Announcements

1. Homework 1 is due today.

Quick Review

1. Last time you worked in groups to describe DFAs that recognized several languages:
   - Binary strings representing binary numbers divisible by 3.
   - Binary strings that form valid BCD encodings of decimal numbers.
   - Binary strings that form valid Huffman encodings relative to a certain Huffman tree.

2. We did not have time to really discuss the DFAs you designed last class, so we will talk about them now. However, I included my notes for that discussion in the notes for the preceding lecture as a way to make the material available to you before the homework due today had to be submitted.

3. One item I want to talk about today that I did not include in the previous notes is ways in which the DFA for the Huffman tree can be simplified.
   - The “obvious” approach to building a FSM for a given Huffman tree is to create a machine with one state for each node of the tree keeping the labels on the tree’s edges as labels on the machine’s transitions and turning all the leaves into final states since if a leaf is reached we know we have a valid input. Finally, we add edges from all the leaves back to the states corresponding to seeing the first 0 or 1 of a new symbol’s encoding.
   - Fortunately, there is an easy way to simplify this machine. The transitions out of all of the final states in this machine are the same! On 0 they all go to state 0 and on 1 they all go to state 1. Basically, once the machine enters one of its final states its future behavior does not depend on which of the final states it entered. We can take advantage of this by merging all of the final states into one state. Below, we show the result of this change without the edges that loop back to the 0 and 1 states.
   - Sorry, but we are not done yet! If you look at the 00 and 01 states
of this new machine you might notice that the transitions out of these two states are identical just as the transitions out of all of the final states of the original machine were. That means we can combine these states to get:

![Diagram of DFA]

- We are getting close, but once again if you look at this machine you will find another pair of similar states. The 0 and 1 states have identical transitions. So, we can merge them too now. Doing this and adding the looping transition back from the final state to the merged 0 and 1 state we get:

![Diagram of DFA]

- Looking back at the original code tree, you might now notice the property of the code that makes all these simplifications possible.

All of the two bit codes have a 1 in the second bit and all of the three bit codes have a 0 in this position. By just checking the second bit of each code the machine can determine its length and this is all it really needs to check for valid codes.

### Formalizing DFAs

1. Now it’s time to develop a mathematical formalism for deterministic finite automata. This will enable us to reason more broadly about their properties.

**Definition.** A DFA is a five tuple \( D = (Q, \Sigma, \delta, s, F) \) where:

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

2. Using this notation, we can give a formal description of our machine that recognizes even binary numbers:

- \( Q = \{ e, o \} \)
- \( \Sigma = \{ 0, 1 \} \)
- \( \delta : \begin{array}{cc} 0 & 1 \\ e & e \ o \\ o & e \ o \end{array} \)
- \( s = o \)
- \( F = \{ e \} \)

### Families of Languages and Machines

1. An advantage of this formal definition of a DFA is that it allows us to describe and reason about collections of similar DFAs rather than needing to draw a diagram for a specific DFA.
2. Through our examples, we have seen that the set of strings that represent binary numbers that are even is regular and that the language of binary numbers that are divisible by 3 is regular. We might well speculate that the language \( L_{\text{DivN}} = \{ w \mid w \in \{0, 1\}^* \text{ and } w \text{ encodes a number divisible by } N \} \) is regular for all values of \( N \).

3. To show that all these languages are regular, we will describe a family of DFAs, \( M_{\text{DivN}} \) such that \( L(M_{\text{DivN}}) = L_{\text{DivN}} \). In particular, we define
\[
M_{\text{DivN}} = (Q_N, \{0, 1\}, \delta_N, m, 0)
\]
where:
\[Q_N = \{ m, 0, 1, \ldots, N-1 \}\]
\(\delta_N\) is defined by
- \(\delta_N(m, 0) = 0\)
- \(\delta_N(m, 1) = 1\)
- \(\delta_N(i, d) = (2^i + d) \mod N \) for \(i \in \{0, 1, \ldots, N-1\}\)

4. You might recognize that the machine presented as a solution to one of our class exercises is \( M_{\text{Div3}} \):

5. When giving a formal description of a machine, it is useful to take time to think about various ways to describe the state set. Constructing a state set out of tuples of values including subsets of the integers and booleans can also make it possible to described the transition function \(\delta\) clearly and concisely using the well know operations on integers and booleans.

- Hopefully, it was evident that we could not have included the line
  \(\delta_N(i, d) = (2^i + d) \mod N \) for \(i \in \{0, 1, \ldots, N-1\}\)
in our definition of \( M_{\text{DivN}} \) if we had defined the state set for this machine as \(Q = \{ \text{none, divisible, remainder is 1, remainder is 2} \}\).
- Another example that illustrates how useful it can be to include subsets of the integers in your state set is the BCD example we considered.
- We can build an FSA to solve this problem by keeping track of the value of the prefix of the group of four bits current being scanned:
• We could take this idea even farther and replace “good” by 10 states labeled with the values 0 through 9 and “bad” with 6 states labeled with 10 through 15.

• The transitions out of each of the states in this diagram depend on the value written in the center of the state and the column in the diagram in which the state occurs. The column each state falls in corresponds to the position the machine is up to in the group of four digits currently being scanned.

• Based on this observation, we can build a state space out of pairs of integers. The first integer in each pair will encode the column each state in the diagram appears in, the second integer will encode the value of the prefix of the group of four digits that has been scanned so far.

• Given this approach, we can define \( M_{BCD} = (Q, \Sigma, \delta, s, F) \) where:
  - \( \Sigma = \{0, 1\} \)
  - \( Q = \{0, 1, 2, 3, 4\} \times \{0...15\} \)
  - \( s = (0, 0) \)
  - \( F = \{4\} \times \{0, 9\} \)
  - \( \delta((c, v), i) = (c + 1, 2v + i) \), if \( c < 4 \)
  - \( \delta((c, v), i) = (1, i) \), if \( c = 4 \) & \( v < 10 \)
  - \( \delta((c, v), i) = (c, v) \), if \( c = 4 \) & \( v \geq 10 \)

**Formalizing L(M)**

1. Earlier we gave simple rules for evaluating a string with respect to the diagram of a finite automaton. Now we can give a formal definition of how to decide if an automaton accepts or rejects a string.

   Our text defines the notion that a string belongs to the language of a DFA in terms of the existence of a sequence of states related to the sequence of symbols in the string and the machine’s definition in ways that reflect the intent to capture transitions with the \( \delta \) function.

   Definition: We say that a FSA \( M = (Q, \Sigma, \delta, s, F) \) accepts a string \( w \) and write \( w \in L(M) \) if and only if for some sequences \( (w_1, \ldots, w_n) \) and \( (q_0, \ldots, q_n) \) where \( w_i \in \Sigma \) and \( q_i \in Q \):
   - \( w = w_1w_2\ldots w_n \)
   - \( q_0 = s \)
   - \( q_i = \delta(q_{i-1}, w_i) \) for \( 1 \leq i \leq n \)
   - \( q_n \in F \)

2. With \( \hat{\delta} \) in hand we can give a definition of acceptance:

   **Definition** A string \( w \in \Sigma^* \) is accepted by a DFA \( D \) if and only if \( \hat{\delta}(s, w) \in F \).

3. The language of a DFA \( D \) follows naturally:

   **Definition** \( L(D) = \{ w \mid w \in \Sigma^* \text{ and } \hat{\delta}(s, w) \in F \} \)