EECS 2001A : Introduction to the Theory of Computation

Suprakash Datta

Course page: `http://www.eecs.yorku.ca/course/2001`
Also on Moodle

Regular Languages: Definition

- The language recognized by a finite automaton (DFA or NFA) M is denoted by $L(M)$
- A regular language is a language for which there exists a recognizing finite automaton
- We study properties of regular languages to understand finite automata

Important Questions

- Given the description of a finite automaton $M = (Q, \Sigma, \delta, q, F)$, what is the language $L(M)$ that it recognizes?
- In general, what kind of languages can be recognized by finite automata? (What are the regular languages?)
- It is easiest to define regular languages RECURSIVELY

Towards a Recursive Definition of Regular Languages

Recall: a language is a set of words over some alphabet
Base case:

- The empty language is regular (WHY?)
- Every set $\{a\}$, $a \in \Sigma$ is a regular language
- Later: We will show that every finite language is regular

Towards a Recursive Definition of Regular Languages - 2

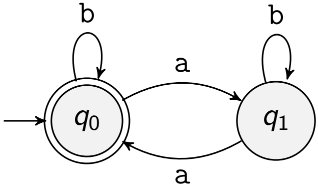
Need rules to build up bigger languages

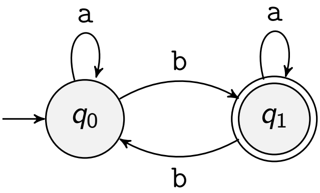
- The union of two regular languages is regular
- This needs proof
- If we can run two DFA "in parallel", we can decide if a word belongs to the union. How do we get a DFA to do this?
- We will present a somewhat complicated proof (Theorem 2.3.1 in the text) that simulates the idea above, but first let us use the idea on an example problem

An Example (Tutorial problem)

Consider the alphabet $\Sigma = \{a, b\}$. Design a DFA for the language $\{w : |w| > 0, \text{ and } w \text{ has an even number of } a's \text{ and an odd number of } b's\}$.

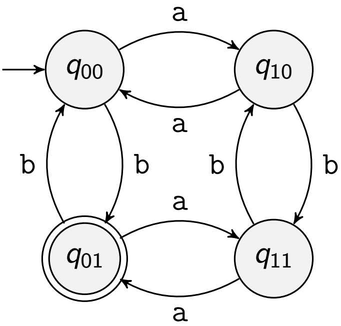
The two DFA's are:





The Solution

Consider the alphabet $\Sigma = \{a, b\}$. Design a DFA for the language $\{w : |w| >$
$0,$ and w has an even number of $a's$ and an odd number of $b's\}$.



The Union of Two Regular Languages is Regular

Suppose $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ accept languages L_1, L_2. We prove that $L_1 \cup L_2$ is regular
Proof idea: construct a DFA M that tracks the state of M_1, M_2

- M accepts $L_1 \cup L_2$. So,

$$\forall w \in \Sigma^*, M \text{ accepts } w \Leftrightarrow M_1 \text{ accepts } w \text{ or } M_2 \text{ accepts } w$$

- Define $M = (Q_3, \Sigma, \delta_3, q_3, F_3)$ by
  - $Q_3 = Q_1 \times Q_2 = \{(r_1, r_2) | r_1 \in Q_1, r_2 \in Q_2\}$
  - $\delta_3((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a))$
  - $q_3 = (q_1, q_2)$
  - $F_3 = \{(r_1, r_2) | r_1 \in F_1 \text{ or } r_2 \in F_2\}$
- We can complete the proof by induction on the length of w

Towards a Recursive Definition of Regular Languages - 3

Need rules to build up bigger languages

- The complement of a regular language is regular
- The proof idea is straightforward: Take the DFA that recognizes the language and make all non-accepting states accepting and vice versa

The Complement of a Regular Language is Regular: Proof

Take the DFA M that recognizes the language and construct M' that is identical to M except that all non-accepting states in M are accepting in M' and vice versa.

- We show that $w \in \overline{L}$ if and only if M' accepts w
- Since the set of states, the initial state and the transition function of M and M' are identical, the sequence of states $(r_0, r_1, \ldots, r_n)$ that M goes through on input w is identical to the sequence of states M' goes through on input w
- Now consider 2 cases:
  - $w \in L$
  - $w \notin L$ (i.e., $w \in \overline{L}$)

Towards a Recursive Definition of Regular Languages - 4

Define concatenation of languages: $L_1 \cdot L_2 = \{xy | x \in L_1, y \in L_2\}$
Example: $\{a, b\} \cdot \{0, 11\} = \{a0, a11, b0, b11\}$
Caveat: If any of the 4 elements are missing, the set is not $L_1 \cdot L_2$!

- Another rule to build up bigger languages: The concatenation of two regular languages is regular
- Terminology: regular languages are **closed** under concatenation (and also closed under union from the prior result)
- This also needs proof

Proving the Concatenation Theorem

- Given the two languages, we "know" DFA M_1, M_2 that recognize the two languages
- If a word $w \in L_1 \cdot L_2$ then $w = w_1 w_2$ such that w_1 is accepted by M_1 and w_2 is accepted by M_2
- Problem: given a string w, how does the automaton know where the part accepted by M_1 stops and the part accepted by M_2 substring starts?

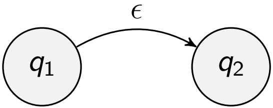
We need a new idea!

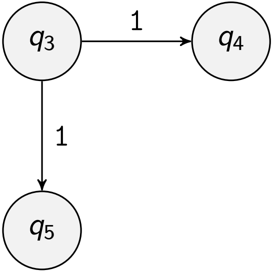
Nondeterminism

- Nondeterministic machines are capable of being lucky, no matter how small the probability
- Alternatively, it can "magically" make the right choices
- As mentioned before, nondeterminism cannot be implemented
- For any (sub)string w, the nondeterministic machine can be in a set of possible states
- If any if the final states is an accepting state, then the machine accepts the string
- "The automaton processes the input in a parallel fashion. Its computational path is no longer a line, but a tree." (Sipser)

Nondeterministic Finite Automata (NFA)

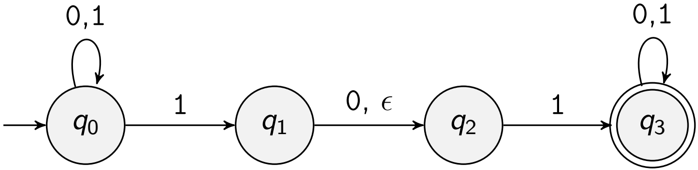
A NFA **may** have transition rules/possibilities like

- 

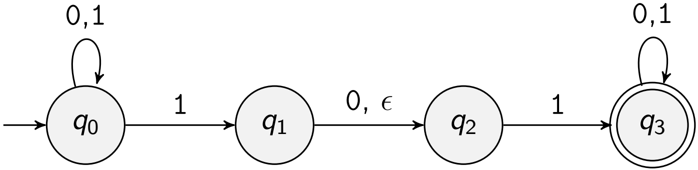
- 

Nondeterministic Finite Automata (NFA) - 2

What does this NFA do?



NFA: Tracing Examples



- 1: May be in states q_0, q_1, q_2! None of those are accepting states, so reject
- 01: May be in states q_0, q_1, q_2! None of those are accepting states, so reject
- 0110: It can reach state q_3, hence accept!
  $(q_0 \to q_0 \to q_1 \to q_2 \to q_3 \to q_3)$
- the fact that there are non-accepting paths is of no consequence

NFA Drawing Conventions

- All transitions need **not** be present
- All but one state **must** be drawn
- Unlabeled transitions are assumed to go to a reject state (not drawn) from which the automaton **cannot** escape

NFA: Formal Definition

A NFA M is defined by a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, with

- Q: finite set of states
- Σ: finite alphabet
- δ: transition function $\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$
- $q_0 \in Q$: start state
- $F \subseteq Q$: set of accepting states

NFA: More on the Transition function

- The function $\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ is the crucial difference between DFA, NFA
- It means: "When reading symbol 'a' while in state q, the machine can go to one of the states in $\delta(q, a) \subseteq Q$"
- The ϵ in $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ takes care of the empty string transitions

NFA: recognizing languages

- Informal idea: Given a language, the NFA recognizes it, i.e., it accepts every string in the language, and rejects every string not in the language
- Formally: A NFA $M = (Q, \Sigma, \delta, q, F)$ accepts a string/word $w = w_1 \dots w_n$ if and only if we can rewrite w as $y_1 \dots y_m$ with $y_i \in \Sigma_\epsilon$ and there is a sequence $r_0, \dots, r_m$ of states in Q such that:
  - $r_0 = q_0$
  - $r_{i+1} \in \delta(r_i, y_{i+1})$ for all $i = 0, 1, \dots, m$–1
  - $r_m \in F$

NFA: Exercises - 1

Give NFAs with the specified number of states that recognize the following languages over the alphabet $\Sigma = \{0, 1\}$:

- $\{w | w \text{ ends with } 00\}$, three states
- $\{0\}$; two states
- $\{w | w \text{ contains even number of zeroes, or exactly two ones}\}$, six states
- $\{0^n | n \in \{0, 1, 2, \ldots\}\}$, one state