Announcements

1. Homework 3 due Friday.

Generalized Nondeterministic Finite Automata

1. The book presents an algorithm that translates the description of a DFA into a regular expression describing the same language. The existence of and correctness of this algorithm proves that all regular languages are described by some regular expression.

2. Last class, I completed an example of converting a DFA into an equivalent regular expression using a process based on the same intuition behind the algorithm in the text. My approach differed from the text in two important ways.

   • First, rather than waiting to add a single, separate final state when I was almost done with the presentation in our last class, the algorithm starts by adding both a new start state and a new final state and connecting these new states to the original start state and final states with epsilon-transitions. This ensures that a process of repeatedly removing one of the original states at a time will eventually lead to a machine with one start state, one final state and an edge between them labeled with a regular expression describing the language of the original machine.

   • Second, so that they don’t have to handle the merging of edges as a special case, they immediately add edges between all states not connected directly by edges (except for their new start and final state) that are labeled with the regular expression $\emptyset$. They can get away with this because such edges act as if they are not there. Adding these edges simplifies the formal description of the algorithm. In class (and when doing your homework), it is not worth adding these edges. Just merge or add edges when appropriate.

3. With this in mind, let’s solidify our understanding of the algorithm by again reducing the “multiples of 3” machine. Luckily, we can make the experience different by removing states in a different order states in a different order. Also, I will ask you to help by telling me which edges I will have to add or augment when I remove an existing state and by telling me what the labels on these edges should be.

4. As our first step, rather than waiting until we get in trouble, we immediately augment the machine with a new start state and a new final state. We add epsilon-transitions from the new start state to the old one and from all old final states to the new one.

5. Now, instead of removing state 2, let’s try removing state 1 first. This will require adding edges from m to 2, from 2 to 0, and from 0 to 2. We will also have to update the label of the edges from m to 0, and the loops from 0 to itself and from 2 to itself. The result looks like:
6. Next we will remove state 2. This requires updating the labels of the edges from m to 0 and from 0 to itself to reflect the paths between these sources and destinations that currently pass through 2.

7. Finally, since the only edge from start to m is an epsilon-transition, it is clear that we can remove both states m and 0 to obtain:

8. My hope is that this practice gives you a clear enough understanding of how to use GNFA's to extract a regular expression that describes the language of a DFA. The book gives a more formal presentation (almost a proof). You should reread (or read) that section now to solidify your understanding and convince yourselves that the algorithm can be applied to any DFA.

**Not Regular ≠ Irregular**

1. In case you didn’t notice, I want to point out that we obtained a different regular expression by performing this sequence of state eliminations than we did last class. Eliminating 2 then 1 then 0 and m gave us:

   \[(0 \cup (1(01^*0)^1))(0 \cup (1(01^*0)^1))^*\]

   Eliminating 1 then 2 then 0 and m gave us:

   \[(0 \cup 11 \cup 10(1(00)^*01))(0 \cup 11 \cup 10(1(00)^*01))^*\]

   Hopefully, these two regular expressions describe the same sets!

2. After spending hours making up the slides showing how to extract a regular expression for a language from a DFA that recognizes the language, I could not help thinking that it would be nice to have some sort of “regular expression checker” that would tell me for sure that two regular expressions actually do describe the same languages.
3. If you think about it, you will realize that what I really wanted was a **decider** for the language:

\[ L_{EQ-RE} = \{ e = e' \mid e \& e' \text{ are regular expressions over } \Sigma \text{ and } L(e) = L(e') \} \]

4. This language is a bit more interesting than most of the examples we have been talking about so far this semester. Certainly, it would be harder for you to write a program that decided whether an input belonged to this language than it would be to decide if a binary string represented a number divisible by 3.

5. If this problem does not impress you, consider the similar problem for a language somewhat richer than the language of regular expressions:

\[ L_{EQ-Java} = \{ j = j' \mid j \& j' \text{ are Java programs that behave identically} \} \]

Those in the know might even suspect that writing a program to recognize strings that belong to this language is more than difficult.

6. For now, let’s stick to regular languages and ask whether a set like

\[ L_{EQ-RE} = \{ e = e' \mid e \& e' \text{ are regular expressions over } \Sigma \text{ and } L(e) = L(e') \} \]

is regular.

7. In fact, let’s start with something even easier. In your last homework assignment I mentioned that \{a + b = c \mid a, b, c \in \{0, 1\}^* \text{ and the sum of the numbers represented by } a \text{ and } b \text{ in binary notation is the number represented by } c \} \} was not regular. Let’s consider an even simpler representation of addition:

\[ \{ a + b = c \mid a, b, c \in \{1\}^* \text{ and the sum of the numbers represented by } a \text{ and } b \text{ in unary notation is the number represented by } c \} \}

8. That is, we would like to determine whether the language

\[ L_{UnaryAdd} = \{ 1^a + 1^b = 1^c \mid 1^k \text{ refers to a string of } k \text{ 1s and } a + b = c \} \]

is regular.

---

**Getting Loopy**

1. One approach to showing that a particular language is not regular involves recognizing that strings of sufficient length will encounter loops of states as they are processed by a DFA. I want to make this notion very concrete for you before using it in a more abstract way to show languages are not regular.

   - Consider the following machine (which happens to be the “5” version of the “binary numbers that are multiples of N” DFA we presented using our formal notation earlier).

   ![DFA Diagram]

   - What I would like to do with this machine is look for simple cycles. To be more precise I would like to look for strings of varying lengths that take the machine from start state to final state while repeating at least one but as few as possible of the states in the path.
   - Here are some examples:

<table>
<thead>
<tr>
<th>input</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>( \epsilon \rightarrow 0 \rightarrow 0 )</td>
</tr>
<tr>
<td>11001</td>
<td>( \epsilon \rightarrow 1 \rightarrow 3 \rightarrow 1 \rightarrow 0 )</td>
</tr>
<tr>
<td>101101</td>
<td>( \epsilon \rightarrow 1 \rightarrow 2 \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 0 )</td>
</tr>
<tr>
<td>111101</td>
<td>( \epsilon \rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 0 )</td>
</tr>
</tbody>
</table>
I haven’t included inputs that caused simple cycles of length 5 and 6 because there are no such simple cycles in this machine’s state graph.

The interesting thing about the sorts of loops in the state diagram we have been looking at is that they correspond to substrings in the input that can be repeated (or omitted) without changing the final state of the corresponding path. This means from each of our examples, we can generate an infinite set of inputs that lead to the same final state by just “starring” the input subsequence that leads the machine through the loop. The following table show how this would be done for the examples discussed above.

<table>
<thead>
<tr>
<th>input</th>
<th>related strings</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00*</td>
</tr>
<tr>
<td>11001</td>
<td>1(10)*01</td>
</tr>
<tr>
<td>101101</td>
<td>1(011)*01</td>
</tr>
<tr>
<td>1111101</td>
<td>1(1111)*01</td>
</tr>
</tbody>
</table>

An alternate question we can ask this machine is if there are inputs of various lengths that would cause the machine to visit states without encountering any loop (i.e., any repeated states).

Here are some examples:

<table>
<thead>
<tr>
<th>input</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0→1</td>
</tr>
<tr>
<td>11</td>
<td>0→1→3</td>
</tr>
<tr>
<td>111</td>
<td>0→1→2</td>
</tr>
<tr>
<td>1111</td>
<td>0→1→2→0</td>
</tr>
<tr>
<td>01110</td>
<td>0→1→3→2→0</td>
</tr>
</tbody>
</table>

Note that the last example visits every state in the machine. That means that any longer input sequence must visit more states than there are in the machine. Thus, for 6, 7 and any larger length inputs, a loop in the states must occur.

Addition is too Hard to be Regular

1. I suggested that to make progress on thinking about how to show a language was not regular it would be best to start with a language that was very simple:

\[ L_{\text{UnaryAdd}} = \{ 1^a + 1^b = 1^c | 1^k \text{ refers to a string of } k \text{ 1s and } a + b = c \} \]

is regular.

2. We just saw that if a DFA has n states then it must encounter a loop in its state graph when processing any input of length greater than N. We can use this property to see that \( L_{\text{UnaryAdd}} \) is not regular.

3. If this language were regular, then there would be some DFA M such that \( L_{\text{UnaryAdd}} = L(M) \). if \( M = (Q, \Sigma, \delta, s, F) \), and \( n = |Q| \) then the input \( 1^{2n} + 1 = 1^{2n+1} \):

a) must lead to a final state in \( M \), and

b) must lead \( M \) through at least one cycle in its set of states (since its length requires more states be visited than there are distinct states).

4. In case the notation \( 1^{2n} + 1 = 1^{2n+1} \) isn’t sufficiently illuminating, we can show the structure of the input we have in mind as:

<table>
<thead>
<tr>
<th>input</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>111...1111</td>
<td>+1 = 111...1111</td>
</tr>
<tr>
<td>length</td>
<td>2n</td>
</tr>
</tbody>
</table>

5. The “cloudy” diagram below gives some names to the “dimensions” of the loop we know the machine must encounter. In particular we are assuming that the first loop begins after \( k \) 1’s have been read and that it is completed after \( l \) steps implying that the number of 1s in the input that drive the machine around the loop is \( l - k > 0 \).
This suggests another way we can partition the input:

<table>
<thead>
<tr>
<th>input</th>
<th>11...11</th>
<th>11...11</th>
<th>11...11</th>
<th>+1 =</th>
<th>111...111</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>k</td>
<td>l-k</td>
<td>2n-l</td>
<td>3</td>
<td>2n+1</td>
</tr>
</tbody>
</table>

7. Now, the key observation is that since the string of \( l-k \) ones leads the machine \( M \) through a loop, we can describe an infinite set of strings that must belong to \( L(M) \) by “starring” this sequence of 1’s.

<table>
<thead>
<tr>
<th>input</th>
<th>11...11</th>
<th>(11...11)*</th>
<th>11...11</th>
<th>+1 =</th>
<th>111...111</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>k</td>
<td>l-k</td>
<td>2n-l</td>
<td>3</td>
<td>2n+1</td>
</tr>
</tbody>
</table>

8. Since the closure (*) means we can either repeat or delete (i.e., repeat 0 times) a substring, this would allow us to conclude that the string

\[
\text{input} | \begin{array}{c} 11\ldots11 \\ 11\ldots11 \\ +1 = \\ 111\ldots111 \\ \end{array} \\
\text{length} | \begin{array}{c} k \\ l-k \\ 2n-l \\ 3 \\ 2n+1 \\ \end{array} \\
\]

which is just

\[
\text{input} | \begin{array}{c} 11\ldots11 \\ +1 = \\ 111\ldots111 \\ \end{array} \\
\text{length} | \begin{array}{c} 2n-(l-k) \\ 3 \\ 2n+1 \\ \end{array} \\
\]

which is \( 1^{2n-(l-k)} + 1 = 1^{2n+1} \in L(M) \). This, however, should only be true if \( 2n - (l - k) + 1 = 2n + 1 \) which would imply \( l - k = 0 \) contrary to our conclusion that there must be a cycle of length 1 or greater in the path followed processing a string longer than the number of states in the machine \( M \).

9. This contradiction allows us to conclude that our assumption that \( L(M) = L_{UnaryAdd} \) was false, so \( L_{UnaryAdd} \) must not be regular.

**The Pumping Lemma (for regular languages)**

1. We can generalize the partitioning we performed on \( 1^{2n} + 1 = 1^{2n+1} \) in a way that leads to an understanding of a more general result that can be used to show that certain languages are not regular.

- The key to our discussion of \( 1^{2n} + 1 = 1^{2n+1} \) was the subsequence of 1s in \( 1^{2n} \) that could be repeated. That is, there were really three key parts to our partition as shown below:

<table>
<thead>
<tr>
<th>input</th>
<th>11...11</th>
<th>(11...11)*</th>
<th>11...11 +1 =</th>
<th>111...111</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>k</td>
<td>l-k</td>
<td>2n-l</td>
<td>3</td>
</tr>
</tbody>
</table>

- Better yet, rather than counting all the digits so carefully, we can just name subparts as in:

<table>
<thead>
<tr>
<th>input</th>
<th>11...11</th>
<th>(11...11)*</th>
<th>11...11 +1 =</th>
<th>111...111</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>x</td>
<td>y</td>
<td>z</td>
<td></td>
</tr>
</tbody>
</table>

- Now, we can summarize the logic behind our argument by saying that for any string \( w \) that is long enough, we must be able to write \( w = xyz \) in such a way that \( y \) corresponds to a string that leads \( M \) through a loop and therefore, it must be the case that \( xy^iz \in L \) (or equivalently \( xy^iz \in L, i \geq 0 \)).

2. This generalization of our approach to \( L_{UnaryAdd} \) is encapsulated in the FAMOUS Pumping Lemma:

**Lemma:** Suppose \( L \) is a regular language. Then there exists a positive integer \( p \) such that any string \( s \in L \) with length at least \( p \) may be partitioned into \( s = xyz \) where
3. It may at times be useful to use a slightly different statement of this result:

**Lemma:** Suppose $L$ is a regular language. Then there exists a positive integer $p$ such that any string $s \in L$ with length at least $p$ may be partitioned into $s = xyz$ where

(a) $|y| > 0$
(b) $|xy| \leq p$
(c) $xy^iz \in L$, for all $i \geq 0$.

4. The proof of this lemma is in the text. For not, we will just assume the lemma is true and work to make sure we know how to use it to show that languages are not regular.

**Pumping Iron**

1. I started with unary addition because I had previously mentioned that at least one language encoding addition (binary addition) was not regular in a homework assignment.

2. As our first exercise with the Pumping Lemma, let’s consider an even simpler language.

3. Let’s show that $L_{EQ} = \{1^n = 1^n \mid n \geq 0\}$ is not regular.

   - Using the Pumping Lemma, all we need to do is given a possible $p$ for this language, show how to find a string that cannot be pumped (i.e., a string that turns into strings that don’t belong in $L_{EQ}$ when pumped).
   - Consider $1^p = 1^p$.
   - The pumping lemma requires that we be able to remove or duplicate some substring $y$ which is non-empty ($|y| > 0$) that appears within the first $p$ symbols ($|xy| \leq p$) of $1^p = 1^p$.

   - Any prefix $xy$ of $1^p = 1^p$ of length at most $p$ must contain only 1s. So, $x = 1^k$ and $y = 1^l$ for some $l$ and $k$ such that $l > 0$ and $l + k \leq p$.
   - The Pumping lemma allows us to conclude that if $L_{EQ}$ is regular all strings of the form $1^k(1^l)^*1^{p-(l+k)} = 1^p \in L_{EQ}$. However, $1^k1^{p-(l+k)} = 1^p \notin L_{EQ}$.
   - Therefore, $L_{EQ}$ must not be regular!

4. This brings us back to the example I started with to remind you that decision problems can be interesting.

$$L_{EQ-RE} = \{ e = e' \mid e \ & e' \text{ are regular expressions over } \Sigma \text{ and } L(e) = L(e') \}$$

   - At first this may seem like a very complicated example.
   - Remember, however, that $\{1^n = 1^n \mid n \geq 0\} \subset L_{EQ-RE}$.
   - Using closure properties, we can often focus on such a sub-language to prove a language that contains it is not regular.
   - In this case, consider the language $1^* = 1^*$ (which is clearly regular since we used a regular expression to describe it).
   - The intersection of $L_{EQ-RE}$ with $1^* = 1^*$ is just $L_{EQ}$.
   - If $L_{EQ-RE}$ was regular, then since regular languages are closed under intersection, $L_{EQ}$ would have to be regular.
   - We just showed, however, that $L_{EQ}$ is not regular.
   - So, $L_{EQ-RE}$ must not be regular.
   - The lesson is that you should not work too hard.