Lecture 5:
Variants of Turing machines, and Decidability

Valentine Kabanets

September 28, 2016

1 Turing machine computation

The TM M starts in state q_0, scanning the leftmost symbol of the input string. The remaining cells of the tape hold blanks. The TM computes according to its transition function δ, going from one configuration to the next configuration. So, a computation is a sequence of configurations of the TM. A TM accepts its input if it enters an *accepting* configuration (which is any configuration where the TM is in state q_{accept}).

Language of TM M is defined as $L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$.

The *time* taken by the TM M on an input x is the number of configurations the machine goes through before halting (or infinity, if the machine never halts).

1.1 Examples

1.1.1 $L = \{0^n 1^n \mid n \geq 0\}$

Recall the language $L = \{0^n 1^n \mid n \geq 0\}$, which we argued is non-regular. While there is no FA deciding this language, we show that there is a TM deciding this L.

The idea is to match the left-most symbol (which should be a 0) with the right-most symbol (which should be a 1), remove these two symbols (replacing them with the blanks), and then iterate starting at the new left-most symbol.

More formally, we have the following TM:

$$\delta(q_0, \sqcup) = (q_{accept}, \sqcup, R)$$
$$\delta(q_0, 1) = (q_{reject}, 1, R)$$
$$\delta(q_0, 0) = (q_1, \sqcup, R)$$

$$\delta(q_1, \sqcup) = (q_2, \sqcup, L)$$
$$\delta(q_1, 0) = (q_1, 0, R)$$
$$\delta(q_1, 1) = (q_1, 1, R)$$

$$\delta(q_2, \sqcup) = (q_{reject}, \sqcup, R)$$
$$\delta(q_2, 0) = (q_{reject}, 0, R)$$
$$\delta(q_2, 1) = (q_3, \sqcup, L)$$

$$\delta(q_3, \sqcup) = (q_0, \sqcup, R)$$
$$\delta(q_3, 0) = (q_3, 0, L)$$
$$\delta(q_3, 1) = (q_3, 1, L)$$

The described TM has the tape alphabet $\Gamma = \{0, 1, \sqcup\}$ and the set of states $Q = \{q_0, q_1, q_2, q_3\}$.

In words, in state q_0 we erase the left-most 0, and enter state q_1, which takes us all the way to the right end of the input string. Once we move past the last symbol, we come one symbol left and enter state q_2. In state q_2, scanning the right-most symbol, we check if this is a 1. If not, we reject; otherwise, we erase that 1, and start moving left all the way to the new left-most symbol, using state q_3. Once we determine the left-most symbol, we enter q_0 and repeat the algorithm on the new, shorter input string.

1.1.2 Palindromes

Next we construct a TM for deciding the language of palindromes

$$L = \{w \in \{0,1\}^* \mid w = w^R\},$$

where w^R means w written in reverse (from right to left).

Deciding the language of palindromes, we design a TM that zig-zags over the input tape, crossing off the leftmost and rightmost symbols of the input, rejecting iff a mismatch is found. (*Exercise:* Write a detailed TM program for the language of palindromes.)

This TM (as well as the TM for the language $\{0^n1^n \mid n \geq 0\}$ defiend earlier) takes time $\Theta(n^2)$, where n is the length of the input string, and the time is the number of configurations used in the computation of our TM on an input of length n (in the worst case).

2 Variants of Turing machines

2.1 2-tape TMs

Can we design a faster TM for palindromes? Not a single-tape TM. It can be shown that one-tape TM needs time n^2. However, if we had a 2-tape TM, we could recognize palindromes in linear time. The idea is to copy the input string onto the second tape (in reverse order), and then go through the strings on both tapes, checking the equality symbol by symbol.

This example shows that 2-tape TM can be faster that 1-tape TM. However, every k-tape TM (for any constant k) can be simulated by a 1-tape TM with only quadratic slowdown. That is, if $t_k(n)$ is the running time of a given k-tape TM on inputs of length n, then there is a usual 1-tape TM accepting the same language and whose running time is at most $O((t_k(n))^2)$. Note the exponent 2 in the time upper bound is independent of the number k of tapes of the machine that we need to simulate on a 1-tape TM; the number k is hidden inside the "Big O" notation as a constant factor in front of $(t_k(n))^2$.

One way to do the simulation is to write the contents of all k tapes on a single tape, separated by special marker symbols #. Each portion corresponding to tape i, will also contain a label on the symbol to mark the location of the tape head for tape i. During the computation, we may need to write an extra symbol on tape i. In the simulation using one tape only, this will require us to make more room. This extra room can be created by moving the contents of the tape one position to the right.

2.2 Other variants of TMs

In general, many variations of TMs were considered (k tapes, k heads, k-dimensional tapes, etc., for a *constant* $k \geq 1$), and all of them were shown equivalent to each to other to within a polynomial factor. That is, any TM M in one of the variations can be simulated by a one-tape TM with only polynomial slowdown (if the original TM runs in time $t(n)$, then the constructed one-tape TM will run in time $(t(n))^c$, for some constant c.) Even general-purpose computers (with their RAM, cache, etc.) can be simulated by one-tape TM with only polynomial slowdown.

Of course, in practice, algorithm running in time n is much more desirable than an algorithm running in time n^2, and so it's better to use sophisticated architecture rather than the basic TM. But, at least theoretically, simple TMs are as powerful as any supercomputer.

Remark 1. *A quantum computer (not built yet) can also be simulated on a Turing machine, using at most exponential time (and polynomial-size memory). We don't know if any faster simulation is possible. One problem where quantum computers currently have an advantage over standard classical computers is Factoring (given an integer in binary, find its prime factors): there is a famous quantum polytime algorithm for Factoring (due to Peter Shor), yet no classical algorithm substantially faster than exponential-time is currently known.*

3 Decidable and semi-decidable languages

Recall that the language of TM M is defined as $L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$.

A TM that halts on all inputs is called a *decider*. If a language $L = L(M)$ for some *decider* M, then L is called *decidable* (or Turing-decidable, or recursive).

If a language $L = L(M)$ for some TM M, then L is called *semi-decidable* (or recognizable, or recursively enumerable (r.e.), or Turing-recognizable). **NOTE:** We do not require that TM M be a decider, that is, M may not halt on some inputs not in the language L.

4 Nondeterministic Turing machine (NTM)

We define nondeterministic computation, a variant of TM that is not known to be efficiently implementable. A *nondeterministic* Turing machine (NTM) is like a deterministic TM (DTM), except there may be more than one possibility for a next configuration. The transition function of a NTM is $\delta(q, a) = \{(q_1, b_1, m_1), \ldots, (q_k, b_k, m_k)\}$ where $m_i \in \{L, R\}$.

The computation of a DTM is a sequence of configurations, i.e., a *path*. The computation of a NTM is a *tree* whose nodes represent configurations.

We say that a NTM M *accepts* an input string w if there exists a legal computation of M on w that ends with an accepting configuration. In terms of the tree of computations of M on w, this means that there exists some branch in the tree that is an accepting computation.

Theoretically, we can test if an NTM M accepts w by performing the breadth-first search on the computation tree of M on w: start with the initial configuration, create a list of all possible next configurations (those reachable for the initial configuration in a single step), then a list of all configurations reachable from the initial one in two steps, and so on. If at some point in time, we see that an accepting configuration is reachable, we stop and accept. Otherwise, we continue going until M halts, or forever if M never halts. What this argument gives us is: if M accepts w, then

our breadth-first search simulation of M on w will halt and accept; if M doesn't accept w, our simulation will never accept either, but it may run forever.

4.1 Decidability and semi-decidability by NTMs

We observe that if a language L is semi-decided by some NTM M, then there is a DTM M' that also semi-decides L. The idea is to do breadth-first search on the computation tree of M on a given input x (layer by layer) accepting and halting if we ever see an accepting configuration. Note that if M accepts x, then our breadth-first search will terminate and accept. If, on the other hand, M doesn't accept x, our DTM M' may or may not terminate, but it certainly will not accept x.

Similarly, if L is decidable by some NTM M, there there is a decider DTM M' that decides L as well. (Use the same algorithm as above. Note that since NTM M is a decider, we get by definition that the computation tree of M on x is finite. Thus we can test in finite time if any branch on this tree is an accepting computation of M on x.)

The upshot is:

> The definition of decidable and semi-decidable does not depend on the TM being deterministic or non-deterministic!

For simplicity, however, we will assume by default that deciders and semi-deciders are DTMs.

Next, we exhibit an explicit problem that cannot be solved on a computer, the famous Halting Problem of Turing.

5 Halting Problem

Can we tell if a given TM is a decider? Or, even simpler, can we tell if a given TM halts on a given input string? No, the latter is the famous Halting Problem that was shown undecidable by Alan Turing in 1936.

Consider the language

$$A_{TM} = \{\langle M, w\rangle \mid \text{ TM } M \text{ accepts input } w\}.$$

Theorem 1. *A_{TM} is semi-decidable.*

Theorem 2. *A_{TM} is undecidable.*

We'll see the proofs of these theorems next time.