Lecture 1: Computational Models, Branching Programs and Circuits

Anup Rao

September 27, 2023

Our goal in this course is to mathematically capture the concept of computation. In the broadest sense, what is computation? What is the best way to model computation? How can we reason about whether something can be efficiently computed or not? What are the resources that are worth measuring with regards to computational efficiency? Complexity theory is about addressing these questions.

A first attempt at defining a computation might be to say that it is a process that *manipulates information* in some way, or has an *input-output* behavior. For example, when reading this sentence, your brain takes the information encoded graphically and translates that information into letter, words and ideas, and so performs a computation.

At this point you may feel that we have made the model too general to be useful, so it might be useful to explain what is *not* computation. A useful mathematical abstraction that captures some of the things we have discussed is the abstraction of *functions*. Given two sets D, R, a function

$$f : D \to R$$

assigns a value $f(x) \in R$ to every element of $x \in D$. So, if we think of D as the set of all possible images, and R as the set of all sentences, f can be defined to be the function that maps the picture of a sentence to the actual sentence. This abstraction misses something that is inherent about physical computational processes: computations are local. At any point, the state of two parts of the brain that are far away from each other cannot affect each other.

Informally, a computational process is a process that manipulates information in some *local* or restricted way. Interesting computational processes are ones that manage to have a complicated global effect through incremental local steps. There are many such computational processes, and we will not be able to talk about all of them in detail in this course. The aim of this course is to show you, in broad strokes, how you can start to reason about such computational processes and their complexity.

Computational Complexity

How can we distinguish functions that are easy to compute from functions that are hard? What makes some things easy and other things hard?

In order to tackle this kind of question, we first need mathematical models that captures exactly what the process we are interested in can do cheaply, and what takes more effort. We would like our models to be general enough that they capture most real computational processes, and simple enough that we can ask and understand easy questions about them.

For example, we are very good at reading text, but multiplying 100 digit numbers takes us considerably more time, even though the amount of information is contained in a picture is much more than the information contained in a 100 digit number.

A crucial issue is how the information being manipulated is encoded. For example, if numbers are encoded using their prime factorization (both in the input and output), then it is slightly easier for us to multiply two 100 digit numbers than if they are encoded using their digits. Under this representation, addition of numbers, and comparing two numbers becomes harder.

Addition is hard when numbers are represented in terms of their factorization because we would have to factor the sum to bring it back in this representation. We do not know of any efficient algorithms for factoring numbers.

So, given a function $f : D \to R$, we would like to be able to quantify how difficult it is to compute this function. We shall make two immediate simplifications.

- We shall identify the the input domain with the set of binary strings of arbitrary length $\{0,1\}^*$, or the set of strings of some fixed length $\{0,1\}^n$. Since every countable set can be mapped to the first set, and every finite set can be mapped to the second, this does not lose too much generality.
- We shall often restrict the output domain to be $R = \{0,1\}$. This does lose some generality, but it will turn out that most of our ideas will easily translate to the situation when the output domain is bigger. Further, for most examples of functions to bigger domains that are hard to compute, we shall be able to easily find corresponding boolean functions that are hard to compute.

Next we give several examples of computational models, and discuss some strengths and weaknesses of each of them.

Finite Automata

A finite automaton can be used to compute functions $f : \{0,1\}^* \to \{0,1\}$. An example is shown in Figure 1. One starts at the start state (labeled S), and reads the input bit by bit, transitioning on the states. The output is 1 if and only if we ever hit the accept state (labeled A).

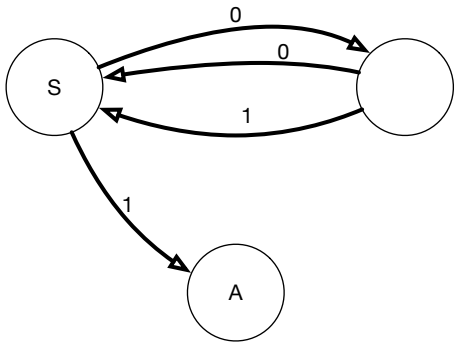

Figure 1: A finite state automaton that accepts strings that have a 1 in some odd location.

The weaknesses of this model:

- The set of functions $f : \{0,1\}^* \to \{0,1\}$ that can be computed by any finite automaton is not very general. Even more damning—finite automata cannot compute functions that we can efficiently compute in practice.

Only functions that represent *regular* languages can be computed by a finite automaton.
Try to prove that no finite automaton can compute the function whose output is 1 if and only if the input is a palindrome.

- It does not give a way to measure the complexity of computing functions. We can count the number of states in the automaton, but since this is always a constant independent of n, this measure of complexity doesn't scale with the input size, and so does not map to what we are trying to capture — the level of difficulty involved in carryout the computation.

Finite automata are a *uniform* model of computation. This means that there is a single description of the process that can be used to carry out the computation no matter how long the length of the input is.

Branching Programs

Branching programs are very similar to finiate automata. This is a model that computes functions $f : \{0,1\}^n \to \{0,1\}$.

A branching program is a directed acyclic graph where every vertex is labeled by either a variable or an output value. Every vertex that is labeled by a variable has exactly two edges coming out of it, one labeled 1 and the other labeled 0. The vertices are partitioned into disjoint sets $L_0, \ldots, L_k$, and all edges go from some set L_i to L_{i+1}. There is a special designated start node in the graph. To carry out the computation, we start at the start state, and follow the indicated path

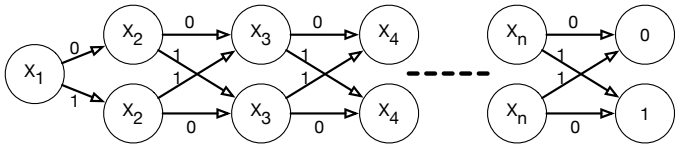

Figure 2: A branching program that computes whether or not the number of 1's in the input is even.

by reading the variable that labels the state that we are on in each step. When we hit a node labeled by an output value, the output of the computation is that output value.

There are two measures of the complexity of the program. The *width* is the size of the largest layer. The length is the number of layers.

This model has the advantage that it can actually compute all functions:

Fact 1. *Every function* $f : \{0,1\}^n \to \{0,1\}$ *can be computed by a branching program with width* 2^n*, and length* $n+1$*.*

Proof Consider the branching program that is a rooted tree, where every node at level i reads the variable x_i. The program simply remembers the entire input. Each of the 2^n inputs x gets mapped to a distinct leaf, so that leaf can be labeled by the value $f(x)$ to compute f. ■

In class, we discussed how every finite automaton for a function $f : \{0,1\}^* \to \{0,1\}$ can be used to get a branching program of size $O(n)$ that computes f on all the inputs of length n.

Branching programs will be used later in the course as a way to capture computations that use a small amount of space (or memory). The main drawback of branching programs is that there are algorithms which run very quickly on computers in practice that we don't know how to model as small branching programs. So, the branching program complexity doesn't seem to capture everything we want to capture about efficient computation.

Branching programs define a *non-uniform* model of computation. The program only tells you how carry out computation on an input of length n. To talk about the asymptotic complexity of computing a function $f : \{0,1\}^* \to \{0,1\}$ that is defined on strings of all lengths, we need to talk about families of branching programs, and discuss the complexity of the programs as n gets larger and larger.

Boolean Circuits

A *boolean circuit* computing a function $f : \{0,1\}^n \to \{0,1\}$ is a directed acyclic graph with the following properties. Every vertex (also called a gate) has at most 2 edges coming in to it. If there are 0

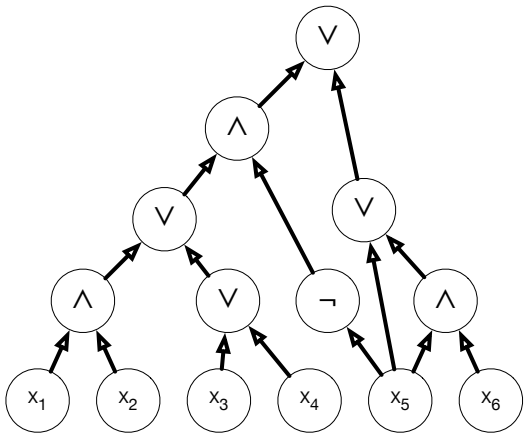

Figure 3: An example of a boolean circuit.

edges coming in, then the vertex is labeled with an input variable x_i, or the constants 0 or 1. Otherwise, the vertex is labeled with one of the boolean operators $\wedge, \vee, \neg$, and computes the specified operation on the bits that come in along the incoming edges. One of the gates in the circuit is designated the output node. This is the node whose value is the output of the circuit.

When every gate has out-degree at most 1, the circuit is called a *formula*. In the case of a formula, the graph of the circuit looks like a tree after edges have been converted into undirected edges.

A circuit can also be viewed as a program in a simple programming language, where every line is an assignment. For example, the circuit in Figure 3 is equivalent to this program:

1. $y_1 = x_1 \wedge x_2$
2. $y_2 = x_3 \vee x_4$
3. $y_3 = \neg x_5$
4. $y_4 = x_5 \wedge x_6$
5. $y_5 = y_1 \vee y_2$
6. $y_6 = x_5 \vee y_4$
7. $y_7 = y_5 \wedge y_3$
8. $y_8 = y_7 \vee y_6$

There are two major quantities we can measure to capture the complexity of a circuit:

Definition 2. *The* size *of the circuit is the number of gates in the circuit.*

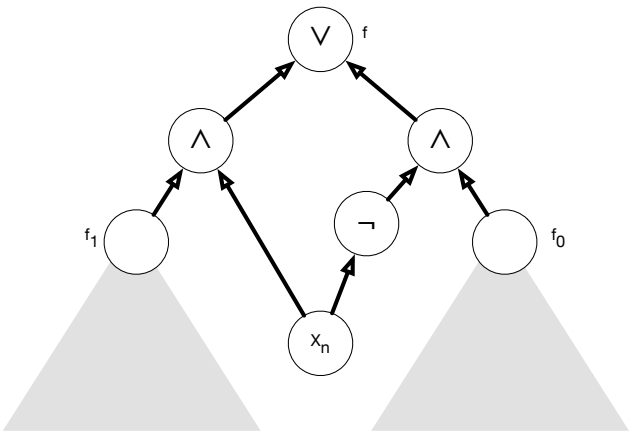

Figure 4: Recursive construction of a circuit for f.

Since every gate in the circuit has at most 2 incoming edges, the size of the circuit is proportional to the number of edges in the graph that defines the circuit:

Fact 3. *The size of the circuit is the same as the number of edges in the circuit, up to a factor of* 2.

We can also measure the *depth* of the circuit:

Definition 4. *The* depth *of the circuit is the length of the longest input to output path.*

The depth complexity is a measure of how much parallel time it takes to compute the function.

Just like branching programs, boolean circuits can complete *every* function:

Theorem 5. *Every function* $f : \{0,1\}^n \to \{0,1\}$ *can be computed by a circuit of size at most* $O(2^n)$.

Proof We construct the circuit recursively. When $n = 1$, there is clearly a constant sized circuit that computes f, since f must be either a constant, x_1 or $\neg x_1$.

For $n > 1$, let f_0 denote the function on $n-1$ bits given by $f_0(x) = f(x,0)$, and $f_1(x) = f(x,1)$. Then by induction we can compute f_0, f_1 recursively, and combine them using the value of the last bit to obtain f, as in Figure 4. When $x_n = 1$, the circuit outputs $f_1(x_1, \ldots, x_{n-1})$, and when $x_n = 0$, the circuit outputs $f_0(x_1, \ldots, x_{n-1})$.

If S_n is the size of the resulting circuit when the underlying function takes an n bit input, we have proved that

$$S_n \leq 2S_{n-1} + 5.$$

Expanding this recurrence, and using the fact that $S_1 \leq 5$, we get

that

$$S_n \leq \sum_{i=1}^{n} 2^i 5 = 5 \cdot (2^{n+1} - 1) < 10 \cdot 2^n,$$

where here we used the formula for computing the sum of a geometric series. ■

The above theorem is not the best result we know about this subject. In fact, we know:

Theorem 6. *Every function $f : \{0,1\}^n \to \{0,1\}$ can be computed by a circuit of size at most $O(2^n/n)$.*

You will be asked to prove this on your homework.

Some interesting relationships between Branching Programs and Circuits

Natural models often have unexpected connections between them. Here we explore one such unexpected connection between branching programs and circuits, that was discovered by Barrington. Barrington showed:

Theorem 7. *If $f : \{0,1\}^n \to \{0,1\}$ can be computed by a circuit of depth d, then it can be computed by a branching program of width 5 and length $O(4^d)$.*

The theorem is not known to hold with width 4.

This is a really powerful statement. It is especially useful if you want to prove lower bounds — if you want to show that a function cannot be computed in small depth, you can try to prove that the function cannot be computed using a small width branching program of small length. It is much easier to show the converse of the theorem:

Theorem 8. *If $f : \{0,1\}^n \to \{0,1\}$ can be computed by a branching program of width $O(1)$ and length 2^d, then it can be computed by a circuit of depth $O(d)$.*

Sketch of Proof Every width w branching program can be thought of as computing a function $g_x : [w] \to [w]$, where x is the input to the program. We shall prove inductively that you can compute the function g_x in depth Cd, for some large constant C.

The idea is to break up the program into the first half of the program, which computes h_x, and the second half, which computes q_x. Then $g_x = h_x \circ q_x$ is composition of these two functions. We recursively compute h_x and q_x. This computation should take depth $C(d-1)$. Then we use a constant number of gates to compute g_x from the descriptions of the two functions. Since the width is just a

constant, this takes depth C for some constant C. Our final depth is $C(d-1)+C = C(d)$. ■

Now, let us turn to proving Theorem 7.

Proof We are given a circuit of depth d computing f and need to compute the same function using a width 5 branching program. We shall restrict our attention to width 5 branching programs that compute permutations of $[5] = \{1,2,3,4,5\}$. Before we give the construction, we need to describe some nice properties of *cyclic* permutations.

A cyclic permutation is a permutation π with the property that if you start at 1, and keep applying the permutation, you eventually visit all elements of $[5]$. For example, the permutation shown in Figure 5 are cyclic. Here are some nice properties of cyclic permutations. These are all easy to verify, but we leave it as an exercise to do it:

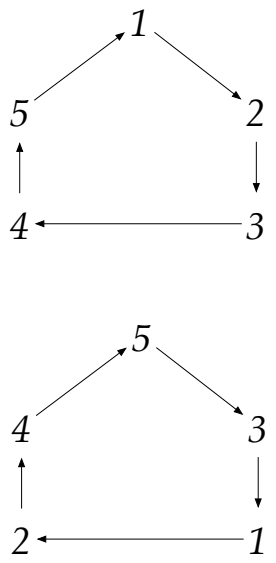

Figure 5: Two cyclic permutations.

- If π is cyclic, then so is π^{-1}.
- There are two cyclic permutations of $[5]$, π, σ with the property that $\pi\sigma\pi^{-1}\sigma^{-1}$ is another cyclic permutation. This will be called the *commutator property* below. For example, set $\pi = (12345), \sigma = (13542)$, and then the composition is (13254).
- For any two cyclic permutations π, σ, there is a permutation τ (not necessarily cyclic), such that $\tau\pi\tau^{-1} = \sigma$. This we be called conjugation.

Now, for the purpose of carrying out the proof, we shall design a branching progam that on input x computes a permutation π_x, such that if $f(x) = 0$, then π_x is the identity permutation, but if $f(x) = 1$, then π_x is a fixed cyclic permutation, say $\gamma = (12345)$. This branching program computes $f(x)$.

Suppose we have already made a program computing π_x that represents $g(x)$, and we want to compute $\neg g(x)$. To do this, we simply add a layer that computes γ^{-1}. The new program computes $\pi_x\gamma^{-1}$. Call the new program σ_x. If $g(x)$ is 0, $\sigma_x = \gamma^{-1}$, and if $g(x) = 1$, σ_x is the identity permutation. Now, by conjugation, there is another permuation τ such that $\tau\gamma^{-1}\tau^{-1} = \gamma$. We apply two more layers to implement this, and so recover the program that corresponds to $\neg g(x)$.

Suppose the final gate of the circuit is a $\wedge$ gate. So, the final output is $f(x) = g(x) \wedge h(x)$. Then, by induction we have two programs, one computing π_x that corresponds to $g(x)$, and the other computing σ_x that corresponds to $h(x)$. After doing some conjugation, we can ensure that if $g(x) = h(x) = 1$, then π_x, σ_x satisfy the commutator property. If either of them is the identity, then we have $\pi_x\sigma_x\pi_x^{-1}\sigma_x^{-1}$ is also the identity. So, we get that $\pi_x\sigma_x\pi_x^{-1}\sigma_x^{-1}$ is cyclic if and only

if $f(x) = 1$. Applying another conjugation gives us back the final program.

Gates that compute $\vee$ can be handled using the above methods, since $g(x) \vee h(x) = \neg(\neg g(x) \wedge \neg h(x))$.

We see that the length of the program generated in the above process satisfies $\ell_d \leq 4\ell_{d-1} + O(1)$. The solution to this recurrence is $\ell_d \leq O(4^d)$. ■