Lecture 2: Turing Machines and Boolean Circuits

Anup Rao

April 1, 2021

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 1. *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 2 (Alphabet does not matter too much). *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 3 (Number of tapes does not matter too much)**.** *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, which we interpret as coloring the contents of the tape either red or blue. During the execution, we maintain the invariant that exactly one symbol from each tape is colored red. This red symbol is the symbol that the pointer is currently pointing to. The actual pointer in the new single-tape Turing machine will then do a big left to right sweep of the single tape to simulate a single step in the execution 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 4 (Universal Simulator)**.** *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.

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 edges coming in, then the vertex is labeled with an input variable x_i,

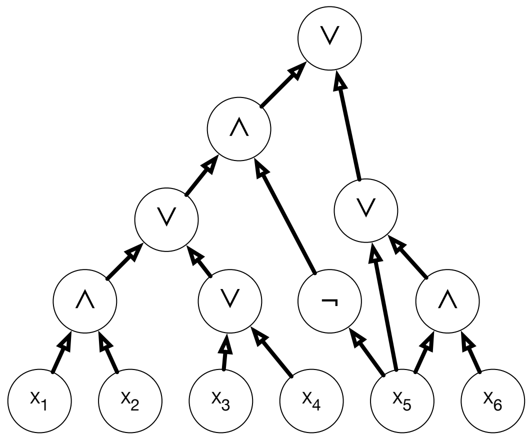

Figure 1: An example of a boolean circuit.

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 1 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 5. *The* size *of the circuit is the number of gates in the circuit.*

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 6. *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 7. *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.

Given a function $f : \{0,1\}^* \to \{0,1\}$, we say that the function has a size $s(n)$ circuit family if for every n, there is a circuit of size $s(n)$ that computes the function correctly on inputs of length n. Similarly, we can talk about the depth complexity of computing a function.

Is it true that every function that can be computed by a circuit of size s can be computed by a circuit of depth $O(\log s)$? Surprisingly, we have no idea how to prove or disprove that statement. Most people believe that it is not true—for it would imply that any algorithm can be parallelized to be exponentially faster!