Algebraic Complexity Theory

A quick tour

Mohannad Shehata

Contact:mohannad.shehata@mail.utoronto.ca

Undergrad Seminar Summer 2023

Table of Contents

Table of Contents

Prelimenaries

In computer science, we often analyze a program's efficiency by measuring the time it takes to execute a program on an input with respect to the size of the input. For example, incrementing the values in a list of length n takes time proportional to n. We begin by introducing a few bits of notation:

- We say f(n) is O(g(n)) if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ where c is a real number

Prelimenaries

In computer science, we often analyze a program's efficiency by measuring the time it takes to execute a program on an input with respect to the size of the input. For example, incrementing the values in a list of length n takes time proportional to n. We begin by introducing a few bits of notation:

- We say f(n) is O(g(n)) if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ where c is a real number
- Similarly, we say f(n) is Ω(g(n)) if $\lim_{n\to\infty} \frac{g(n)}{f(n)} = c$ where c is a real number

Prelimenaries

In computer science, we often analyze a program's efficiency by measuring the time it takes to execute a program on an input with respect to the size of the input. For example, incrementing the values in a list of length n takes time proportional to n. We begin by introducing a few bits of notation:

- We say f(n) is O(g(n)) if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ where c is a real number
- Similarly, we say f(n) is Ω(g(n)) if $\lim_{n\to\infty} \frac{g(n)}{f(n)} = c$ where c is a real number
- f(n) is o(g(n)) if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

Prelimenaries

In computer science, we often analyze a program's efficiency by measuring the time it takes to execute a program on an input with respect to the size of the input. For example, incrementing the values in a list of length n takes time proportional to n. We begin by introducing a few bits of notation:

- We say f(n) is O(g(n)) if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ where c is a real number
- Similarly, we say f(n) is Ω(g(n)) if $\lim_{n\to\infty} \frac{g(n)}{f(n)} = c$ where c is a real number
- f(n) is o(g(n)) if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
- f(n) is ω(g(n)) if $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$

Prelimenaries

In computer science, we often analyze a program's efficiency by measuring the time it takes to execute a program on an input with respect to the size of the input. For example, incrementing the values in a list of length n takes time proportional to n. We begin by introducing a few bits of notation:

- We say f(n) is O(g(n)) if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ where c is a real number
- Similarly, we say f(n) is Ω(g(n)) if $\lim_{n\to\infty} \frac{g(n)}{f(n)} = c$ where c is a real number
- f(n) is o(g(n)) if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
- f(n) is ω(g(n)) if $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$

1 and 3 are called Big O and little o notation respectively, similarly with 2 and 4(Omega)

- We will now introduce the notion of a decision problem: a language L is a nonempty set of strings, and a decision problem is the question "Is string x given as input in L or not?" Note that we can think of this string more abstractly depending on the problem, such as a graph when the decision problem involves a graph or a list when the problem involves a list since we can encode any of these objects as a string.

- We will now introduce the notion of a decision problem: a language L is a nonempty set of strings, and a decision problem is the question "Is string x given as input in L or not?" Note that we can think of this string more abstractly depending on the problem, such as a graph when the decision problem involves a graph or a list when the problem involves a list since we can encode any of these objects as a string.
- To solve them, we use the notion of a Turing Machine(TM), an abstract computer that has access to an infinite tape divided into segments with a pointer at a particular segment(each containing a single character). For each step, the TM reads whatever character on the tape the pointer points to, potentially modifies it, and then moves right or left on the tape and repeats the whole process all over again. It starts with input x already written on the tape and everything else blank

- We will now introduce the notion of a decision problem: a language L is a nonempty set of strings, and a decision problem is the question "Is string x given as input in L or not?" Note that we can think of this string more abstractly depending on the problem, such as a graph when the decision problem involves a graph or a list when the problem involves a list since we can encode any of these objects as a string.
- To solve them, we use the notion of a Turing Machine(TM), an abstract computer that has access to an infinite tape divided into segments with a pointer at a particular segment(each containing a single character). For each step, the TM reads whatever character on the tape the pointer points to, potentially modifies it, and then moves right or left on the tape and repeats the whole process all over again. It starts with input x already written on the tape and everything else blank
- We measure time by the number of times the TM moves its "pointer"

- A language(or problem) is said to be in P if there exists a TM that can decide whether any input x is in the language or not in P in time polynomial in the length n of x (ie time that lies in $O(n^c)$ for some constant c.)

- A language(or problem) is said to be in P if there exists a TM that can decide whether any input x is in the language or not in P in time polynomial in the length n of x (ie time that lies in $O(n^c)$ for some constant c.)
- An example is whether a list contains an element divisible by 5

- A language(or problem) is said to be in P if there exists a TM that can decide whether any input x is in the language or not in P in time polynomial in the length n of x (ie time that lies in $O(n^c)$ for some constant c.)
- An example is whether a list contains an element divisible by 5
- Similarly, a language is in NP if there exists a TM that can decide whether any input x is in the language or not in time polynomial in the length n of x when given a correct certificate c, which is a string that has size polynomial in n that aids in solving a problem.

- A language(or problem) is said to be in P if there exists a TM that can decide whether any input x is in the language or not in P in time polynomial in the length n of x (ie time that lies in $O(n^c)$ for some constant c.)
- An example is whether a list contains an element divisible by 5
- Similarly, a language is in NP if there exists a TM that can decide whether any input x is in the language or not in time polynomial in the length n of x when given a correct certificate c, which is a string that has size polynomial in n that aids in solving a problem.
- An example would be whether a set S of nonnegative integers has a subset whose sum of all its elements is a particular number n, a certificate here would be such a subset of S with the desired property

- A language(or problem) is said to be in P if there exists a TM that can decide whether any input x is in the language or not in P in time polynomial in the length n of x (ie time that lies in $O(n^c)$ for some constant c.)
- An example is whether a list contains an element divisible by 5
- Similarly, a language is in NP if there exists a TM that can decide whether any input x is in the language or not in time polynomial in the length n of x when given a correct certificate c, which is a string that has size polynomial in n that aids in solving a problem.
- An example would be whether a set S of nonnegative integers has a subset whose sum of all its elements is a particular number n, a certificate here would be such a subset of S with the desired property
- We know any language in P is in NP since we have a machine that can solve without a certificate(ie we can just discard it). Is every language in NP also in P? This is called the P vs NP problem, and most computer scientists believe $NP \neq P$, meaning there is some language in NP but not in P

Table of Contents

Motivation

We are not nearly close to finding a solution to the P vs NP problem. One approach is to find a problem in NP such that for all TMs solving it, the time is in $\omega(n^c)$ for all constants c;in other words, the time for such TM grows faster than any polynomial, which we will refer to as superpolynomial time, and thus the language corresponding to such a problem cannot be in P. A bound on how fast a problem can be feasibly solved is called a lower bound , and we don't know a lot of those, partly due to how difficult it is to analyze the operation of TMs. Therefore we study arithmetic circuits to shed more light on the P vs NP problem, and as we can see later there are a few useful techniques to come up with lower bounds in the circuit world. Arithmetic circuits are also a natural way to analyze the complexity of computing polynomials.

Table of Contents

Definitions

- An arithmetic circuit over a field F is a directed acyclic graph where all gates of in-degree 0 are either variables or Field constants(like 0,1)

Definitions

- An arithmetic circuit over a field F is a directed acyclic graph where all gates of in-degree 0 are either variables or Field constants(like 0,1)
- A gate(node in the graph) is either labeled by a variable, field constant, or operation, either addition or multiplication, which we refer to as sum or product gates respectively. Input gates are those with in-degree(fan-in) 0, and those are labeled with a variable or field constant. Output gates are those with out-degree(fan-out) 0, representing the end result(s) of a computation. Usually we restrict fan-in to be at most 2.

Definitions

- An arithmetic circuit over a field F is a directed acyclic graph where all gates of in-degree 0 are either variables or Field constants(like 0,1)
- A gate(node in the graph) is either labeled by a variable, field constant, or operation, either addition or multiplication, which we refer to as sum or product gates respectively. Input gates are those with in-degree(fan-in) 0, and those are labeled with a variable or field constant. Output gates are those with out-degree(fan-out) 0, representing the end result(s) of a computation. Usually we restrict fan-in to be at most 2.
- The size of a circuit is the number of gates it has.

Definitions

- An arithmetic circuit over a field F is a directed acyclic graph where all gates of in-degree 0 are either variables or Field constants(like 0,1)
- A gate(node in the graph) is either labeled by a variable, field constant, or operation, either addition or multiplication, which we refer to as sum or product gates respectively. Input gates are those with in-degree(fan-in) 0, and those are labeled with a variable or field constant. Output gates are those with out-degree(fan-out) 0, representing the end result(s) of a computation. Usually we restrict fan-in to be at most 2.
- The size of a circuit is the number of gates it has.
- The depth of a gate is the maximum length of a path starting at the gate, while the depth of the circuit is the maximum depth over all gates in the circuit.

Definitions

- An arithmetic circuit over a field F is a directed acyclic graph where all gates of in-degree 0 are either variables or Field constants(like 0,1)
- A gate(node in the graph) is either labeled by a variable, field constant, or operation, either addition or multiplication, which we refer to as sum or product gates respectively. Input gates are those with in-degree(fan-in) 0, and those are labeled with a variable or field constant. Output gates are those with out-degree(fan-out) 0, representing the end result(s) of a computation. Usually we restrict fan-in to be at most 2.
- The size of a circuit is the number of gates it has.
- The depth of a gate is the maximum length of a path starting at the gate, while the depth of the circuit is the maximum depth over all gates in the circuit.
- Both size and depth are involved in analyzing complexity.

Definitions

- A family of Polynomials $\{f_n\}$ is said to be in VP if there exists a polynomial t(n) such that the number of variables and degree of $\{f_n\}$ are bounded above by t(n), and there exists a circuit of size at most t(n) computing $\{f_n\}$.

Definitions

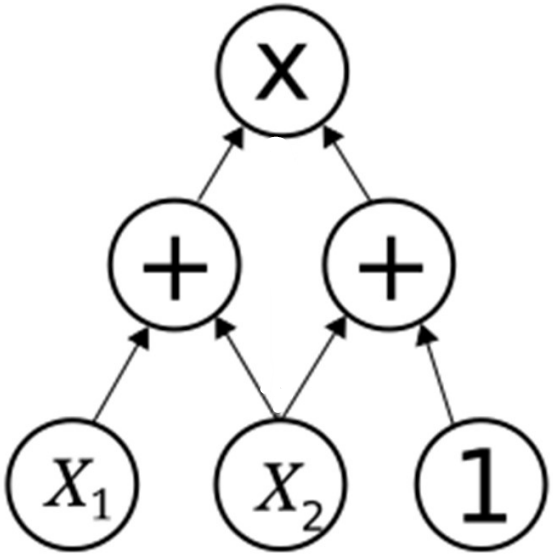
- A family of Polynomials $\{f_n\}$ is said to be in VP if there exists a polynomial t(n) such that the number of variables and degree of $\{f_n\}$ are bounded above by t(n), and there exists a circuit of size at most t(n) computing $\{f_n\}$.
- An famous example is the Determinant polynomial: Take an n*n matrix where each entry contains a different variable and find the determinant of that. Recall that each element in the determinant has a "signature",either - or $+$, corresponding to the entries picked from the matrix for a particular monomial.

Definitions

- A family of Polynomials $\{f_n\}$ is said to be in VP if there exists a polynomial t(n) such that the number of variables and degree of $\{f_n\}$ are bounded above by t(n), and there exists a circuit of size at most t(n) computing $\{f_n\}$.
- An famous example is the Determinant polynomial: Take an n*n matrix where each entry contains a different variable and find the determinant of that. Recall that each element in the determinant has a "signature",either - or +, corresponding to the entries picked from the matrix for a particular monomial.
- A family of Polynomials $\{f_n\}$ is said to be in VNP if there exists a polynomial t(n) such that the number of variables and degree of $\{f_n\}$ are bounded above by t(n), and for $f(x_1, x_2, ...x_n)$, there exists a polynomial $g(x_1, x_2, .., x_n, y_1, y_2, ...y_m)$ where m is at most t(n), the family of g's is in VP, and $f(x_1, x_2, ...x_n) = \Sigma_{a_1,a_2,..a_m \in \{0,1\}^m} g(x_1, x_2, ..x_n, a_1, a_2, ...a_m)$ Think of the extra variables as a "certificate" just like in NP.

- An example of a polynomial in VNP that is theorized not to be in VP is the Permanent, which is defined exactly as the determinant with the "signature" of each element being always positive.

- An example of a polynomial in VNP that is theorized not to be in VP is the Permanent, which is defined exactly as the determinant with the "signature" of each element being always positive.
- These serve as the analogues to P and NP in this model, and we made more progress in trying to show $VP \neq VNP$ than we did in P vs NP as we will see in a bit.



Sample circuit computing $x_1x_2+x_2^2+x_1+x_2$

Table of Contents

Interesting Concepts

- A homogenous polynomial is one whose terms are all of the same degree, such as the determinant polynomial. Similarly a homogenous circuit is one where the polynomial computed by each gate is homogenous.

Interesting Concepts

- A homogenous polynomial is one whose terms are all of the same degree, such as the determinant polynomial. Similarly a homogenous circuit is one where the polynomial computed by each gate is homogenous.
- If we have a non homogenous circuit of size s computing a homogenous polynomial, we can use it to construct a homogenous circuit of size $O(r^2 s)$ computing the same polynomial where r is the degree. This immediately implies if we have a nonhomogenous circuit computing a homogenous polynomial of polynomial size, we can construct a homogenous circuit of polynomial size computing the same polynomial, meaning we can just focus on analyzing general circuits for the purposes of polynomiality without worrying whether they are homogenous.

- An alternative model would be to consider bounding the depth of the circuit and making the fan-in unbounded(except possibly at the first level from the bottom). The models we usually examine have alternating layers of sum and product gates, with the most common being $\Sigma\Pi$, $\Sigma\Pi\Sigma$, and $\Sigma\Pi\Sigma\Pi$ which have maximum depth of 2,3, or 4 respectively. Having an superscript k on either Π or Σ signifies the restriction of fan-in of gates in that layer to at most k. Many results, often referred to as depth reduction results, try to examine the relation between general circuits and those with bounded depth.

A sample $\Sigma\Pi\Sigma$ circuit

Figure 2.. A depth three circuit.

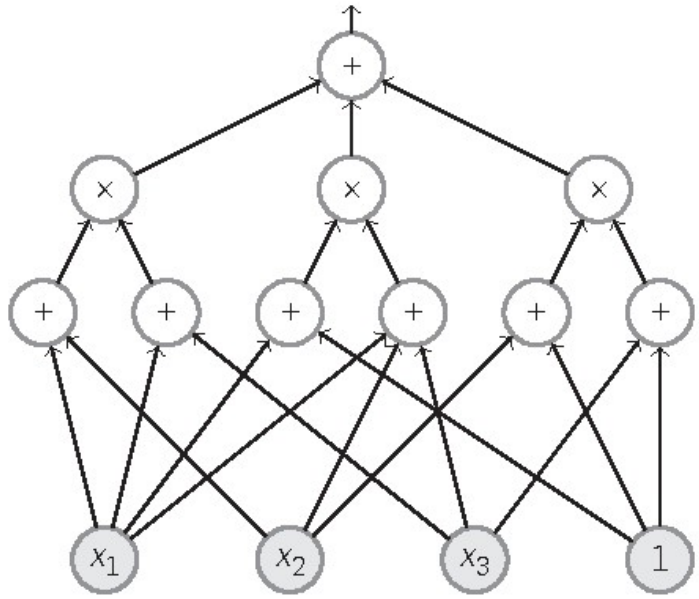


Table of Contents

Combinatorial Techniques

Consider $\Sigma\Pi$ circuits. We demonstrate an example of a family of polynomials that is "hard" for this model, meaning no polynomial size circuits exist.

Combinatorial Techniques

Consider $\Sigma\Pi$ circuits. We demonstrate an example of a family of polynomials that is "hard" for this model, meaning no polynomial size circuits exist. Let $f(x_1, x_2, ...x_n) = (x_1 + x_2 + ... + x_n)^n$ be our family of polynomials where n ranges over the natural numbers. Since $\Sigma\Pi$ circuits have a sum layer followed by a product layer, it can only compute sums of products, so the only way to compute a polynomial is by directly computing each of its monomials and adding the results, meaning at best a circuit computing f has as many multiplication gates as there are monomials in f.

Combinatorial Techniques

Since the degree of all monomials is n, letting $y_1, y_2, ...y_n$ represent the powers of $x_1, x_2, ...x_n$ in a monomial, we get that the number of unique monomials is equal to the number of solutions to $y_1 + y_2 + ...y_n = n$ where each y_i is an integer at least 0, which by the stars and bars method we know is $\binom{n+n-1}{n-1}$ which is in $\Omega(2^n)$, meaning no circuit of size polynomial in n exists for this family of polynomials since we must have at least $\Omega(2^n)$ product gates.

Table of Contents

Partial Derivative Complexity Measure

To show that a certain family of polynomials is "hard" for a certain model, meaning no polynomially sized circuit computes the polynomial, we need to exhibit a complexity measure, a property that has a relatively low value for polynomially sized circuits but a high value for the family of polynomials, demonstrating that finding such a polynomial sized circuit is impossible.

Sym_n^d family of polynomials

Let S=$\{x_1, x_2, ...x_n\}$ be our set of variables. Consider all subsets of d variables of n, and for each subset construct a monomial that is the product of all its variables. Then our desired polynomial is the sum of all such monomials constructed from such subsets.

Examples

$Sym_4^3 = x_1x_2x_3 + x_2x_3x_4 + x_1x_2x_4 + x_1x_3x_4$

Partial Derivative Complexity Measure

Definition of the measure μ_k

Consider all k-th order partial derivatives of a polynomial f. Then μ_k is the dimension of the span of all these partial derivatives.

It is immediate from the derivative rules for addition and the definition of dimension that $\mu_k(f+g) \leq \mu_k(f) + \mu_k(g)$, we call a measure with this property subadditive. We now upper bound m_k of an arbitrary depth-3 homogenous circuit that can compute Sym_n^d. Let f be the polynomial computed by such a circuit. f= $M_1 + M_2 + ...M_T$ where T is the number of multiplication gates. Each $M_1 = L_1 \cdot L_2 \cdot ..L_d$ where each L_i is a sum of variables. The derivative of each L_i with respect to a variable is either a constant or 0, so the kth order derivatives lie in the span of the set containing all products defined by taking a subset of the $L_i's$ and multiplying them. Since the number of subsets is 2^d, $\mu_k(M_j) \leq 2^d$, meaning $\mu_k(f) \leq T * 2^d$.

Partial Derivative Complexity Measure

To prove that Sym_n^d is hard in this model, we need the following lemma

Lemma

If k=$\frac{d}{2}$, then Sym_n^d has a μ_k of at least $\binom{n}{0.5d}$

If we let $d = \frac{n}{8}$, using the lemma we get that if a homogenous depth-3 circuit computes our desired polynomial it must be the case $T * 2^d \geq \binom{n}{0.5d}$, meaning $T \geq \binom{n}{\frac{n}{16}} * 2^{-\frac{n}{8}} \geq 16^{\frac{n}{16}} * 2^{-\frac{n}{8}} \geq 2^{\frac{n}{8}}$. Recall that T is the number of multiplication gates in the circuit, so the size of a homogneous depth-3 circuit computing the desired polynomial is bounded below by $2^{\Omega(n)}$ which grows faster than any polynomial.

Table of Contents

VP vs VNP

We illustrate two major theorems that shed light on how close we are in solving the VP vs VNP problem:

Tavenas 13

Any (homogeneous) poly(n) sized arithmetic circuit can be depth reduced to an equivalent (homogeneous) $\Sigma\Pi\Sigma\Pi^{\sqrt{n}}$ circuit of size $2^{O(\sqrt{n}\log n)}$

Kayal-Saha-Saptharishi 13

There is a explicit polynomial in VNP that requires homogenous circuits of size $\Sigma\Pi\Sigma\Pi^{\sqrt{n}}$ circuit of size $2^{\Omega(\sqrt{n}\log n)}$

By the former theorem, it suffices to show that to separate VP from VNP, we need to exhibit a polynomial family that needs a $\Sigma\Pi\Sigma\Pi^{\sqrt{n}}$ circuit of size $2^{\omega(\sqrt{n}\log n)}$ to compute it, but by the latter we have one with size $2^{\Omega(\sqrt{n}\log n)}$, any better than this and we prove $VP \neq VNP$!

Discussion Time!

Thank you for your attention! Feel free to ask questions.