CS 6902 (Theory of computation) – Lecture 2

Antonina Kolokolova [*]

May 9, 2018

1 Models of computation

In this course, we will mainly study models of computation that compute decision problems (in most cases the complexity of solving decision and function problems is very similar).

The two main models we will consider are the Finite State Machines (or Finite Automata) which are described in terms of states, and state changes as a reaction to the environment (i.e., input) and a "finite automaton with a memory", a Turing machine. Intuitively, an example of a finite automaton is something like a simple coke machine, or a sensor-operated bus door; a Turing machine, which can revisit its input multiple times and write things down is a full-blown computer. There is an intermediate model of a Pushdown Automaton (where the memory access is limited to be a first-in-last-out pipe) with an intermediate complexity.

Most of you have seen buses where you open the door by waving a hand in front of it. The door would not open, though, if the bus is moving, and would close when the motion sensor stops receiving the "wave" signal. This is an example of a system which receives a sequence of signals (this is its input string) and responds to them by changing its state and performing an action (output). For example, if the bus is in the "stopped with door closed" state, and it receives a "wave" signal, it will open the door. Eventually the bus shuts down, ending the input string.

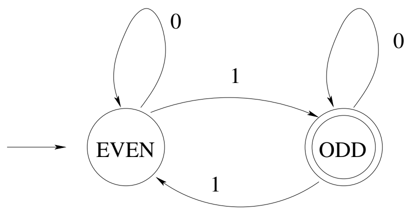
Note that this kind of controller does not have any memory: all it does is look at the input symbol by symbol, and change its state accordingly. It would only have finitely many states.

As we are focusing on decision problems here, we will focus on whether a system finishes in a state corresponding to a "yes" answer ("accepting" a given input string) or "no" answer

[*]The material in this set of notes came from many sources, in particular "Introduction to Theory of Computation" by Sipser and course notes of U. of Toronto CS 364. Special thank you to Richard Bajona for sharing his scribed lecture notes.

(”rejecting” its input). Here is an example of a finite automaton which determines whether the input string has an odd number of 1s. Here, circles represent states (double-circles are for ”accepting” states, signifying ”yes” answer), and arrows are transitions, labeled by its input symbols. The automaton starts at state denoted by a little unlabeled arrow (start state), and finishes when it exhausted its input; the state in which it finishes determines whether it accepts or rejects this input string.

Example 1 The following finite automaton receives a string of bits (0 and 1), and should finish in a final state (double-circle) if and only if the string contains an odd number of 1s.



More precisely, we define (deterministic) Finite Automata as follows.

Definition 1 *A (deterministic)* finite automaton *(a DFA) is a 5-tuple* $(Q, \Sigma, \delta, q_0, F)$*, where*

1) Q is a finite set of states*.*

2) Σ is the alphabet *(a finite set of symbols).*

3) $\delta : Q \times \Sigma \rightarrow Q$ is the transition function *(pronounced as “delta”).*

4) $q_0 \in Q$ is the start state

5) $F \subseteq Q$ is the set of accept (final) states*.*

We say that a DFA accepts a string $w = \{w_0 \dots w_{n-1}\}$ *if there exists a sequence of* n *states* $r_0, \dots, r_n$ *such that* $r_0 = q_0$, $r_n \in F$ *and* $\forall i, 0 \leq i < n, \delta(r_i, w_i) = r_{i+1}$. *A DFA* accepts (recognizes) a language L *if it accepts every string in L, and does not accept any string not in L. A language is called a* regular language *if it is recognized by some finite automaton.*

So to fully specify a finite automaton it is sufficient to say which states it is composed of (Q), what are the possible input symbols (Σ), where to start (q_0), where to end on good strings (F) and, the main part, what are the arrows and labels on them (δ). In example 1, $Q = \{even, odd\}$, $\Sigma = \{0, 1\}$, $q_0 = even$, $F = \{odd\}$ and δ is encoded by the following table:

	0	1
even	even	odd
odd	odd	even

For example, on the string 1011001 it will go through the sequence of states “even, odd, odd, even, odd, odd, odd, even”, ending in a non-accepting state; but a string 101100 it will accept. So this automaton accepts a language of all strings over 0,1 in which the number of 1s is even.

Definition 2 *A* non-deterministic *finite automaton (NFA) is a 5-tuple* $(Q, \Sigma, \delta, q_0, F)$*, where* Q*,* Σ*,* q_0 *and* F *are as in the case of deterministic finite automaton (*Q *is the set of states,* Σ

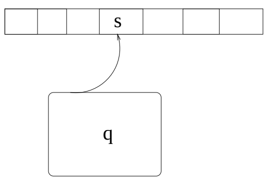
an alphabet, q_0 is a start state and F is a set of accepting states), but the transition function δ is $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$. Here, $\mathcal{P}(Q)$ is the powerset (set of all subsets) of Q.

A non-deterministic finite automaton accepts *a string $w = w_1 \dots w_m$ if there exists a sequence of states $r_0, \dots r_m$ such that $r_0 = q_0$, $r_m \in F$ and $\forall i, 0 \leq i < m, r_{i+1} \in \delta(r_i, w_i)$.*

There are two differences between this definition of δ and the deterministic case. First, δ gives a (possibly empty) set of states, as opposed to exactly one state. Second, there can be a transition without seeing any symbols, on empty string ϵ. Also, now the acceptance condition is that there exists a good sequence of states leading to acceptance: there can be many computational paths, some of which would lead to rejecting states, some would "crash" with no next state to go to, and, if the string is in the language, at least one will finish in an accepting state.

Turing Machines

In 1936, Alan Turing gave the first definition of an algorithm, in the form of (what we call today) a Turing machine. He wanted a definition that was simple, clear, and sufficiently general. Although Turing was only interested in defining the notion of "algorithm", his model is also good for defining the notion of "efficient" (polynomial-time) algorithm.



A Turing machine consists of an infinite tape and a finite state control. The tape is divided up into squares, each of which holds a symbol from a finite tape alphabet that includes the blank symbol $\not{b}$. Some definitions give two-way infinite tape; Sipser's book uses tape infinite to the right (so it has a start cell, but no end cell).

The machine has a read/write head that is connected to the control, and that that scans squares on the tape. Depending on the state of the control and symbol scanned, it makes a move, consisting of

- printing a symbol
- moving the head left or right one square
- assuming a new state

The tape will initially be completely blank, except for an input string over the finite input alphabet Σ; the head will initially be pointing to the leftmost symbol of the input.

A Turing machine M is specified by giving the following:

- Σ (a finite input alphabet).
- Γ (a finite tape alphabet). $\Sigma \subseteq \Gamma$. $\not{b} \in \Gamma - \Sigma$.
- Q (a finite set of states). There are 3 special states:
    - q_0 (the initial state)
    - q_{accept} (the state in which M halts and accepts)
    - q_{reject} (the state in which M halts and rejects)
- $\delta : (Q - \{q_{accept}, q_{reject}\}) \times \Gamma \to Q \times \Gamma \times \{L, R\}$ (the transition function)
  (If $\delta(q, s) = (q', s', h)$, this means that if q is the current state and s is the symbol being scanned, then q' is the new state, s' is the symbol printed, and h is either L or R, corresponding to a left or right move by one square.

The Turing machine M works as follows on input string $x \in \Sigma^*$.
Initially x appears on the tape starting from its left end, with infinitely many blanks to the right, and with the head pointing to the leftmost symbol of x. (If x is the empty string, then the tape is completely blank and the head is pointing to the leftmost square.) The control is initially in state q_0.
M moves according to the transition function δ.
M may run forever, but if it halts, then it halts either in state q_{accept} or q_{reject}. (We will mainly be interested in whether M accepts or rejects; there are definitions of what it means for a Turing machine to output a string: e.g., the output is what is left on the tape when the machine halts)

Definition 3 *M* accepts *a string $x \in \Sigma^*$ if M with input x eventually halts in state q_{accept}. We write $\mathcal{L}(M) = \{x \in \Sigma^* | M \text{ accepts } x\}$, and we refer to $\mathcal{L}(M)$ as the language accepted by M.*

(Notice that we are abusing notation, since we refer both to a Turing machine accepting a string, and to a Turing machine accepting a language, namely the set of strings it accepts.)

Example 2 PARITY= the set of all binary strings that have an even number of 1s = $\{x | x \in \{0,1\}^*, x \text{ has even number of occurrences of symbol } 1\}$.

A very simple Turing machine recognizes the set of such strings in time linear in the number of symbols in the strings. For that, it moves once through the string from left to right, at each point remembering whether the number of 1s it has seen so far is even or odd. If by the time it reaches a blank symbol it has seen an even number of 1s, the Turing machine accepts, and if it has seen an odd number of 1s, it rejects.

State q	*Symbol* s	*Action* $\delta(q,s)$
q_0	0	$(q_0, \not{b}, R)$
q_0	1	$(q_1, \not{b}, R)$
q_0	$\not{b}$	$(q_{accept}, \not{b}, L)$
q_1	0	$(q_1, \not{b}, R)$
q_1	1	$(q_0, \not{b}, R)$
q_1	$\not{b}$	$(q_{reject}, \not{b}, L)$

A more complicated example is a Turing machine recognizing the set of palindromes.

Example 3 PAL = the set of even length palindromes =
$\{yy^r | y \in \{0,1\}^*$, where y^r means y spelled backwards.

We will design a Turing machine M that accepts the language PAL$\subseteq \{0,1\}^*$. M will have input alphabet $\Sigma = \{0,1\}$, and tape alphabet $\Gamma = \{0,1,\not{b}\}$. (Usually it is convenient to let Γ have a number of extra symbols in it, but we don't need to for this simple example.) We will have state set $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_{accept}, q_{reject}\}$.

If, in state q_0, M reads 0, then that symbol is replaced by a blank, and the machine enters state q_1; the role of q_1 is to go to the right until the first blank, remembering that the symbol 0 has most recently been erased; upon finding that blank, M enters state q_2 and goes left one square looking for symbol 0; if 0 is not there then we *reject*, but if 0 is there, then we erase it and enter state q_3 and go left until the first blank; we then goes right one square, enter state q_0, and continue as before.

If we read 1 in state q_0, then we operate in a manner similar to that above, using states q_4 and q_5 instead of q_1 and q_2.

If we read $\not{b}$ in state q_0, this means that all the characters of the input have been checked, and so we accept.

Formally, the transition function δ is as follows. (Note that when we accept or reject, it doesn't matter what we print or what direction we move in; we arbitrarily choose to print $\not{b}$ and move right in these cases.)

State q	*Symbol s*	*Action $\delta(q,s)$*
q_0	0	$(q_1, \not b, R)$
q_1	0	$(q_1, 0, R)$
q_1	1	$(q_1, 1, R)$
q_1	$\not b$	$(q_2, \not b, L)$
q_2	1	$(q_{reject}, 0, R)$
q_2	$\not b$	$(q_{reject}, 0, R)$
q_2	0	$(q_3, \not b, L)$
q_3	0	$(q_3, 0, L)$
q_3	1	$(q_3, 1, L)$
q_3	$\not b$	$(q_0, \not b, R)$
q_0	1	$(q_4, \not b, R)$
q_4	0	$(q_4, 0, R)$
q_4	1	$(q_4, 1, R)$
q_4	$\not b$	$(q_5, \not b, L)$
q_5	0	$(q_{reject}, 0, R)$
q_5	$\not b$	$(q_{reject}, 0, R)$
q_5	1	$(q_3, \not b, L)$
q_0	$\not b$	$(q_{accept}, 0, R)$

Multi-tape Turing Machines

Definition 4 *A k-tape Turing machine has k tapes and k heads (although still only one control unit). Thus a transition function is* $\delta: (Q - \{q_{accept}, a_{reject})x\Gamma^k \rightarrow \{Q, \Gamma^k, \{L, R\}^k\}$.

Palindrome can be solved faster on a two-tape Turing machine by the following algorithm. Here, a Turing machine has two tapes (and two heads), and it starts with its input written on the first tape, and both heads at the leftmost position. When we write the transition function, list the state, then the symbols being read at the first and second tape, respectively, then the direction of movement for the first and second tape.

1) Copy the input to the second tape: $\{q_0, 0, \not b\} \rightarrow \{q_0, 0, 0, R, R\}$ and the same for 1s.

2) Move the first head to the beginning of the tape; leave the second head at the end of the input (to imitate a head staying at one place, use two movements, one left and one right).

3) Now start moving first head right and the second left, comparing the symbols they are looking at. If at any moment they look at different symbols, reject, otherwise accept when the first head sees the blank.

We now formally define the notion of worst case time complexity of Turing machines.

Let M be a Turing machine over input alphabet Σ. For each $x \in \Sigma^*$, let $t_M(x)$ be the number of steps required by M to *halt* (i.e., terminate in one of the two final states) on input x. (Each step is an execution of one instruction of the machine, and we define $t_M(x) = \infty$ if M never halts on input x.)

Definition 5 (Worst case time complexity of M) *The worst case time complexity of M is the function $T_M : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ defined by*

$$T_M(n) = \max\{t_M(x) \mid x \in \Sigma^*, |x| = n\}.$$

If T_M is polynomial in $|x|$, we say that the Turing machine runs in polynomial time; if it is linear in $|x|$, then in linear time.

Church-Turing thesis

Church-Turing thesis: A Turing machine captures exactly the intuitive notion of an algorithm. That is, anything that can be algorithmically solved (in the intuitive sense of "algorithmically") can also be solved by a Turing machine.

An extension of the Church-Turing thesis is as follows:

Extended Church–Turing Thesis *Everything* efficiently *computable on a physical computer (in the intuitive sense) is* efficiently *computable by a Turing machine (in the formal sense).*

Quantum computers pose a potential challenge to this thesis, if there is a technology to build them for arbitrarily large inputs.