Lecture 2: Barrington's Theorem, Turing machines

Anup Rao

January 12, 2022

Natural models often have unexpected connections between them. Let us take a brief interlude to explore one such unexpected connection between branching programs and circuits, that was discovered by Barrington. Barrington showed:

Theorem 1. *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 2. *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 1.

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

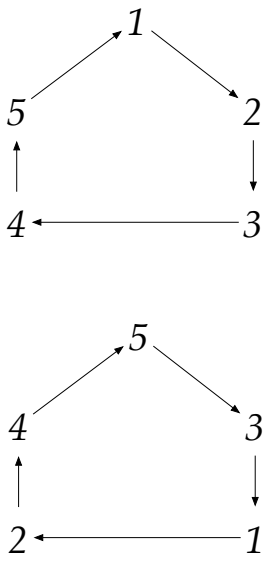

Figure 1: Two cyclic permutations.

visit all elements of $[5]$. For example, the permutation shown in Figure 1 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:

- 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 $\ell_d \leq O(4^d)$. ■

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, and based on those values, (possibly) 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.

Resources of Turing Machines

Once we have fixed the model, we can start talking about the *complexity* of computing a particular function $f : \{0,1\}^* \to \{0,1\}$. Fix

a turing machine M that computes a function f. There are two main things that we can measure:

- Time. We can measure how many steps the turing machine takes in order to halt. Formally, the machine has running time $T(n)$ if on every input of length n, it halts within $T(n)$ steps.
- Space. We can measure the maximum value of j during the run of the turing machine. We say the space is $S(n)$ if on every input of length n, j never exceeds $S(n)$.

The following fact is immediate:

Fact 3. *The space used by a machine is at most the time it takes for the machine to run.*

Robustness of the model: Extended Church-Turing Thesis

THE REASON TURING MACHINES ARE SO IMPORTANT is because of the *Extended Church-Turing Thesis*. The thesis says that *every* efficient computational process can be simulated using an efficient Turing machine as formalized above. Here we say that a Turing machine is efficient if it carries out the computation in polynomial time.

The original (non-extended) thesis made a much tamer claim: that any computation that can be carried out by a human can be carried out by a Turing machine.

The Church-Turing Thesis is not a mathematical claim, but a wishy washy philosophical claim about the nature of the universe. As far as we know so far, it is a sound one. In particular if one changed the above model slightly (say by providing 10 arrays to the machine instead of just 3, or by allowing it to run in parallel), then one can simulate any program in the new model using a program in the model we have chosen.

Claim 4. *A program written using symbols from a larger alphabet Γ that runs in time $T(n)$ can be simulated by a machine using the binary alphabet in time $O(\log|\Gamma| \cdot T(n))$.*

Sketch of Proof We encode every element of the old alphabet in binary. This requires $O(\log|\Gamma|)$ bits to encode each alphabet symbol. Each step of the original machine can then be simulated using $O(\log|\Gamma|)$ steps of the new machine. ■

Claim 5. *A program written for an L-tape machine that runs in time $T(n)$ can be simulated by a program for a 3-tape machine in time $O(L \cdot T(n)^2)$.*

Sketch of Proof The idea is to encode the contents of all the new work arrays into a single work tape. To do this, we can use the first L locations on the work tape to store the first bit from each of the L

arrays, then the next L locations to store the second bit from each of the L arrays, and so on. To encode the location of the pointers, we increase the size of the alphabet so that exactly one symbol from each tape is colored red. This encodes the fact that the pointer points to this symbol of the tape. The actual pointer in the new Turing machine will then do a big left to right sweep of the array to simulate a single operation of the old machine. ■

The following theorem should not come as a surprise to most of you. It says that there is a machine that can compile and run the code of any other machine efficiently:

Theorem 6. *There is a turing machine M such that given the code of any Turing machine α and an input x as input to M, if α takes T steps to compute an output for x, then M computes the same output in $O(CT \log T)$ steps, where here C is a number that depends only on α and not on x.*

We shall say that a machine runs in time $t(n)$ if for every input x, the machine halts after $t(|x|)$ steps (here $|x|$ is the length of the string x). Similarly, we can measure the space complexity of the machine. The crucial point is that small changes to the model of Turing machines does not affect the time/space complexity of computing a particular function in a big way. Thus it makes sense to talk about the running time for computing a function f, and this measure is not really model dependent.

Lower bounds—Counting arguments

WE HAVE SHOWN THAT every function $f : \{0,1\}^n \to \{0,1\}$ can be computed by a circuit of size at most $O(2^n/n)$, and on the other hand we show that for n large enough there is a function that cannot be computed by a circuit of size less than $2^n/(3n)$. The lower bound we prove here was first shown by Shanon. He introduced a really simple but powerful technique to prove it, called a *counting argument*.

Theorem 7. *For every large enough n, there is a function $f : \{0,1\}^n \to \{0,1\}$ that cannot be computed by a circuit of size $2^n/3n$.*

Proof We shall count the total number of circuits of size s, where $s > n$. To define a circuit of size s, we need to pick the logical operator for each (non-input) gate, and specify where each of its two inputs come from. There are at most 3 choices for the logical operation, and at most s choices for where each input comes from. So the number of choices for each non-input gate is at most $3s^2$. The number of choices for an input gate is at most $n < 3s^2$. So, the total

number of choices for each gate is at most $3s^2 + n$, and the number of possible circuits of size s is at most

$$(3s^2 + n)^s \leq (4s^2)^s = 2^{s \log(4s^2)} < 2^{3s \log s},$$

when $n > 4$.

This means that the total number of circuits of size $2^n/3n$ is less than $2^{3 \cdot \frac{2^n}{3n} \cdot n} = 2^{2^n}$. On the other hand, the number of functions $f : \{0,1\}^n \to \{0,1\}$ is exactly 2^{2^n}. Thus, not all these functions can be computed by a circuit of size $2^n/(3n)$. ■

Indeed, the above argument shows that the fraction of functions $f : \{0,1\}^n \to \{0,1\}$ that can be computed by a circuit of size $2^n/4n$ is at most $\frac{2^{\frac{3}{4} \cdot 2^n}}{2^{2^n}} = \frac{1}{2^{2^{n-2}}}$, which is extremely small.

Similar arguments can be used to show that not every function has an efficient branching program (as you will do on your homework).