Finite Automata

SUBMITTED BY:SHABNAM
COMPUTER SCIENCE DEPARTMENT

Finite Automata

- When there are multiple automata for a system, it is useful to incorporate all of the automata into a single one so that we can better understand the interaction.
- Called the *product* automaton. The product automaton creates a new state for all possible states of each automaton.
- Two types – both describe what are called **regular languages**
  - Deterministic (DFA) – There is a fixed number of states and we can only be in one state at a time
  - Nondeterministic (NFA) –There is a fixed number of states but we can be in multiple states at one time

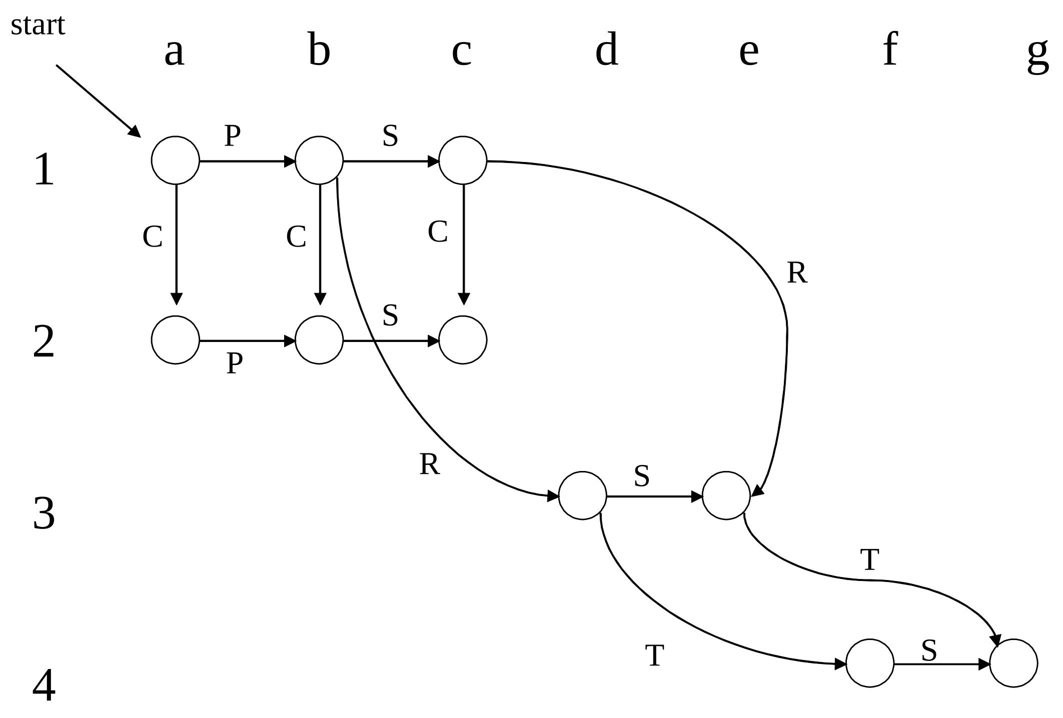
Actions

- The automata only describes actions of interest
  - To be more precise, with a DFA (deterministic finite automaton) we should specify arcs for all possible inputs.
  - E.g., what should the customer automaton do if it receives a "redeem"?
  - What should the bank do if it is in state 2 and receives a "redeem"?

Entire System as Automaton

- Since the customer automaton only has one state, we only need to consider the pair of states between the bank and the store.
  - For example, we start in state (a,1) where the store is in its start state, and the bank is in its start state. From there we can move to states (a,2) if the bank receives a cancel, or state (b,1) if the store receives a pay.
- To construct the product automaton, we run the bank and store automaton "in parallel" using all possible inputs and creating an edge on the product automaton to the corresponding set of states.

Product Automaton

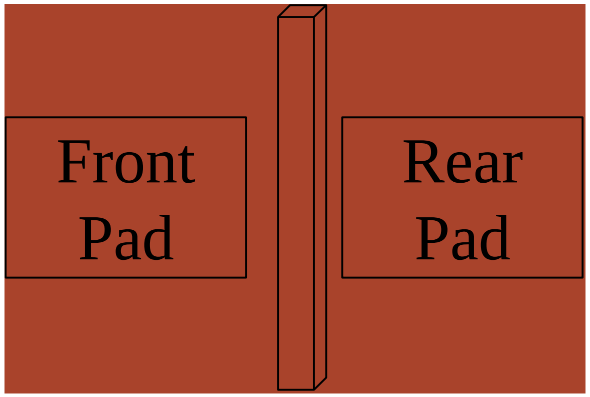


Product Automaton

- How is this useful? It can help validate our protocol.
- It tells us that not all states are reachable from the start state.
  - For example, we should never be in state (g,1) where we have shipped and transferred cash, but the bank is still waiting for a redeem.
- It allows us to see if potential errors can occur.
  - We can reach state (c, 2). This is problematic because it allows a product to be shipped but the money has not been transferred to the store.
  - In contrast, we can see that if we reach state (d, 3) or (e, 3) then the store should be okay – a transfer from the bank must occur
    - assuming the bank automaton doesn't "die" which is why it is useful to add arcs for all possible inputs to complete the automaton

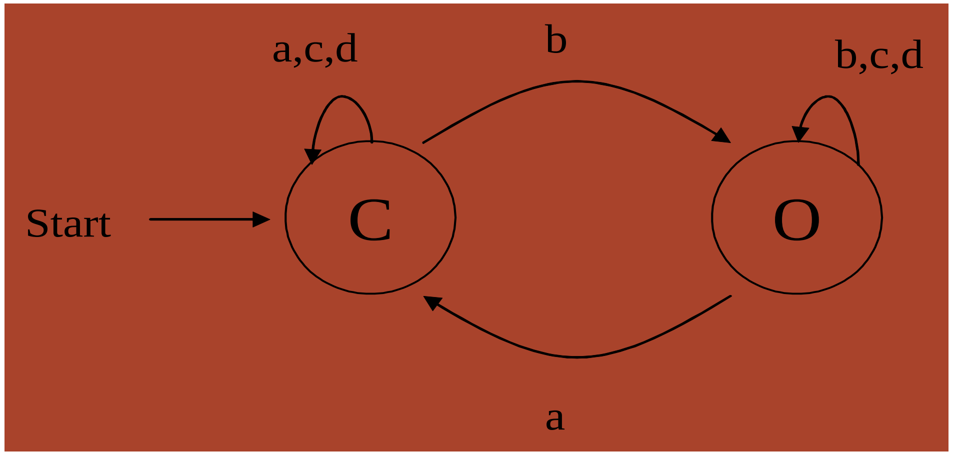
Simple Example – 1 way door

- consider a one-way automatic door. This door has two pads that can sense when someone is standing on them, a front and rear pad. We want people to walk through the front and toward the rear, but not allow someone to walk the other direction:



One Way Door

- Let's assign the following codes to our different input cases:
  - a - Nobody on either pad
  - b - Person on front pad
  - c - Person on rear pad
  - d - Person on front and rear pad
- We can design the following automaton so that the door doesn't open if someone is still on the rear pad and hit them:



Formal Definition of a Finite Automaton

1. Finite set of states, typically Q.
2. Alphabet of input symbols, typically Σ
3. One state is the start/initial state, typically q_0
4. Zero or more final/accepting states; the set is typically F. A transition function, typically δ. This function
   - Takes a state and input symbol as arguments.
   - Returns a state.
   - One "rule" would be written $\delta(q, a) = p$, where q and p are states, and a is an input symbol.
   - Intuitively: if the FA is in state q, and input a is received, then the FA goes to state p (note: q = p OK).
5. A FA is represented as the five-tuple: $A = (Q, \Sigma, \delta, q_0, F)$. Here, F is a set of accepting states.

Formal Definition of Computation

- Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton and let $w = w_1 w_2 \ldots w_n$ be a string where each w_i is a member of alphabet Σ.
- M **accepts** w if a sequence of states $r_0 r_1 \ldots r_n$ in Q exists with three conditions:
  1. $r_0 = q_0$
  2. $\delta(r_i, w_{i+1}) = r_{i+1}$ for $i=0, \ldots, n-1$
  3. $r_n \in F$

In other words, the language is all of those strings that are accepted by the finite automata.

DFA Example

* Here is a DFA for the language that is the set of all strings of 0's and 1's whose numbers of 0's and 1's are both even:

$$\begin{tikzcd}
\text{Start} \arrow[r] & q_0 \arrow[r, bend left, "1"] \arrow[d, bend left, "0"] & q_1 \arrow[l, bend left, "1"] \arrow[d, bend left, "0"] \\
& q_2 \arrow[u, bend left, "0"] \arrow[r, bend left, "1"] & q_3 \arrow[l, bend left, "1"] \arrow[u, bend left, "0"]
\end{tikzcd}$$

DFA Exercise

- The following figure below is a marble-rolling toy. A marble is dropped at A or B. Levers x_1, x_2, and x_3 cause the marble to fall either to the left or to the right. Whenever a marble encounters a lever, it causes the lever to reverse after the marble passes, so the next marble will take the opposite branch.
- Model this game by a finite automaton. Let acceptance correspond to the marble exiting at D. Non-acceptance represents a marble exiting at C.

Regular Operations

- Brief intro here – will cover more on regular expressions shortly
- In arithmetic, we have arithmetic operations
  - + * / etc.
- For finite automata, we have **regular operations**
  - Union
  - Concatenation
  - Star

Algebra for Languages

1. The union of two languages L and M is the set of strings that are in both L and M.
   - Example: if L = { 0, 1} and M = {111} then L ∪ M is {0, 1, 111}.
2. The concatenation of languages L and M is the set of strings that can be formed by taking any string in L and concatenating it with any string in M. Concatenation is denoted by LM although sometimes we'll use L•M (pronounced "dot").
   - Example, if L = {0, 1} and M = {ε, 010} then LM is { 0, 1, 0010, 1010}.

Closure Properties of Regular Languages

- **Closure** refers to some operation on a language, resulting in a new language that is of the same "type" as those originally operated on
  - i.e., regular in our case
- We won't be using the closure properties extensively here; consequently we will state the theorems and give some examples. See the book for proofs of the theorems.
- The regular languages are closed under union, concatenation, and *. I.e., if A_1 and A_2 are regular languages then
  - $A_1 \cup A_2$ is also regular
  - A_1A_2 is also regular
  - A_1^* is also regular

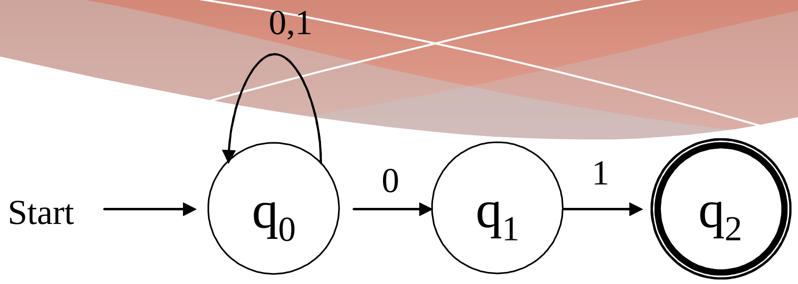
Nondeterministic Finite Automata

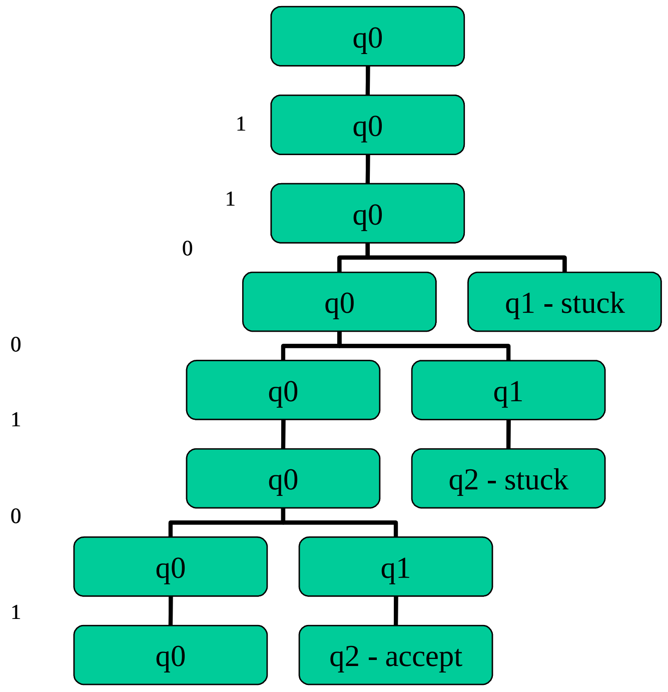
- A NFA (nondeterministic finite automata) is able to be in several states at once.
  - In a DFA, we can only take a transition to a single deterministic state
  - In a NFA we can accept multiple destination states for the same input.
  - You can think of this as the NFA "guesses" something about its input and will always follow the proper path if that can lead to an accepting state.
  - Another way to think of the NFA is that it travels all possible paths, and so it remains in many states at once. As long as at least one of the paths results in an accepting state, the NFA accepts the input.

NFA Example

- This NFA accepts only those strings that end in 01

- Running in “parallel threads” for string 1100101





Formal Definition of an NFA

- Similar to a DFA

1. Finite set of states, typically Q.
2. Alphabet of input symbols, typically $\sum$
3. One state is the start/initial state, typically q_0
4. Zero or more final/accepting states; the set is typically F.
5. A transition function, typically δ . This function:
   - Takes a state and input symbol as arguments.
   - Returns a **set of states** instead of a single state, as a DFA
6. A FA is represented as the five-tuple: A = $(Q, \sum, \delta, q_0, F)$. Here, F is a set of accepting states.

Previous NFA in Formal Notation

The previous NFA could be specified formally as:

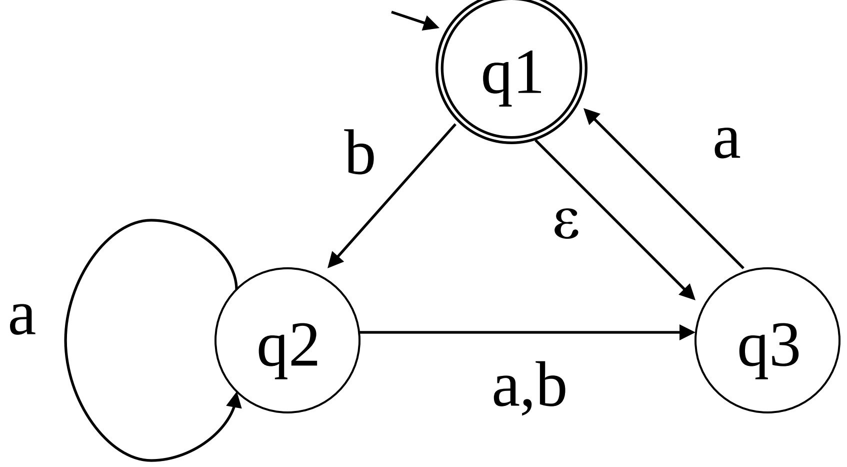
$(\{q_0,q_1,q_2\}, \{0,1\}, \delta, q_0, \{q_2\})$

The transition table is:

		0	**1**
→	**q0**	{q0,q1}	{q0}
	q1	**Ø**	{q2}
*	**q2**	**Ø**	**Ø**

NFA Example

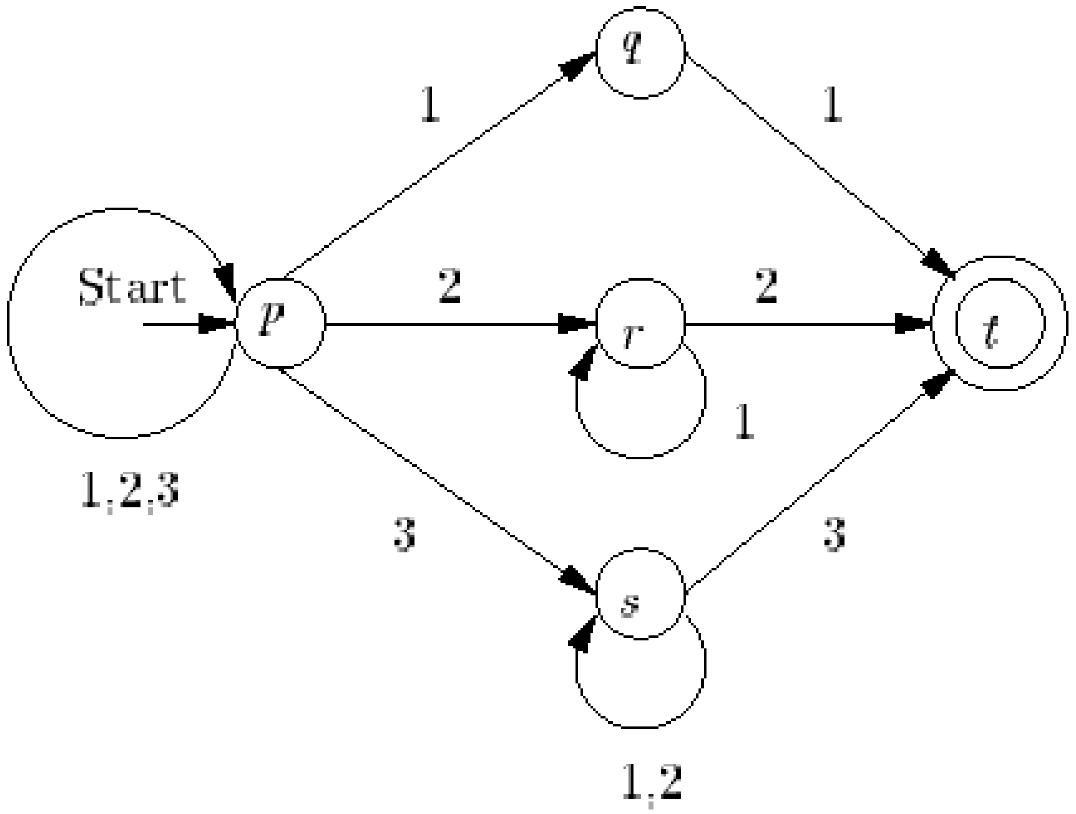
- Practice with the following NFA to satisfy yourself that it accepts ε, a, baba, baa, and aa, but that it doesn't accept b, bb, and babba.



NFA Exercise

- Construct an NFA that will accept strings over alphabet {1, 2, 3} such that the last symbol appears at least twice, but without any intervening higher symbol, in between:
  - e.g., 11, 2112, 123113, 3212113, etc.
- Trick: use start state to mean "I guess I haven't seen the symbol that matches the ending symbol yet." Use three other states to represent a guess that the matching symbol has been seen, and remembers what that symbol is.

NFA Exercise



You should be able to generate the transition table

Formal Definition of an NFA

- Same idea as the DFA
- Let N = (Q, ∑, δ, q_0, F) be an NFA and let w = $w_1 w_2 \ldots w_n$ be a string where each w_i is a member of alphabet ∑.
- N **accepts** w if a sequence of states $r_0 r_1 \ldots r_n$ in Q exists with three conditions:
    1. $r_0 = q_0$
    2. $r_{i+1} \in \delta(r_i, w_{i+1})$ for i=0, … , n-1
    3. $r_n \in F$

Observe that δ(ri, w_{i+1}) is the **set** of allowable next states

We say that N **recognizes** language A if A = {w | N accepts w }

Equivalence of DFA’s and NFA’s

- For most languages, NFA’s are easier to construct than DFA’s
- But it turns out we can build a corresponding DFA for any NFA
  - The downside is there may be up to 2^n states in turning a NFA into a DFA. However, for most problems the number of states is approximately equivalent.
- Theorem: A language L is accepted by some DFA if and only if L is accepted by some NFA; i.e. : L(DFA) = L(NFA) for an appropriately constructed DFA from an NFA.
  - Informal Proof: It is trivial to turn a DFA into an NFA (a DFA is already an NFA without nondeterminism). The following slides will show how to construct a DFA from an NFA.

* THANK

YOU