INFR11102: Computational Complexity 19/09/2018

Lecture 2: Turing Machines

Lecturer: Heng Guo

1 Turing Machines

TM consists of a finite alphabet Σ (with a special "blank" symbol, say $\sqcup$), a tape (imagine an infinitely long strip of paper), a head, and a finite set of states (think line numbers of some code). Basic operations of TMs are:

- move the head to the left or right;
- depending on the current symbol, go to another state;
- write a symbol from Σ on the current head location;
- accept or reject.

Formally speaking, a (single-tape) TM is a tuple $(Q, \Sigma, \delta, q_0, q_{acc}, q_{rej})$ where Q is a finite set of states, Σ is a finite alphabet, δ is a transition function, $q_0 \in Q$ is the starting state and $q_{acc}, q_{rej} \in Q$ are two halting states. The transition function $\delta : (Q \setminus \{q_{acc}, q_{rej}\}) \times \Sigma \to Q \times \Sigma \times \{L, R\}$ describes, given the current state and symbol, what is the next state, what do we want to write on the tape, and which direction do we want to move the head. TM is essentially a function, and these descriptions tell us what it maps the input to. We usually use a transition graph to describe a TM.

During the execution, we can describe the status of TM by the symbols on the tape, the location of the head, and the state TM is in. Such a "snapshot" is called a configuration. The transition function thus dictates what is the new configuration given the current one.

Finiteness is important, since if we allow infinite states, we could recognize any language.

This formal (and original) definition requires that in every step, the head moves, the state changes, and the head writes. It is easy to see that these restrictions are not essential. E.g. if we do not want to move the head, we can move to the left and back. More details on basic properties of TM can be found in [**?**, Chapter 1], [**?**, Chapter 2], or pretty much any book on the theory of computing.

Notice that not all TMs necessarily halt in finite number of steps. Hence for a TM M, it defines a partial function f_M as follows:

$$f_M(x) := \begin{cases} 1 & \text{if } M \text{ accepts } x; \\ 0 & \text{if } M \text{ rejects } x; \\ \text{undefined (or } \infty) & \text{otherwise.} \end{cases}$$

We may simply write $M(x)$, which takes values from $\{0, 1, \infty\}$, instead of $f_M(x)$.

A good model of computation should be robust to small tweaks in the definition, and this is indeed the case for TM. For example, the size of the alphabet Σ does not matter as long as it is finite and has at least two elements (one of which is the blank symbol $\sqcup$). It is even okay if $\Sigma = \{1, \sqcup\}$, but it is quite annoying to consider this unary case and we will not do so. Using $\{0, 1\}^*$, we can encode any character from an alphabet Σ using at most $\lceil \log |\Sigma| \rceil$ bits. This will only incur a constant[1] slowdown.

Also, instead of a single tape, we can define TM with multiple tapes, and it is easy to show that these two models have the same power of computation. (However their efficiencies are different!) We will use multi-tape TM as our standard model of computation.

In [**?**], Turing also introduced the notion of a (single-tape) *Universal Turing Machine* (UTM), which can simulate any other (possibly multi-tapes) TM. A UTM $U(M, x)$ is defined as

$$U(M, x) = M(x),$$

where M is an arbitrary Turing machine (with any appropriate encoding), and x is the input to M.

Note that it is easy to encode TMs so that they can be fed as inputs. All elements of the tuple $(Q, \Sigma, \delta, q_0, q_{acc}, q_{rej})$ are finite, so we can encode all of them using $\{0, 1\}^*$. Not all strings will correspond to a valid TM in this way. For those that are not valid, we may simply map them to a trivial TM, say, one with a single state. Also, for every TM, we can have infinitely many strings encoding it. E.g. we could pad an arbitrarily long string of 0s in the end. Moreover, the tuple (M, x) can be encoded as a single string. We can introduce a special "glue" symbol to concatenate the two parts, and then map down to a smaller alphabet as explained earlier.

We give a high-level sketch of the UTM here. (A detailed description of (single-tape) UTM can be found in e.g. [**?**, Claim 1.6].)

First, if we have a TM with k tapes, we may transform it into a TM with a single tape with quadratic slowdown. Suppose the input of the UTM is (M, x) with $|x| = n$, and the time and space complexity of M are $T(\cdot)$ and $S(\cdot)$. We define a single-tape TM M' by splitting the single tape into k "tracks", so that each track represents one tape of the target TM. This is equivalent to blow up our alphabet to Σ^k. Moreover, we will need an extra bit to record where are the heads of each track, further blowing up the alphabet to $(\Sigma \times \{0, 1\})^k$. Eventually we will use a small alphabet, like $\{0, 1\}$ to encode these enlarged alphabet. For each move of M, we need to first scan the whole tape to find out where the head is, and then move accordingly. Each scan takes time at most $S(n)$, and thus the time complexity of M' is $C_1 T(n) S(n)$ for some constant C_1. The space complexity of M' is $C_2 S(n)$ for some constant C_2.

Then here is what the UTM does on input (M, x) where M is a k-tape machine:

1. Transform M into a single-tape TM M' as discussed above.

[1] Whenever we say *constant*, it means some number that are independent from the input.

2. Write down the description of the new TM M' on a separate working space. In particular, this means we write down a description of the transition function δ.

3. Now we simulate M' on x. Use another separate space to write down the state of the TM M'. We scan the whole working tape to find out what is the content the head is pointing at. Then we match the current state and the content with the corresponding cells in the transition function δ, to figure out what our next move is, and follow it.

Notice that the extra space used only depends on M but not on x. Therefore the UTM uses $C_2 S(n) + C_{M'} = O(S(n))$ space where C'_M is some constant that does not depend on x.

How (in)efficient is this simulation? Steps (1) and (2) take only a constant amount of time (depending on M), since they do not depend on x. In step (3), for each operation of M', we spend some constant amount of time, where the constant C_3 only depends on M'. However the running time of M' is $C_1 T(n) S(n)$, due to the transformation from k-tape to 1-tape. The total running time is therefore $C_1 C_3 T(n) S(n)$ where C_1 and C_3 does not depend on x. The amount of space used by a TM can never exceed the amount of time it has spent. Thus for any TM, $S(n) \leq T(n)$. Hence the total running time is $O(T^2(n))$.

2 Undecidability

The central computational problem in Turing's landmark paper [**?**] is the following one:

Name: HALTING

Input: A TM M and an input x.

Output: Does M with input x halt in finite number of steps?

A very natural attempt to solve HALTING is to use the Universal Turing Machine (UTM) U to simulate (M, x). If $U(M, x)$ halts, then great, we know M halts on input x. However, if $U(M, x)$ does not halt, what can we do?

Turing showed that HALTING is undecidable. It is easy to see that this question can be stated in first-order logic, and thus Turing's result answers Hilbert's ENTSCHEIDUNGSPROBLEM.

The proof of the undecidability is an adaptation of Cantor's diagonalization method. It goes as follows.

Since any TM can be encoded as a natural number, we list all TMs as $M_1, M_2, \cdots$. We build a matrix with infinitely many rows and columns. The rows are all TMs as listed. The columns are all possible inputs, namely we list all elements in Σ^*: $x_1, x_2, \cdots$. The entry of row i and column j is the value of $M_i(x_j) \in \{0, 1, \infty\}$.

Now suppose there is an algorithm (or TM) to determine HALTING. Call it A, so that

$$A(M, x) = \begin{cases} 1 & \text{if } M \text{ halts on } x; \\ 0 & \text{otherwise.} \end{cases}$$

We build another TM A' such that

$$A'(x_i) = \begin{cases} 1 & \text{if } A(M_i, x_i) = 0; \\ \infty & \text{if } A(M_i, x_i) = 1. \end{cases}$$

This A' can be constructed, as all it needs to do is first to figure out the index i, and then to simulate A. For example, we can use the Universal Turing Machine to simulate A. In the second case, A' can simply go into an infinite loop once it figures out $A(M_i, x_i) = 1$.

Since A' is a TM itself, it must be listed with some index k. What is the value of $A'(x_k)$? (There are only two possible values, 1 and ∞.)

- If $A'(x_k)$ does not halt, then $A(M_k, x_k) = 0$. However, by the construction of A', $A'(x_k) = 1$ in this case. Contradiction.
- If $A'(x_k) = 1$, then $A(M_k, x_k) = 1$. However, by the construction of A', $A'(x_k) = \infty$ in this case. Contradiction.

To summarize, there is a contradiction in either case. Hence A does not exist.

References

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[Tur37] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(1):230–265, 1937.