CS 361 Meeting 6 — 9/21/16

Announcements
1. Homework 2 due today.
2. Sample solutions for homework 1 are online. These are for current 361 students only. You are expected not to share them with other students who might take the course in the future.
3. Starting with this week’s homework we will revise the homework grading system in a silly but helpful way. The grading categories described on the website will remain the same, but we will add 5 points to the score received on each question (as long as that score is greater than 0). Therefore, all scores assigned will come from the set \{10, 9, 8, 7, 6, 0\}.

Quick Review
1. Nondeterministic Finite Automata allow multiple possible transitions out of a single state for a single input symbol:

2. On a given input, the presence of multiple transitions out of some states for a given symbol means that there will be multiple paths the machine could follow for a given input string.

3. We interpret such diagrams as describing the language of strings for which it is possible to find a path from the start state to some accepting state by following any one of the transitions that are shown for each input symbol.

4. Just as we gave formal definitions to explain how to understand DFAs, we can do the same for NFAs.
   - **Definition.** A NFA is a five tuple \( D = (Q, \Sigma, \delta, s, F) \) where:
     \( Q \) is a finite set of states
     \( \Sigma \) is the input alphabet
     \( \delta : Q \times \Sigma \rightarrow P(Q) \) is a state transition function
     \( s \in Q \) is the start state
     \( F \subseteq Q \) is a set of accept states
   - We define a function \( \hat{\delta} \), which extends \( \delta \) so that we can apply it to strings instead of just individual symbols:
     \[ \hat{\delta}(\pi, \varepsilon) = \pi \quad (\pi \in P(Q)) \]
     \[ \hat{\delta}(\pi, wx) = \bigcup_{q \in \hat{\delta}(\pi,w)} \delta(q, x) \quad (\pi \in P(Q), x \in \Sigma, w \in \Sigma^*) \]
   - **Definition** \( L(D) = \{ w | w \in \Sigma^* \text{ and } \hat{\delta}(q_0, w) \cap F \neq \emptyset \} \)

Nondeterminism = Determinism?
1. Last class, I began the presentation of a proof that any language recognized by a nondeterministic finite automaton can also be recognized by a deterministic finite automaton.
   - Actually, my proof only covered a simplified version of NFAs. There is one additional feature I will introduce today. The hope, however, is that working with the simplified definition will make the fundamental idea in the proof of equivalence stand out more clearly. Then, if we have time we will extend it to the standard definition of NFA.
2. What we wanted to prove is

**Theorem:** if \( L = L(N) \) for some nondeterministic finite automaton \( N \) then there is a deterministic finite automaton, \( D \) such that \( L(D) = L \).
3. The basic idea of the proof is that given a NFA $N$, we can construct a DFA $D$ each of whose states corresponds to some subset of states that might be simultaneously active under the parallel or “follow all possible paths” interpretation of nondeterminism.

4. With this in mind, if the machine $N$ described in the statement of our theorem is $N = (Q, \Sigma, \delta_N, s_N, F_N)$, we will build a machine $D$ whose set of states has one state corresponding to each subset of $Q$. That is, the states of $D$ will be the set of all subsets of $Q$ which is just the power set of $Q$.

$$D = (\mathcal{P}(Q), \Sigma, \delta_D, s_D, F_D)$$

5. Given this set of state, $s_D$ should be $\{s_N\}$.

6. $F_D$ should contain all subsets of states that include any final state of $N$. That is

$$F_D = \{\pi \mid \pi \in \mathcal{P}(Q) \land \pi \cap F_N \neq \emptyset\}$$

7. $\delta_D$ should be defined to figure out where we might go from each state of $N$ in the current state of $D$. That is:

$$\delta_D(\pi, x) = \bigcup_{q \in \pi} \delta_N(q, x)$$

8. Last time, I ran out of time as I was demonstrating how this construction would work when applied to the NDA shown below which recognizes strings that end in 10.

9. The resulting machine would look like the machine below.

The label on each set indicates the subset of states in the NFA that correspond to that state in the DFA.

The arcs shown in red all originate from nodes that cannot be reached from the start state. Therefore, these transitions and all the states except the three in the upper left can be removed from the machine without changing its behavior.

$\epsilon$-transitions

1. Given that DFAs and NFAs are of equivalent power it is often easier to prove languages are regular by designing an appropriate NFA than by defining a DFA.

- Consider how you would prove that if $F = L(D)$ is the language of a DFA $D$ with a single final state then $F^R = \{w \mid w^R \in F\}$ (where $w^R$ is the string obtained by reversing the symbols in $w$) is a regular language?

  - You can construct an NFA, $N$, from $M$ by reversing all of the edges in the diagram for $D$ and interchanging the start and final states. More technically, if $D = (Q, \Sigma, \delta, s, \{f\})$, we can define $N = (Q, \Sigma, \delta', s, \{f\})$ where

$$\delta'(q, x) = \{q' \mid \delta(q', x) = q\}$$
• The machine $N$ described in this construction is likely to be a NFA even if $D$ was deterministic. For example, if we apply this construction to our deterministic machine for recognizing binary numbers divisible by 3, we get a machine for recognizing strings that are formed by reversing binary numbers divisible by 3, but this machine has to guess when it is about to read the last symbol.

– Since the original machine had multiple final states, the NDFA we would produce would need to have multiple start states. This is not allowed in either a DFA or a NDFA.

2. Suppose we try to generalize the result about reversing languages we considered above to machines with multiple final states:

**Lemma:** If $F = L(D)$ is the language of a DFA $D$ then $R = \{w \mid w^R \in F\}$ is a regular language.

• The construction we sketched out above does not work because in the NFA we described, the single final state of $D$ became the single start state of the NFA $N$.

  – Suppose we start with the machine for the language we considered last time of binary strings that don’t contain the substring 110

  – We could add a new start state to $N$ whose transitions were chosen so that on the first input symbol, it could reach any state that could have been reached from any of the former final states of $D$, but...

  • There is a nice feature that is usually included in the definition of the NFA that provides a cleaner way.

  • The idea is to allow transitions that are labeled with the empty string. These are called epsilon-transitions. If an NFA reaches state $s$ at some point in its computation and there is an epsilon transition from $s$ to $s'$, then the NFA can move to $s'$ without consuming any input.

  – Using $\epsilon$-transitions our reversed version of the machine for strings that don’t contain 110 would look like:
Another Interesting Exercise

1. Let’s look at an example that takes full advantage of the power of non-determinism.

2. Suppose that we define

\[ L_{\frac{1}{2}} = \{ x \mid \text{there exists } y \text{ such that } |x| = |y| \text{ and } xy \in L \} \]

Consider how we can prove that \( L_{\frac{1}{2}} \) is regular if \( L \) is regular.

3. I want to explore two distinct approaches to building NFAs that show that regular languages are closed under this operation.

- One approach is closely related to the language reversal operation we just discussed. In this approach, we will build an NFA that simultaneously simulates a DFA examining its input in two ways at the same time. One copy of the simulated DFA starts at the beginning of the actual input. The other simulated copy works its way backward from the end of an imagined second-half of the input. The goal of the backward simulation is to guess a string that could serve as the \( y \) in the definition of \( L_{\frac{1}{2}} \).

- The other approach still manages two simultaneous simulations, but both run forward. One scans the actual input. The other guesses and scan a string that might form \( y \) from a state that the machine also guesses will correspond to \( \delta(s, x) \).

- In both of these simulations, we will use non-determinism to guess things (all the letters of \( y \) and either the machine state that comes between \( x \) and \( y \) or the final state after \( y \)). We will, however, carefully define our final states so that they will only be reachable if the machine makes the right guesses!

4. Let’s consider the parallel forward scan version first:

- Suppose that \( D \) is a DFA that accepts \( L \).
- The idea is that given input \( x = x_1x_2 \ldots x_n \), we want so simulate one version of \( D \) scanning \( x \) at the same time we simulate another version of \( D \) scanning some string \( y = y_1y_2 \ldots y_n \) in the hope of verifying that \( xy \in L(D) \).
- This involves a lot of guessing. Most obviously, the second version of \( D \) we simulate has to guess what all the \( y \)s are.
- In addition, while we want the simulation of the scan of \( x \) to start in the start state, \( s \) of \( D \), that is not where the scan of \( y \) should start. Instead, we should start the scan of \( y \) in the state \( m = \delta(s, x) \). Our first scan will figure out what \( m \) is. Unfortunately, it will not have figured this out yet when we need to start the simulated scan of \( y \). So, we also need to guess \( m \)!
- The trick that makes all this guessing work is that we will design \( N \) to verify that all our guesses were correct. In this case, there are two things we must verify when both scans are finished:
  - We have to verify that the state we reach at the end of scanning \( x \) is equal to the \( m \) we guessed as we started scanning \( y \). This verifies our machine’s guess of \( m \).
  - We need to verify that the scan of \( y \) from state \( m \) ends in a final state. This verifies that the machine guessed the letters of \( y \) correctly.
- To make all this formal, we have to specify the 5 components of an NFA.
  - The new machine will use the same alphabet, \( \Sigma \), as the machine it will simulate.
- We need to be able to remember which state each of our scans is in while also remembering the state \( m \) we guessed at the beginning so that we can verify it at the end. Therefore, our states must hold three states of the simulated machine so \( Q \times Q \times Q \subset Q_N \). The first state in each triple will hold the state of the scan of \( x \), the second state will be the value of \( m \), and the third state will be the current state of the scan of \( y \).
- As our start state we will introduce a new state \( s_N \) from which \( \epsilon \)-transitions can be used to make our guess of \( m \).
- To be successful, our simulation must reach a state where the scan of \( x \) ends in \( m \) and the scan of \( y \) ends in a final state of \( D \). So \( F_N = \{(m, m, f) \mid m \in Q, f \in F\} \).
- The initial guess of \( m \) is accomplished by including the following \( \epsilon \)-transitions in the definition of \( \delta \): 
  \[
  \delta(s_n, \epsilon) = \{(s, m, m) \mid m \in Q\}
  \]
- Finally, the simulated scan proceed using the transitions 
  \[
  \delta((s_x, m ,s_y), x_i) = \{(\delta(s_x, x_i), m, \delta(s_y, y_i)) \mid y_i \in \Sigma\}
  \]
Here, \( s_x \) refers to the state of the simulated version of the machine scanning the actual input that corresponds to “\( x \)” in the definition of \( L_{1,2} \), \( s_y \) is the state of the simulated scan of the guessed string \( y \). \( x_i \) is the next input symbol. \( y_i \) is the next symbol guessed to be in the string \( y \).