Lecture 2: Circuits and Turing Machines

Anup Rao

March 28, 2024

LAST TIME, we introduced the computational models of branching programs and Boolean circuits. We discussed how every function $f : \{0,1\}^n \to \{0,1\}$ can be computed by a Branching program. Just like branching programs, boolean circuits can compute *every* function:

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

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 1. 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 2. *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.

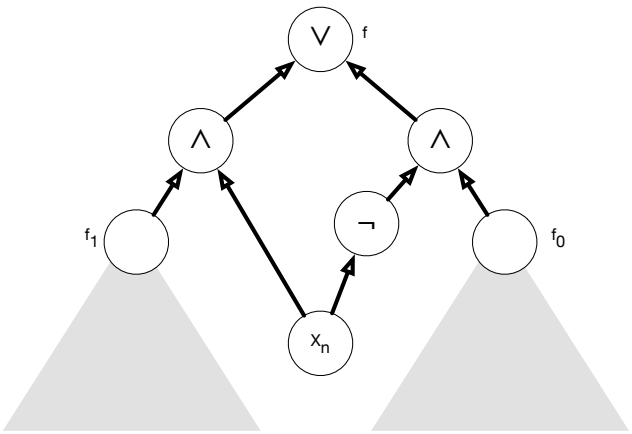

Figure 1: Recursive construction of a circuit for f.

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 for an L-tape machine that runs in time $T(n)$ can be simulated by a program with 1 input tape, 1 work tape and 1 output tape 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. ■