CS 3719 (Theory of Computation and Algorithms) – Lecture 3

Antonina Kolokolova[*]

January 10, 2011

1 Finite Automata

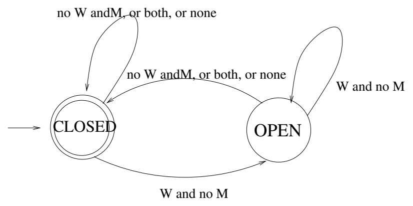
In the last lecture, we talked about regular expressions. A natural question to ask is how powerful and how complex regular expressions are. In particular, are there languages that cannot be expressed using regular expressions? And how much of computational resources are needed to find out if a given string matches regular expression?

To answer these questions, we will introduce a model of computation called Finite Automata. This is a pretty simple model, in that it will have no memory, and just process the input string symbol by symbol.

Example 1 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.

Let's describe a possible controller for a door like this. It will have two states: open and closed, and 4 pairs of possible events (both moving and waving, waving not moving, moving not waving and neither). Also, we will require that at the power-up the door is closed, and it is closed at power-down.

	Both	W., not M	M., not W	Neither
Open	Closed	Open	Closed	Closed
Closed	Closed	Open	Closed	Closed



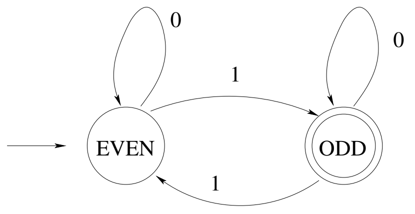
[*]The material in this set of notes came from many sources, in particular "Introduction to Theory of Computation" by Sipser and course notes of U. of Toronto CS 364.

The circles denote the states, and the arrows denote inputs. The unlabeled arrow pointing to the "Closed" state denotes that the controller starts when the door is closed; it is a double-circle to denote that it is also a final state before power-down.

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.

Here is another example of a problem that can be solved by a similar kind of device.

Example 2 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) $\delta : Q \times \Sigma \to Q$ *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.

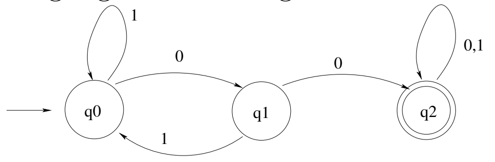
We say that a DFA accepts a string $w = \{w_0 \dots w_{n-1}\}$ *if there exists a sequence of n states* $r_0, \dots, r_n$ *such that* $r_0 = q_0$, $r_n \in F$ *and* $\forall i, 0 \le i < n, \delta(r_i, w_i) = r_{i+1}$. *A DFA* accepts (recognizes) a language L *if it accepts every string in L, and does 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 2, $Q = \{even, odd\}$, $\Sigma = \{0, 1\}$, $q_0 = even$, $F = \{odd\}$ and δ is encoded by the following table:

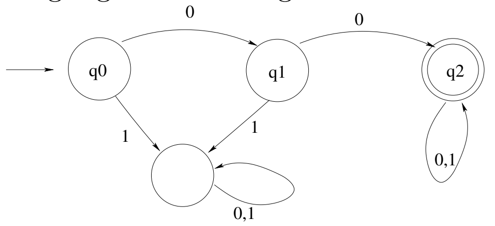
	0	1
even	even	odd
odd	odd	even

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.

Example 3 This automaton recognizes the language of all strings that contain 00.



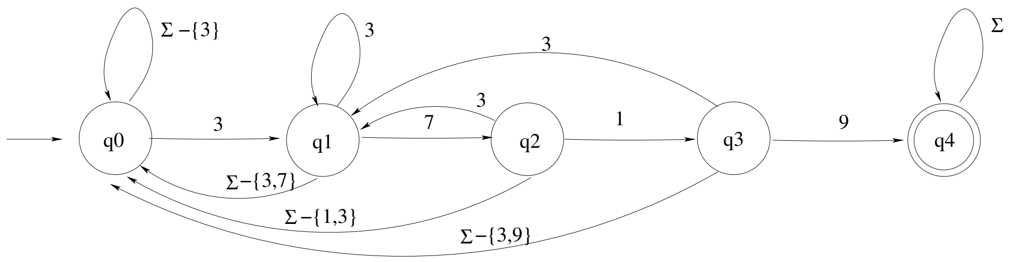
Example 4 This automaton recognizes the language of all strings that start with 00.



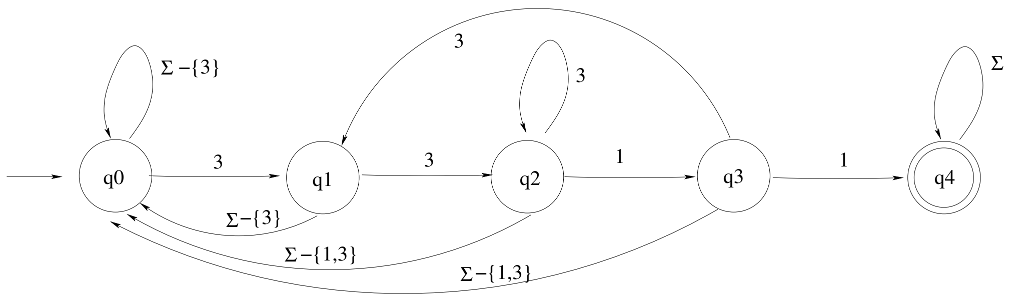
1.1 String matching

One area where automata have found much use is in string matching. Here, the pattern string (of length m is encoded by an automaton, which then gets the text (of usually much larger length n) as an input; the automaton should end in an accepting state if and only if this pattern occurs in the text. Such an algorithm runs very fast: in time $O(n|\Sigma|)$ plus time it takes to build an automaton: easy $O(m^3)$ (ignoring $|\Sigma|$); can be as efficient as $.O(m)$. In particular, Knuth-Morris-Pratt algorithm you have seen in CS 2711 can be viewed as constructing a DFA in its preprocessing stage: think symbols of the pattern corresponding to the states $q_1 \dots q_m$, matching transitions going q_i to q_{i+1} and the partial match table encoding non-matching transitions.

Example 5 Suppose you want to match a string "3719" in a file (here, take Σ to be all symbols that can occur in a text file). It is enough to feed the text from the file to this DFA.



Example 6 To make it more interesting, let's build an automaton for a string with repeated letters, for the same Σ. Notice how backward transitions correspond to the "suffix matches the prefix" jumps in the KMP algorithm.



q0
q1
q2
q3
q4
Σ −{3}
3
3
3
3
1
1
Σ
Σ−{3}
Σ−{1,3}
Σ−{1,3}