Lecture 1: Computational Models

Anup Rao

Semptember 27, 2018

In this first lecture, we discuss what computation is, and see a few examples of computational models.

Our goal in this course is to mathematically capture the concept of computation. A program is certainly a recipe for carrying out a computation, but is this the only type of computation that makes sense?

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 argue 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.

One can think of the weather as a computational process: given the current state of clouds, ocean currents and many other factors, the laws of the universe produce an outcome that uses the information about the current state to generate a new state. But is there any benefit to thinking about the weather this way?

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 computational processes that are doing something complicated from processes that are doing something

simple. 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).

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—

Only functions that represent *regular* languages can be computed by a finite automaton.

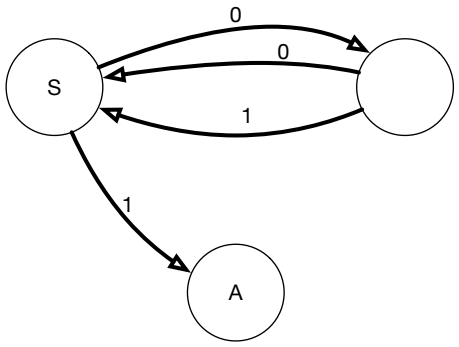

Figure 1: A finite state automaton that accepts strings that have a 1 in some odd location.

finite automata cannot compute functions that we can efficiently compute in practice.

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 is not very meaningful.

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. 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 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.

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

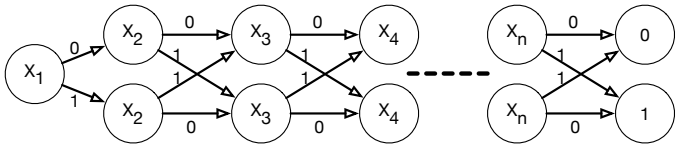

Figure 2: A branching program that computes whether or not the number of 1's in the input is even.

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

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.

Moreover, it comes with a meaningful measure of complexity: we can count the number of nodes in the branching program and use that as a measure of how complex the computation is. 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.

Linear Programs

WE DID NOT DISCUSS LINEAR PROGRAMS IN CLASS. Nevertheless, many of you may encounter linear programs in other classes, so I will discuss them here in the notes.

Linear programs are an extremely popular way to define algorithms in practice. Given an n bit input x, a linear program defines an $m \times k$ matrix A_x, a $m \times 1$ column vector b_x and a $1 \times k$ vector c. The

output of the program is the solution to:

$$\begin{aligned} &\text{minimize} && c_x y \\ &\text{subject to} && A_x y \leq b_x, \end{aligned}$$

where here $A_x y \leq b_x$ asserts that every coordinate of $A_x y$ is at most b_x.

For example, suppose the input x is a directed graph on n vertices, and we wish to compute the length of the shortest path from s to t. For every potential edge $e = (u, v)$, setting

$$x_e = \begin{cases} 1 & \text{if } e \text{ is an edge in the graph,} \\ n & \text{otherwise,} \end{cases}$$

one can show that the output of the following linear program (with y_e being a variable for each edge e) is the length of the shortest path from s to t:

$$\begin{aligned} \text{minimize} \quad & \sum_e x_e \cdot y_e \\ \text{subject to} \quad & \sum_{e \text{ out of } s} y_e \geq 1, \\ & \sum_{e \text{ into } t} y_e \geq 1, \\ & \sum_{e \text{ into } v} y_e = \sum_{e \text{ out of } v} y_e, && \text{for every } v \neq s, t \\ & y_e \geq 0, && \text{for every } v \neq s, t \end{aligned}$$

The complexity of the program is the number of equations used to define it. There is something we left out of this model: we gave no way to measure the complexity of computing A_x, b_x, c_x from x. If we allow arbitrarily complex ways to get to A_x, b_x, c_x from x, you can compute *any* function with this model: just set $c_x = f(x)$, and $A_x = b_x = 0$ to compute $f(x)$.

It is known that linear programs *cannot* be used to compute whether or not a graph has a perfect matching, which is an algorithmic problem that we do know how to solve efficiently, if A_x, b_x are not allowed to depending on x.

Communication Complexity

Communication complexity has been a very useful model for proving lower bounds. In this model, there are two parties Alice and Bob. Alice is given an n-bit string x, and Bob is given an n-bit string y. In order to compute a function $f(x, y)$, they exchange messages about their inputs.

So, Alice sends Bob a message $m_1(x)$. Bob responds with a message $m_2(y, m_1)$. In this way, they exchange messages until someone

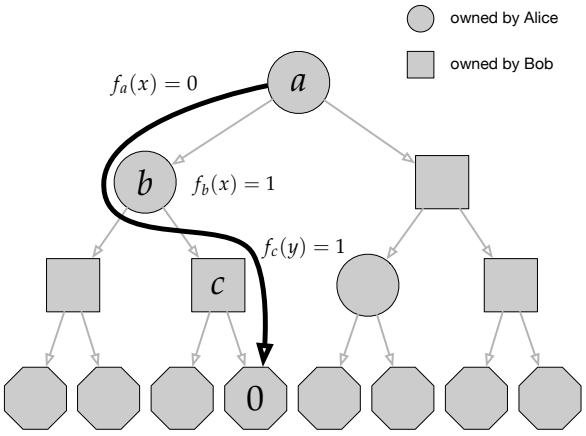

Figure 3: An execution of a protocol.

knows the value of f. The communication complexity of f is the minimum number of bits that must be communicated before the players know the value of f.

To formally define a communication protocol, we use a protocol tree. See Figure 3. This is a rooted binary tree where every node is associated with one of the players, and associated with a boolean function. To execute the protocol, Alice and Bob start at the root of the tree. If the root is owen by Alice, she evaluates the function associated with the root on her input and sends the result to Bob. The result of the evaluation determines which of the two children the protocol moves to next. In this way, the players reach a leaf of the tree, which is labeled with the output of the computation. The cost of the tree is simply its depth, which determines the maximum number of bits that may be exchanged during any execution of the protocol.

The main drawback of this model is that it is not very practical: just because a function has an efficient communication protocol doesn't mean that we can compute it efficiently in practice. For example, in this model, one can compute any boolean function of x with 1 bit of communication. However most functions cannot be computed efficiently in practice.

The main advantage of this model is that almost every other computational model seems to involve communication. So lower bounds on the communication complexity of functions are extremely useful to prove lower bounds on the complexity of computing functions in other models of computation.

Turing Machines

A Turing Machine is essentially a program written in a particular programming language. The program has access to three

arrays and three pointers:

- x which is accessed using the pointer i. x is an array that can be read but not written into.
- y which is accessed using the pointer j. y can be read and written into.
- z which is accessed using the pointer k. z can only be written into.

The machine is described by its code. Each line of code reads the bits x_i, y_j, z_k, and based on those values, writes new bits into y_j, z_k, and then possibly after incrementing or decrementing i, j, k, jumps to a different line of code or stops computing. Initially, the input is written in x and the goal is for the output to be written in z at the end. i, j, k are all set to 1 to begin with. The arrays all have a special symbol to denote the beginning of the tape and a special symbol to denote the blank parts of the tape.

For example here is a program that copies the input to the output using a single line:

1. If x_i is empty, then HALT. Else set $z_k = x_i$ and increment each of i, k. Jump to step 1.

Here is another that outputs the input bits which are in odd locations:

1. If x_i is empty, then HALT. Else set $z_k = x_i$, increment each of i, k and jump to step 2.
2. If x_i is empty, then HALT. Else increment each of i, k and jump to step 1.

The exact details of this model are not important. The main reason we introduce it is to have a fixed model of computation in mind. For example, it is easy to show that adding more tapes or increasing the alphabet size does not change the model significantly, as we shall discuss further next time.