CSE 105: Introduction to the Theory of Computation Fall 2015

Lecture Notes: Finite State Automata

Instructor: *Daniele Micciancio* UCSD CSE

This lecture notes are provided as a supplement to the textbook. The notes are very terse, and you should first read the relevant sections from the book. (Sections 1.1 and 1.2 from Chapter 1.) Then refer to these notes as a brief summary of the notation and definitions used in class to describe finite state automata and computations. Notice that the definition of DFAs and NFAs is identical to the one given in the book. The only difference here is in the style used to define computations.

1 DFAs

A Deterministic Finite Automaton (DFA) is a 5-tuple $M = (Q, \Sigma, \delta, s, F)$ consisting of

1. A finite set of states Q
2. Finite set of input symbols Σ
3. A transition function $\delta : Q \times \Sigma \to Q$
4. A start state $s \in Q$
5. A set of accepting states $F \subseteq Q$

A DFA takes an input string w over the alphabet Σ, and either accepts or rejects the string. Identifying acceptance with the value 1 and rejection with 0, one can think of the DFA as a machine that takes a string w as input, and outputs a single bit $b \in \{0, 1\}$.

1.1 Modeling computation as a function from the input to the output

The computation performed by the automaton can be defined is several ways. The simplest approach is to model the computation as a function $f_M \colon \Sigma^* \to \{0, 1\}$ mapping the input string $w \in \Sigma^*$ to the output bit $f_M(w) \in \{0, 1\}$. Here Σ^* is the standard notation (used in Section 1.3 of the textbook) to represent the set of all strings over the alphabet Σ, i.e., the set of all finite sequences of 0,1 or more elements from Σ. The empty string (i.e., the sequence of length 0) is represented by a special symbol ϵ. The function f_M can be formally defined as follows. First extend the transition function $\delta : Q \times \Sigma \to Q$ to a function $\delta^* \colon Q \times \Sigma^* \to Q$. The difference between δ and δ^* is that δ^* takes not just a symbol, but an entire string as its second argument. We let $\delta^*(q, w)$ be the state reached by the DFA when starting in state q and consuming the string w from the input. This extended transition function is defined by induction on (the length of) the second argument according to the rules

- (Base case: $|\epsilon| = 0$): $\delta^*(q, \epsilon) = q$, for every $q \in Q$
- (Inductive case: $|aw| \geq 1$): $\delta^*(q, aw) = \delta^*(\delta(q, a), w)$, for all $q \in Q$, $a \in \Sigma$ and $w \in \Sigma^*$.

As discussed in class, this is an inductive *definition,* not an inductive *proof.* If you are unfamiliar with inductive definitions, just think of it as the definition of a *recursive program* to compute δ^*. Using the extended transition function, the function computed by the automaton is defined as

$$f_M(w) = \begin{cases} 0 & \text{if } \delta^*(s, w) \notin F \\ 1 & \text{if } \delta^*(s, w) \in F \end{cases}$$

All these definitions are easily translated into computer programs. (See notes on implementing DFAs in Haskell.)

1.2 Modeling computation as a transition system

Sometimes one needs a more detailed description of the computation than the one provided by the input/output function f_M. For example, the function describing the input/output mapping of a program P, does not provide information about the running time of P. (In the case of DFAs this is not much of a loss because the number of steps executed by the program is always equal to the length of the input, but for NFAs and more complex models of computation this is not generally true.) Alternatively, one can model the computations of a DFA $M = (Q, \Sigma, \delta, s, F)$ using a more operational style as follows:

1. First one defines a set $C = Q \times \Sigma^*$ of "configurations". A configuration represents a snapshot of the system during an execution, inluding its internal storage, input, etc. or whatever is relevant to carry out the computation one step at a time. In our case, a configuration (q, u) represents the current state of the automaton $q \in Q$, and the portion of the input $u \in \Sigma^*$ that still needs to be consumed.

2. A function $I\colon \Sigma^* \to C$ mapping the input string to a corresponding initial configuration. In our case, $I(w) = (s, w)$ maps the input string w to the configuration consisting of the initial state of the automaton $s \in Q$ and the whole input string $w \in \Sigma^*$.

3. A set of *final* or *halting* configurations $H \subseteq C$, corresponding to configurations in which the computation may terminate. In our case, $H = Q \times \epsilon = \{(q, \epsilon)\colon q \in Q\}$ is the set of all configurations representing an automaton that has no more input symbols to read.

4. A transition relation $R \subseteq C \times C$ consisting of all pairs (c, c') such that the system can go from configuration c to configuration c' in one execution step. The condition $(c, c') \in R$ is usually written as $c \to_R c'$. Equivalently, R can be regarded as a transition function, mapping each configuration $c \in C$ to the set of configurations $\{c'\colon c \to_R c'\} \subseteq C$ that can be reached from c in one step of computation.

   The transition relation of a DFA is the set of all pairs $(q, au) \to_R (\delta(q, a), u)$ where $q \in Q$, $a \in \Sigma$ and $u \in \Sigma^*$. In other words, the automaton can go from state q and a nonempty input au with first symbol a, to the new state $\delta(q, a)$ while keeping the remaining portion of the input u .

5. An output function $O\colon H \to V$ mapping the final configurations (at the end of the computation) to an output value. In the case of a DFA, the output is in $V = \{0, 1\}$ and the output function is defined as $O(q, \epsilon) =$ if $(q \in F)$ then 1 else 0.

A computation on input $w \in \Sigma^*$ is then defined as a sequence of configurations $c_1, c_2, \ldots, c_n$ such that

1. $c_1 = I(w)$ is the initial configuration
2. $c_i \to_R c_{i+1}$ for all $i = 1, \ldots, n-1$
3. $c_n \in H$ is a final configuration

The output of the computation is the value $O(c_n)$. A computation can also be infinite, in which case the sequence $c_1, c_2, \ldots$ extends indefinitely, and condition (3) on the last configuration being final is omitted.

The state transition system associated to a DFA is *deterministic*, i.e., for any configuration $c \in C$, the number of possible "next" configurations c' (i.e., such that $c \to_R c'$) is either 0 (if $c \in H$ is a final configuration) or 1 (if $c \in C \setminus H$ is a nonfinal configuration). If this is the case, for any input $w \in \Sigma^*$ there is precisely one computation of M on input w, and (if the computation is finite, which is always the case for DFAs) one output value $O(c_n)$. So, from the defintion of computation we can recover a function f_M mapping the input string w to the output value $f_M(w) = O(c_n)$. However, the computation (defined as a sequence of configurations $c_1, \ldots, c_n$ provides additional information. For example, the length n of the computation gives the "running time" of the program, measured as the number of transitions followed to go from the initial configuration $I(w)$ to the final configuration $c_n = (q, \epsilon)$. This is not very interesting in the case of DFAs because $n = |w|$ is always equal to the length of the initial input string. But it provides very useful information for more complex computational models.