Outline

• What is a Non-deterministic Finite State Automata
• NFA construction
• Formal Definition of NFA
• Equivalence between NFAs and DFAs
• Regular Operations
  – Closure under union
  – Closure under concatenation
  – Closure under star

From Sipser Chapter 1.2
Nondeterminism

• So far our FA is deterministic in that the state and next symbol determines the next state

• Nondeterministic Finite States Automata introduce two main differences:
  – DFAs have one transition arrow per alphabet symbol, while NFAs have 0 or more for each and \( \varepsilon \) the “empty” symbol
  – DFAs are in a specific single state at all times, while NFAs may be in multiple states
How does an NFA Compute?

• When there is a choice, all paths are followed
  – Think of it as cloning a process and continuing
  – If there is no arrow, the path terminates and the clone dies (it
does not accept if at an accept state when that happens)
  – An NFA may have the empty string cause a transition
  – The NFA accepts if any path terminate in an accepting state
  – Can also be modeled as a tree of possibilities

• An alternative way of thinking of this
  – At each choice you make one guess of which way to go
  – You magically always guess the right way to go
- Try out 010110
  - Is it accepted?
    - Yes

- What is the language?
  - Strings containing a substring of 101 or 11
Construct an NFA that accepts all strings over \{0,1\} with a 1 in the third position from the end.

Hint: the NFA stays in the start state until it guesses that it is three places from the end.
Formal Definition of NFA

A nondeterministic finite automata is a 5-tuple \((Q, \Sigma, \delta, q_0, F)\) where

- \(Q\) is a finite set of states
- \(\Sigma\) is a finite set called the alphabet
- \(\delta : Q \times \Sigma \epsilon \rightarrow P(Q)\) is the transition function
- \(q_0 \in Q\) is the start state
- \(F \subseteq Q\) is the set of accept states

• Similar to DFA except
  - \(\Sigma\) includes \(\epsilon\)
  - The transition function matches states and symbols with a set of possible states
Example of Formal Definition of NFA

NFA $N_1$ is $(Q, \Sigma, \delta, q_1, F)$

- $Q = \{q_1, q_2, q_3, q_4\}$
- $\Sigma = \{0, 1\}$
- $q_1$ is the start state
- $F = \{q_4\}$

<table>
<thead>
<tr>
<th>State</th>
<th>0</th>
<th>1</th>
<th>$\varepsilon$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$</td>
<td>${q_1}$</td>
<td>${q_1, q_2}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>${q_3}$</td>
<td>$\emptyset$</td>
<td>${q_3}$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>$\emptyset$</td>
<td>${q_4}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$q_4$</td>
<td>${q_4}$</td>
<td>${q_4}$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
Equivalence of NFAs and DFAs

• NFAs and DFAs recognize same class of languages
• What does this mean? What is the implication?
  – NFAs have no more power than DFAs
    • With respect to what can be expressed
    • Every NFA has an equivalent DFA
    • But NFAs may make it easier to describe some languages
  – Terminology: Two machines are equivalent if they recognize the same language
Compiler Equivalence

• C, C++, Python, Pascal, Fortran, ...
• Are these languages equivalent?
  – Some are more suited to some tasks, but with enough effort any of these languages can compute anything the others can
  – If necessary, you can even write a compiler for one language using another
Proof of Equivalence of NFA & DFA

Proof idea:

– Simulate an NFA with a DFA
– With NFAs, given an input we follow all possible branches and keep a finger on the state for each
– In the equivalent DFA we need to keep track of all the possible states we would be in for each of the NFA execution
– If the NFA has $k$ states then it has $2^k$ possible subsets
  • Each subset corresponds to one of the possibilities that the DFA needs to remember
  • The DFA will have $2^k$ states
Proof by Construction

• Let $N=(Q, \Sigma, \delta, q_0, F)$ be the NFA recognizing $A$

• Construct equivalent DFA $M = (Q', \Sigma, \delta', q_0', F')$
  
  - $Q = P(Q)$
  
  - Careful in handling $\varepsilon$!
    
    • Let $E(R)$ denote the collection of states reachable by members of $R$ just following $\varepsilon$ arrows

  - $q_0' = \{q_0 \cup E(\{q_0\}) \}$

  - $F' = \{R \in Q' \mid R \text{ contains an accept state of } N\}$

  - Transition function
    
    • The state $R$ in $M$ corresponds to a set of states in $N$
    
    • When $M$ reads symbol $a$ in state $R$, it shows where $a$ takes each state
    
    • $\delta'(R,a) = ( \bigcup_{r \in R} \delta(r,a) ) \cup E(R)$
Example 1.41 (pg. 57 2nd ed.)

– The NFA has 3 states: \( Q = \{1, 2, 3\} \)
– What are the states in the DFA?
  • \( \emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\} \)
– What are the start states of the DFA?
  • Start states of the DFA correspond the collection of just the stating state of the NFA and all the states reachable via \( \varepsilon \)
  • \( \{1, 3\} \)
– What are the accept states?
  • All the states of the DFA which include an accept state of the NFA
  • \( \{\{1\}, \{1,2\}, \{1,3\}, \{1,2,3\}\} \)
Example: Convert an NFA to a DFA

Now lets work on some of the transitions

– Let’s look at state 2 in NFA and complete the transitions for state 2 in the DFA
  • Where do we go from state 2 on an “a” and “b”?  
    – On “a” to state 2 and 3 and on “b” to state 3
  • So what state does \{2\} in DFA go to for a and b?  
    – On a to \{2,3\} and \{3\} for b

– Now lets do state \{3\}
  • On “a” goes to \{1,3\} and on b goes to \emptyset  
    – Why \{1, 3\}? Because first goes to 1 then \(\varepsilon\) permits a move back to 3!
Closure under Regular Operations

• We started this before and did it for Union only
  – Union much simpler using NFA
• Concatenation and Star much easier using NFA
• Since DFAs equivalent to NFAs, we can now just use NFAs
• Fewer states to keep track of because we can act as if we always “guess” correctly
Why do we care about closure?

We need to look ahead

– A regular language is what a DFA/NFA accepts
– We are now introducing regular operators and then will generate regular expressions from them (Ch 1.3)
– We will want to show that the language of regular expressions is equivalent to the language accepted by NFAs/DFAs (i.e., a regular language)
– How do we show this?
  • Basic terms in regular expression can generated by a FA
  • We can implement each operator using a FA and the combination is still able to be represented using a FA
Given two regular languages $A_1$ and $A_2$ recognized by two NFAs $N_1$ and $N_2$, construct $N$ to recognize $A_1 \cup A_2$

How do we construct $N$?

- Start by writing down $N_1$ and $N_2$. Now what?
- Add a new start state and then have it take $\epsilon$ branches to the start states of $N_1$ and $N_2$
Closure under Concatenation

- Given two regular languages $A_1$ and $A_2$ recognized by two NFAs $N_1$ and $N_2$, construct $N$ to recognize $A_1 \cdot A_2$

- How do we do this?
  - The complication is that we did not know when to switch from handling $A_1$ to $A_2$ since we can loop on an accept state
  - Solution with NFA:
    - Connect every accept state in $N_1$ to every start state in $N_2$ using an $\varepsilon$ transition
      - Do not remove transitions from accept state in $N_1$ back to $N_1$
Closure under Concatenation

• Given:
  – \( N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1) \) recognizes \( A_1 \)
  – \( N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2) \) recognizes \( A_2 \)

• Construct \( N = (Q_1 \cup Q_2, \Sigma, \delta, q_1, F_2) \) so that it recognizes \( A_1 \cdot A_2 \)

\[
\delta(q,a) =
\begin{array}{|c|c|}
\hline
\delta_1(q,a) & q \in Q_1 \text{ and } q \notin F_1 \\
\hline
\delta_1(q,a) & q \in F_1 \text{ and } a \neq \varepsilon \\
\hline
\delta_1(q,a) \cup \{q_2\} & q \in F_1 \text{ and } a = \varepsilon \\
\hline
\delta_2(q,a) & q \in Q_2 \\
\hline
\end{array}
\]
Closure under Star

• Given regular language $A_1$ prove $A_1^*$ is also regular
  – Note $(ab)^* = \{\varepsilon, ab, abab, ababab, \ldots\}$
• Proof by construction
  – Take NFA $N_1$ that recognizes $A_1$ and construct $N$ from it that recognizes $A_1^*$
  – How do we do this?
    • Add new $\varepsilon$-transition from accept states to start state
    • Then make the start state the accept state so that $\varepsilon$ is accepted
      – This almost works, but not quite. What is the problem?
        » May have transition from intermediate state to start state and should not accept on this loop-back
    • Solution: add a new start state that is accept state, with an $\varepsilon$-transition to the original start state and have $\varepsilon$-transitions from accept states to old start state
Closure under Star

\[ \varepsilon \]

Theory of Computation - Fall'19
Lorenzo De Stefani