Announcements

1. Homework 5 will appear soon. Due next Friday.

2. 10/23 will be midterm week.

Minimization of DFAs

1. Given that we now know that for any regular language there is a DFA of size equal to the index of the language it recognizes, we would like to have a way to algorithmically find this DFA given any precise description of the language (i.e., a DFA, an NFA, or a regular expression).

2. Given a regular expression, we can construct a NFA for the language using the constructions embedded in the proofs that regular languages are closed under the union, concatenation and closure.

3. Given an NFA, we can build an equivalent DFA using the subset construction presented earlier.

4. All we need is a way to convert a non-minimal DFA into one of minimal size (or to realize that the one we started with was already minimal).

5. Recall that we used ad hoc techniques to “minimize” one of the deterministic finite automata we considered early in the course:

   - In lecture 3, I asked you to consider how to build a DFA that would recognize binary strings that were valid encodings of a message using the Huffman code associated with a given Huffman tree. The obvious machine looked like:

   ![Diagram of a DFA]

   - We then realized that since all the final states in this machine had identical outgoing transitions, we could merge them into one state. This led to further possibilities to merge states that ultimately led to the machine:

   ![Diagram of a minimized DFA]

6. We can precisely specify when two states can be merged by defining state equivalence formally for two states $p, q \in Q$ as:

$$p \approx q \iff \forall w \in \Sigma^*, \hat{\delta}(p, w) \in F \iff \hat{\delta}(q, w) \in F$$
7. It should be clear that this notion of equivalence of states is in fact reflexive, symmetric and transitive. Therefore, it partitions the set of states of a DFA into equivalence classes.

8. If you look at the equivalent states of the originally DFA for the Huffman code example and compare them to the states of the reduced DFA, you will notice that each state of the reduced DFA corresponds to one of the equivalence classes of the original DFA. (Recall that the states of the reduced DFA correspond in turn to the equivalence classes of strings induced by the indistinguishable by L relation induced by the language recognized by the machine.)

9. This suggests a way that we could use the equivalence relation on the states of a DFA to determine the minimal DFA. Namely, if \([q]\) denotes the equivalence class of state \(q\) induced by the state equivalence relation we just defined:

Given \(M = (Q, \Sigma, \delta, s, F)\) define

\[M' = (\{[q] \mid q \in Q\}, \Sigma, \delta', [s], \{[f] \mid f \in F\})\]

where \(\delta'([p], x) = [\delta(p, x)]\).

10. As in the proof of the Myhill-Nerode theorem, we should be careful to verify that \(\delta'\) is well defined, behaves as desired, and that the set of final states is appropriate. We won’t.

11. Instead, we will consider an algorithm that determines which states are equivalent to one another (by actually determining which states are not equivalent to one another).

12. The basis of the algorithm is a somewhat recursive definition of not being equivalent. The base case is basically

\[p \not\equiv q \text{ if } p \in F \iff q \in F\]

and the recursive clause is

\[p \not\equiv q \text{ if } \exists w \in \Sigma^*, \delta(p, w) \not\equiv \delta(q, w)\]

13. The mechanics of the algorithm use a table in which we record all pairs of states we can identify as non equivalent. Each entry in the table reflects our knowledge of the relationship between the states at the top of its column and the right end of its row. For our example machine, the table starts out like this (with names like oh! and eee used to make it easy to distinguish the states for empty and zero from those for the letters O and E).

14. The first step is to use the basis step described above to realize that all final states are not equivalent to all non-final states. We record this by putting big X’s in all of the cells in the table for such pairs.
15. Next, we use the recursive step over and over again for different pairs of states that still appear to be equivalent restricting our attention to strings \( w \) of length 1. For example:

- At this point in our table, the entry for the pair of states \( e, 0 \) is empty because these state might still be equivalent:

\[
\begin{array}{ccccccc}
\text{e} & ? & 0 & 1 & 00 & 10 & \text{oh!} \\
X & X & X & X & X & \text{gee} & \text{eee} \\
X & X & X & X & \text{oh!} & \text{gee} & \text{eee} \\
X & X & X & X & \text{oh!} & \text{gee} & \text{eee} \\
X & X & X & X & \text{oh!} & \text{gee} & \text{eee} \\
X & X & X & X & \text{oh!} & \text{gee} & \text{eee} \\
X & X & X & X & \text{oh!} & \text{gee} & \text{eee} \\
\end{array}
\]

- Looking back at the state diagram, we can see that on input 0, \( \delta(e, 0) = 0 \) and \( \delta(0, 0) = 00 \). Since the entry in our table for this pair of destinations \( (0, 00) \) is still empty, these states might be equivalent, so it would still appear that \( e \) and \( 0 \) might be equivalent.

- On the other hand, on input 1, \( \delta(e, 1) = 1 \) and \( \delta(0, 1) = \text{oh!} \). The entry for the pair of states \( (1, \text{oh!}) \) in our table already has an \( X \) in it indicating we know these states are not equivalent. Therefore, we can conclude that \( e \) and \( 0 \) are not equivalent and record this fact with a new \( X \) in our table.

16. We then continue methodically (we will go left to right and top to bottom) through the table considering all of the unmarked pairs:

\( \text{e,1} \) Since \( \delta(e, 0) = 0 \) and \( \delta(1, 0) = 10 \) and the pair \( (0, 10) \) is still unmarked in our table, we make no changes.

However, \( \delta(e, 1) = 1 \) and \( \delta(1, 1) = \text{gee} \) and the states \( 1 \) and \( \text{gee} \) are known not to be equivalent, so we get to put another \( X \) in for \( e, 1 \):

\( \text{0,1} \) Since \( \delta(0, 0) = 00 \) and \( \delta(1, 0) = 10 \) and the pair \( (0, 10) \) is still unmarked in our table, we make no changes.

Similarly, \( \delta(0, 1) = \text{oh!} \) and \( \delta(1, 1) = \text{gee} \) and the states \( \text{oh!} \) and \( \text{gee} \) are still unmarked so we make no changes.
It is important to note, however, that in both cases, we are deciding whether the two states the machine would move into are not equivalent by checking to see if their entry in our table contains an X before we have even gotten to that entry. If, when we eventually process those entries we discover they should have X’s, we will need to reconsider the pair (0,1). We won’t do this by specially reconsidering (0,1). Instead, we will make an additional pass over all table entries that are still blank after the first pass.

(e,00) Since $\delta(e, 0) = 0$ and $\delta(00, 0) = eee$ and the pair (0, ee) is marked as non-equivalent, we get to mark (e,00)

Similarly, $\delta(e, 1) = oh!$ and $\delta(00, 1) = gee$ and the states oh! and gee are still unmarked so we make no changes.

17. Continuing to consider every empty cell in the table in the same way until we reach (i,n), we eventually get the following:

18. At this point, as mentioned above, we need to reconsider all of the blank cells because when we considered them on the first pass we might have based our decision not to mark them on cells that we had not yet processed. In this case, on the second pass, we will discover that nothing actually changes. In general, we would keep making passes until nothing changes during one complete pass.

19. The information in the table justifies the reductions we made using our ad hoc methods. It indicates that states 0 and 1 can be merged as can 00 and 10. It also says that all of the final states are equivalent and can be merged. Thus, the reduced machine will look like:
Informal Formal Grammars

1. Basic introduction to context-free grammars.

- A grammar is a specification for a language that is composed of rules like the following which (informally) says that anything composed of a variable followed by an assignment operator and an expression is a valid statement.

  \[ <\text{stmt}> \rightarrow <\text{var}> = <\text{expr}> \]

  In this particular notation for writing grammars (a variant of BNF or Backus Normal Form (or Backus Naur Form), it was originally used to describe the programming language Algol 58 back in 1958!), the symbols in angle brackets denote classes of syntactic phrases and are called variables (or non-terminals).

  The symbols not in angle brackets denote components of strings in the language being described. They are called terminals.

  The rules are sometimes called productions.

- The syntactic phrases in most interesting grammars are frequently defined recursively (either directly or indirectly).

  \[ <\text{stmt}> \rightarrow \text{while} \ (<\text{expr}> ) <\text{stmt}> \]

- When used as a notation for specifying languages, various notational conveniences are employed (such as using a \( \mid \) to abbreviate a set of rules that would have the same phrase type on the left hand side).

  \[ <\text{stmt}> \rightarrow <\text{var}> = <\text{expr}> \mid \text{while} \ (<\text{expr}> ) <\text{stmt}> \]

- Given this we can give a complete set of rules describing a simple language capturing some of the syntax of common control structures and assignment statements:

  \[ <\text{stmt}> \rightarrow <\text{var}> = <\text{expr}> \mid \text{if} \ (<\text{expr}> ) <\text{stmt}> \mid \text{if} \ (<\text{expr}> ) <\text{stmt}> \text{ else } <\text{stmt}> \]

  | \textbf{while} \ (<\text{expr}> ) <\text{stmt}>

  \[ <\text{expr}> \rightarrow <\text{var}> \]

  \[ <\text{var}> \rightarrow x \mid y \mid z \]

- We can view the rules of a grammar as specifications of relationships between sets. For example,

  \[ <\text{stmt}> \rightarrow <\text{var}> = <\text{expr}> \]

  implies

  \[ <\text{stmt}> \subset <\text{var}> = <\text{expr}> \]

- It is more common to view the rules of a grammar as replacement rules. That is, given a string containing terminals and variables, we can replace any of the variables with the right hand side of any rule with the variable’s name on the left.

- For example, the grammar above lets us write:

  \[ <\text{stmt}> \Rightarrow \text{if} \ (<\text{expr}> ) <\text{stmt}> \]

  \[ \Rightarrow \text{if} \ (<\text{var}> ) <\text{stmt}> \]

  \[ \Rightarrow \text{if} \ (x) <\text{stmt}> \]

  \[ \Rightarrow \text{if} \ (x) <\text{var}> = <\text{expr}> \]

  \[ \Rightarrow \text{if} \ (x) y = <\text{expr}> \]

  \[ \Rightarrow \text{if} \ (x) y = <\text{var}> \]

  \[ \Rightarrow \text{if} \ (x) y = z \]

2. Grammars of this sort are called context-free grammars.

Slightly More Formal Grammars

1. The notation shown above is typical when context-free grammars are actually used to describe programming languages.

2. When studied as an example of a notation for describing languages from a theoretical standpoint, a slightly different notation is typically used.

- Instead of words in angle brackets, variables are typically denoted using capital letters.
3. In addition, theoretical studies typically focus on sillier languages. For example, consider the following grammar:

\[ \begin{align*}
E & \rightarrow E \\
E & \rightarrow 0E \\
E & \rightarrow 1D \\
D & \rightarrow 1E \\
D & \rightarrow 0D
\end{align*} \]

- Can you tell what language this grammar describes?
- It might help to consider a derivation like:
  \[ \begin{align*}
E & \Rightarrow 0E \\
0E & \Rightarrow 00E \\
00E & \Rightarrow 001D \\
001D & \Rightarrow 0010D \\
0010D & \Rightarrow 00101E \\
00101E & \Rightarrow 00101
\end{align*} \]
- In general, the grammar describes
  \[ L_{\text{Parity}} = \{ w \in \{0, 1\}^* \mid \text{the number of 1s in } w \text{ is even} \} \]
- We encountered this language weeks ago as an early example of a regular language. Hence, we now know that context-free grammars can describe at least some regular languages.

4. Recall the language \( L_{EQ} = \{ 1^n = 1^n \mid n \geq 0 \} \). This was probably the simplest example of a language that is not regular that we discussed. Think about how one could describe this language with a context-free grammar.

- The string “=” is in the language, so we would include the production \( E \rightarrow = \).
- If we have a string in the language, we can form another string that belongs in the language by adding a one to the front and another 1 to the end. The production \( E \rightarrow 1E1 \) captures this.
- Together, these two productions describe all the strings in the language!
- We can see, therefore that at least in some cases, context-free grammars are more expressive than DFAs or regular expressions.