariate polynomials which are not symmetric but are in D_1.

D_2 is the set of polynomials $f \in k[x, y]$ such that $f_x(x, y) = f_x(y, x)$ and $f_y(x, y) = f_y(y, x)$. Let f be any polynomial in D_2, and assume for now that it is homogeneous of degree d (meaning every term in f with a non-zero coefficent has degree d). Then

$$f(x, y) = \sum_{i=0}^{d} a_i x^i y^{d-i}.$$

The constraint $f_x(x, y) = f_x(y, x)$ implies that

$$f_x(x, y) = \sum_{i=1}^{d} i a_i x^{i-1} y^{d-i}$$

$$= f_x(y, x) = \sum_{i=1}^{d} i a_i x^{d-i} y^{i-1} = \sum_{i=1}^{d} (d - i + 1) a_{d-i+1} x^{i-1} y^{d-i}.$$

For these two polynomials to be equal, we must have $i a_i = (d - i + 1) a_{d-i+1}$ for all $i = 1, ..., d$. Likewise, the constraint $f_y(x, y) = f_y(y, x)$ implies that

$$f_y(x, y) = \sum_{i=0}^{d-1} (d - i) a_i x^i y^{d-i-1}$$

$$= f_y(y, x) = \sum_{i=0}^{d-1} (d - i) a_i x^{d-i-1} y^i = \sum_{i=0}^{d-1} (i + 1) a_{d-i-1} x^i y^{d-i-1}.$$

This constraint gives that we must have $(d - i) a_i = (i + 1) a_{d-i-1}$ for all $i = 0, ..., d - 1$.

In summary, here are the constraints we have found, written out in a clearer form:

$$\begin{array}{ll}
a_1 = d a_d & d a_0 = a_{d-1} \\
2a_2 = (d-1) a_{d-1} & (d-1) a_1 = 2a_{d-2} \\
3a_3 = (d-2) a_{d-2} & (d-2) a_2 = 3a_{d-3} \\
\vdots & \vdots \\
d a_d = a_1 & a_{d-1} = d a_0
\end{array}$$

Let's start with a_0 and try to use these constraints to get as many coefficients as we can in terms of a_0. We proceed as follows:

$$\begin{aligned} a_{d-1} &= da_0, \\ a_2 = \frac{d-1}{2}a_{d-1} &= \frac{d(d-1)}{2}a_0 = \binom{d}{2}a_0, \\ a_{d-3} = \frac{d-2}{3}a_2 &= \frac{d(d-1)(d-2)}{6}a_0 = \binom{d}{3}a_0, \\ a_4 = \frac{d-3}{4}a_{d-3} &= \frac{d(d-1)(d-2)(d-3)}{24}a_0 = \binom{d}{4}a_0, \cdots \end{aligned}$$

You may now see the pattern. In general, we are finding that for even i, $a_i = \binom{d}{i}a_0$, and for odd j, $a_{d-j} = \binom{d}{j}a_0$, or equivalently $a_{d-j} = \binom{d}{d-j}a_0$. When d is an even integer, these constraints fully determine every coefficient in terms of a_0, and we get:

$$\begin{aligned} f(x,y) &= a_0 \cdot \left(y^d + \binom{d}{1}xy^{d-1} + \binom{d}{2}x^2y^{d-2} + \cdots + \binom{d}{i}x^iy^{d-i} + \cdots + x^d\right) \\ &= a_0 \cdot (x+y)^d. \end{aligned}$$

If, on the other hand, d is odd, then this only allows us to express a_i in terms of a_0 for even i. Performing an equivalent process starting with a_d allows us to say that $a_i = \binom{d}{i}a_d$ for odd i. Therefore we get:

$$\begin{aligned} f(x,y) = a_0 &\cdot \left(y^d + \binom{d}{2}x^2y^{d-2} + \cdots + \binom{d}{d-1}xy^{d-1}\right) \\ &+ a_d \cdot \left(\binom{d}{1}xy^{d-1} + \binom{d}{3}x^3y^{d-3} + \cdots + x^d\right) \\ &= a_0 \cdot \frac{(x+y)^d + (x-y)^d}{2} + a_d \cdot \frac{(x+y)^d - (x-y)^d}{2} \\ &= \frac{a_0 + a_d}{2} \cdot (x+y)^d + \frac{a_0 - a_d}{2} \cdot (x-y)^d. \end{aligned}$$

In summary, we have shown that if $f \in D_2$ is homogeneous of even degree d, then it is a multiple of $(x+y)^d$, and if it is homogeneous of odd degree d, then it is a linear combination of $(x+y)^d$ and $(x-y)^d$. Since any polynomial is expressible as a sum of homogeneous components, this gives a basis for D_2: $\{1, x+y, x-y, (x+y)^2, (x+y)^3, (x-y)^3, \ldots\}$.

Of particular interest here are the polynomials $x - y, (x - y)^3, (x - y)^5, \ldots$, since they are not themselves symmetric. Exchanging x and y in these polynomials is equivalent to multiplying these polynomials by a factor of -1 (they are known as "alternating polynomials" for this reason). So we have found a family of polynomials that are not symmetric, but which have symmetric partial derivatives!

You are now likely tottering on the edge of your seat hoping that I'm about to tell you that the polynomial family $f_m(x, y) = (x - y)^{2m-1}$ has interesting and nontrivial circuit lower bounds arising from the fact that ∇f_m consists of symmetric polynomials, combined with Theorem 1. And believe me, I would gladly tell you that if I thought I could get away with it. Unfortunately, standards of peer review being what they are, I am forced to remind you that Theorem 1 only improves on the trivial $\Omega(\log d)$ lower bound on degree-d polynomials by a factor of n, the number of variables. If our number of variables is fixed, as in the case of ∇f_m for which $n = 2$, Theorem 1 gives only a constant factor improvement on the trivial lower bound. Since we are already so deeply invested in asymptotic thinking, a constant factor improvement doesn't really make much difference to us. So our search for lower bounds must take us beyond the $n = 2$ case, to the frightening world of polynomials in arbitrary numbers of variables.

Since alternating polynomials (polynomials for whom exchanging two variables is equivalent to multiplying by -1) were so useful in the $n = 2$ case, it is reasonable hope that they might come to our rescue again in the general case. That is, might it be the case that all alternating polynomials in n variables have symmetric partial derivatives? Regrettably not. The following is an alternating polynomial in $n = 3$ variables which provides a counterexample:

$$f(x, y, z) = (x - y)(x - z)(y - z) = x^2y - xy^2 - x^2z + xz^2 + y^2z - yz^2,$$

but

$$f_x = 2xy - y^2 - 2xz + z^2,$$

which is not symmetric. (Interestingly, f_x is alternating with respect to permutations that don't include x, and this trend holds in general, but that isn't helpful for us right now.)

The time constraints of the semester unfortunately kept me from as deep an exploration of the space D_n as I might have hoped for. However, some time spent searching in the $n = 3$ and arbitrary n cases has left me with the following conjecture, for which I have no certain proof or disproof:

Conjecture 1. *Let $n \geq 3$. Then*

$$\{1, x_1 + x_2 + \cdots + x_n, (x_1 + x_2 + \cdots + x_n)^2, (x_1 + x_2 + \cdots + x_n)^3, \ldots\}$$

is a basis for D_n.

Since the basis consists of the powers of the elementary symmetric polynomial e_1^n, Conjecture 1 would imply that for $n \geq 3$, $D_n \subseteq (e_1^n) \subseteq k[x_1, \ldots, x_n]^{S_n}$. That is, all polynomials in $n \geq 3$ variables whose partial derivatives are all symmetric are themselves symmetric. If true, this conjecture would be a death knell for our simple strategy of lower bounding polynomials with their partial derivatives, at least in its most basic form.

The remainder of the chapter will give some ideas for how to adapt this strategy beyond its most basic form, such that we might be able to lower bound functions based on the symmetry of their partial derivatives, even if Conjecture 1 turns out to be true.

5.3 What next?

Let us define two classes of polynomials that generalize D_n.

For $r \in [0, 1]$, let D_n^r be the set of polynomials in $X = x_1, \ldots, x_n$ such that for every $f \in D_n^r$, at least an r-fraction of the partial derivatives of f are symmetric. That is,

$$D_n^r = \{f \in k[X] \mid \exists A \subseteq [n] \text{ such that } |A| \geq rn \text{ and } f_{x_i} \in k[X]^{S_n} \text{ for all } i \in A\}.$$

D_n^1 is simply D_n, but for $r < 1$, D_n^r may possibly contain more polynomials than D_n. For fixed r (say $r = 1/2$), a polynomial family (f_n) with each $f_n \in D_n^{1/2}$ would then have gradients ∇f_n containing a linear number in n of symmetric polynomials. Such a polynomial family may have lower bounds arising from Theorem 1.

Another possible generalization comes from relaxing the invariant ring in question from the ring of symmetric polynomials to an arbitrary invariant ring. Fix a group action of a group G on k^n, and let D_n^G be the set of polynomials $f \in k[X]$ with all their partial derivatives in $k[X]^G$:

$$D_n^G = \{f \in k[X] \mid f_{x_i} \in k[X]^G \text{ for all } i \in [n]\}.$$

Then $D_n^{S_n} = D_n$ for the natural action of S_n on $[n]$. This class of polynomials is less relevant to Theorem 1 specifically, since that theorem deals specifically with $k[X]^{S_n}$. However, this is where the ideas discussed in Chapters 2 and 3 may become relevant again. Chapter 3 specifically exhibited a line of research into the inherent complexity of invariant rings, via circuit encodings of their generating sets. As more information is learned about the computational complexity of an invariant ring $k[X]^G$, the complexity of the class of polynomials D_n^G may then be better understood.

Since the partial derivative is at the heart of the an arithmetic circuit lower bound based on symmetry, it makes sense to ask questions about the interaction of the partial derivative with symmetries of group actions. Further study is needed to understand this interaction, and classes of polynomials like D_n, D_n^r, and D_n^G provide concrete objects at which to direct this study.

Chapter 6

Conclusion

If you have read all the way to this point, you may find yourself wondering what you have gained from your time thereupon spent. What unifying insights was I trying to impart to you in so many painstakingly chosen words? What, in short, was this all about? If my prose has been sufficiently skillful, the separate shards of this answer are by now safely lodged in your brain, but it is worth taking the time to arrange them into a complete, self-contained sculpture that you are now equipped to view in its entirety. You have seen the tail and the trunk and the ears, and now it is time to look upon the full elephant.

6.1 What we have learned

Uniting everything we have discussed are two concepts not unique to mathematics – symmetry and complexity. Invariant polynomials and invariant rings are defined by their symmetry – they remain the same when transformations (group actions) are applied. In other contexts, we may think of symmetry as a form of simplicity, but in the context of polynomials, it is paradoxically through properties symmetry that we are able to construct proofs of inherent complexity, as measured by arithmetic circuit lower bounds. In Chapter 3, symmetry under a particular transformation was exploited to show that an arithmetic circuit could solve the 3-matching problem, which (if that problem is as hard as is believed) shows that the arithmetic circuit must be complex. In Chapter 4, polynomials with symmetry under variable permutations were shown to have a definite structure, with partial derivatives taking a particular form, which then gave definite limits on how efficiently they could be computed.

In both cases, the symmetry of the polynomials makes it possible for mathematicians to prove their complexity, but it would be hasty to walk away with the conclusion that "symmetry causes complexity." This statement is intuitively backwards, and our intuition here is valuable and not worth abandoning just yet. In the world of Boolean circuits, it is known that a randomly chosen Boolean function family is overwhelmingly likely to require large circuits to compute. However, computer scientists struggle to write down a specific Boolean function family that requires large circuits, because if you can construct a function family through some kind of rational process, you can probably compute it with a small circuit as well. Humans are very bad at randomly sampling the space of Boolean function families. This dilemma is sometimes described as "searching for hay in a haystack" – you know there is hay all around you, but every time you reach into the stack, you draw out a needle.

It is not known whether similar facts are true for polynomials, since the proof method does not translate into the arithmetic realm, but we might expect that "random" polynomial families not defined by any special symmetry would be complex and hard to express with a small circuit, by virtue of their arbitrariness. The methods available to mathematicians for proving theorems are inherently biased towards nice polynomials with nice properties that can be exploited, and so it is natural that when we begin proving theorems about polynomial complexity, these are the first polynomials that yield to investigation. So although the idea of symmetry giving rise to provable complexity is an interesting one, the maxim of "symmetry causes complexity" is not and should not be accepted as a philosophy of how arithmetic circuits work.

With this word of caution in mind, however, we have seen in this paper that symmetry does provide a useful tool with which to analyze complexity. Analysis of invariant rings and polynomials may have the potential deepen our understanding of arithmetic complexity yet further. We will conclude by suggesting a few possible areas for such analysis.

6.2 Future work

We recklessly imply throughout Chapter 5 that the lower bounds shown for elementary symmetric polynomials will likely give rise to lower bounds for the more general class of symmetric polynomials. However, this is by no means a solved problem. In order to formalize the intuition that "symmetric polynomials have provable inherent complexity" that Chapter 5 assumes, it would be necessary to show that the lower bounds on elementary symmetric polynomials extend in some way to all symmetric polynomials. If this assumption is indeed true, the most likely directions toward proving it would be either to follow the footsteps of Baur and Strassen (1983) and analyze the algebraic geometry of the gradients of general symmetric polynomials, or to show that small-size circuits for general symmetric polynomials would somehow yield small-size circuits for elementary symmetric polynomials.

In Section 5.3, we give specific examples of not-yet-understood classes of polynomials whose partial derivatives are defined by symmetry. These classes seem natural as next steps to study after Bauer and Strassen's results, since those describe how complexity arises from symmetric polynomials and is preserved by partial derivatives. This is a good potential direction for exploration, especially for someone (like me, and perhaps you the reader as well) without a great deal of prior research experience in algebra or complexity theory, since it seems that an understanding of partial derivatives would be mostly sufficient. The goal of this work would be to find unconditional lower bounds on a wider range of polynomial families, and thereby to better understand arithmetic circuit complexity.

The work presented in Chapter 3 on the complexity of succinct encodings also has the potential for extension to new results. Section 1.5 of Garg et al. (2019) lists several open questions which are natural extensions of the work summarized in this paper, and we refer readers interested in the most natural specific extensions of this work (in the opinions of those who actually did it) to this list. These questions are aimed at understanding other problems related to invariant rings such as the orbit closure and null cone problems, which have implications for the complexity of the permanent polynomial, and fit into the larger goal of separating VP and VNP, perhaps the most fundamental goal of arithmetic complexity.

Arithmetic circuit complexity is an indirect path toward answering questions about the nature of computation, a path that is uniquely poised to take advantage of tools from algebra. We have seen that these tools, in the form of objects like group actions and algebraic varieties, can be applied to polynomials with the right kinds of symmetry. This area of study is exciting because it brings the very old results of algebra into conversation with the very new field of computational complexity theory. With so little known about the complexity of polynomials, it can be safely assured that though the history of algebra is long, its future is longer yet.

Bibliography

Agrawal, Manindra, and Somenath Biswas. 2003. Primality and identity testing via Chinese remaindering. *Journal of the ACM (JACM)* 50(4):429–443.

Arora, Sanjeev, and Boaz Barak. 2009. *Computational Complexity: A Modern Approach*. Cambridge University Press.

Baur, Walter, and Volker Strassen. 1983. The complexity of partial derivatives. *Theoretical computer science* 22(3):317–330.

Berkowitz, Stuart J. 1984. On computing the determinant in small parallel time using a small number of processors. *Information processing letters* 18(3):147–150.

Cox, David, John Little, and Donal O'Shea. 2013. *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Springer Science & Business Media.

Csanky, Laszlo. 1975. Fast parallel matrix inversion algorithms. In *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*, 11–12. IEEE.

Garg, Ankit, Christian Ikenmeyer, Visu Makam, Michael Walter, Rafael Oliveira, and Avi Wigderson. 2019. Search problems in algebraic complexity, GCT, and hardness of generator for invariant rings. *arXiv preprint arXiv:191001251* .

Hilbert, David. 1893. Über die vollen Invariantensysteme. *Mathematische Annalen* 42(3):313–373.

Impagliazzo, Russell, and Avi Wigderson. 1997. P = BPP if E requires exponential circuits: Derandomizing the XOR lemma. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, 220–229.

Kabanets, Valentine, and Russell Impagliazzo. 2004. Derandomizing polynomial identity tests means proving circuit lower bounds. *Computational Complexity* 13(1-2):1–46.

Karp, Richard M. 1972. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, 85–103. Springer.

Ladner, Richard E. 1975. The circuit value problem is log space complete for P. *ACM Sigact News* 7(1):18–20.

Mahajan, Meena. 2014. Algebraic complexity classes. In *Perspectives in Computational Complexity*, 51–75. Springer.

Mahajan, Meena, and V Vinay. 1997. A combinatorial algorithm for the determinant. In *SODA*, 730–738.

Malod, Guillaume, and Natacha Portier. 2008. Characterizing Valiant's algebraic complexity classes. *Journal of Complexity* 24(1):16–38.

Morgenstern, Jacques. 1985. How to compute fast a function and all its derivatives: A variation on the theorem of Baur-Strassen. *ACM SIGACT News* 16(4):60–62.

Mulmuley, Ketan. 2017. Geometric complexity theory V: Efficient algorithms for Noether normalization. *Journal of the American Mathematical Society* 30(1):225–309.

Noether, Emmy. 1915. Der Endlichkeitssatz der Invarianten endlicher Gruppen. *Mathematische Annalen* 77(1):89–92.

Schwartz, Jacob T. 1980. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM (JACM)* 27(4):701–717.

Strassen, Volker. 1973a. Die Berechnungskomplexität von elementarsymmetrischen Funktionen und von Interpolationskoeffizienten. *Numerische Mathematik* 20(3):238–251.

———. 1973b. Vermeidung von Divisionen. *Journal für die Reine und Angewandte Mathematik* 264:184–202.

Tolkien, J. R. R. 1954. *The Two Towers*. Allen and Unwin.

Valiant, Leslie G. 1979. Completeness classes in algebra. In *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing*, 249–261.

Weyl, Hermann. 1946. *The Classical Groups: Their Invariants and Representations*, vol. 45. Princeton University Press.

Wigderson, Avi. 2019. *Mathematics and Computation: A Theory Revolutionizing Technology and Science*. Princeton University Press.

Zippel, Richard. 1979. Probabilistic algorithms for sparse polynomials. In *International symposium on symbolic and algebraic manipulation*, 216–226. Springer.