Lecture 3: Turing Machines

Anup Rao

April 8, 2020

We have the following easy relationship between the size and depth of a circuit. Essentially, the size is at most exponential in the depth, since the worst case is that the circuit looks like the full binary tree:

Fact 1. *Every function computed by a circuit of depth d can be computed by a circuit of size at most 2^{d+1}.*

Proof We prove by induction on the depth that the circuit can be computed using at most $2^d + 2^{d-1} + \ldots + 1 = 2^{d+1} - 1$ gates. When the depth $d = 0$, the circuit must just output the value of a variable, and so has size at most 1.

When $d > 1$, consider the output gate. This gate has two gates that feed into it, each of depth at most $d - 1$. So by induction, the computations of each of those gates can be carried out by circuits of size $2^{d-1} + 2^{d-2} + \ldots + 1$. Thus the overall circuit can be computed with size $1 + 2 \cdot \left(2^{d-1} + 2^{d-2} + \ldots + 1\right) = 2^d + 2^{d-1} + \ldots + 1$, as required. ■

Can we prove the converse? 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!

Complexity of functions

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, and the length and width complexity of the function in terms of branching programs.

Open Problems

One of the reasons I am drawn to complexity theory is that it is the source of many extremely basic open questions. These are questions that are very practically motivated, very easy to state, but no one has a clue about how to solve them.

Having seen the description of circuits and branching program, we can already know enough to state some really fundamental open problems.

It is open to find any *explicit* function that cannot be computed by circuits whose size is $O(n)$ and depth is $O(\log n)$, simultaneously. Note that the size has to be n if the function depends on all its inputs, and the depth has to be $\Omega(\log n)$ if the function depends on all its inputs. So, it is basically open to find any non-trivial circuit lower bound. We do know some highly non-trivial constructions of low-depth circuits. Just a couple of few years ago, Fenner, Gurjar and Thierauf [FGT16] showed that deciding whether or not a bipartite graph has a perfect matching or not can be done in depth $O(\log^2 n)$. The result was later strengthened to handle arbitrary graphs [ST17]. But can such a natural problem be solved in depth $O(\log n)$?

Explicit here means a function that can be succinctly described. For example, as we shall discuss soon, we do know that a random function cannot be computed with small complexity, using counting arguments. But these arguments do not tell you anything about any particular fixed function.

We have seen that every circuit of depth d can be computed by a circuit of depth 2^d. How about the converse? Can every circuit of size 2^d be computed by a circuit of depth $O(d)$? This would be extremely useful — it would show that every algorithm can be parallelized. Most people would believe that this is impossible, but it is open to understand this one way or another.

It is open to find any explicit function that does not admit a branching program of width $O(1)$ and length $O(n \log^3 n)$. We do know of a function that requires $\Omega(n \log^2 n)$ length.

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.

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 2. *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 3. *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 4. *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 5. *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.

References

[FGT16] Stephen A. Fenner, Rohit Gurjar, and Thomas Thierauf. Bipartite perfect matching is in quasi-nc. In Daniel Wichs and Yishay Mansour, editors, *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 754–763. ACM, 2016.

[ST17] Ola Svensson and Jakub Tarnawski. The matching problem in general graphs is in quasi-nc. In Chris Umans, editor, *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, pages 696–707. IEEE Computer Society, 2017.