DISCLAIMER: These notes are not necessarily an accurate representation of what I said during the class. They are mostly what I intend to say, and have not been carefully edited.

Today’s topics:

1. Converting nondeterministic automata to deterministic automata.
2. Regular expressions, and how to build NFAs from them.

1 From NFAs to DFAs

The most important fact about NFAs is that the languages accepted by them is precisely the regular languages.

A DFA is a NFA in that there are no \( \epsilon \) transitions, and all transitions go to exactly one state.

So we need to show that any NFA has an equivalent DFA that accepts the same states. Recall that an NFA \( N = (Q, \Sigma, \delta, q_0, F) \) accepts a string \( w \) if

- we can write \( w = y_1 y_2 \cdots y_m \), where each \( y_i \in \Sigma \);
- there exists a sequence of states \( r_0 r_1 \cdots r_m \) such that
  - \( r_0 = q_0 \)
  - \( r_m \in F \)
  - \( r_{i+1} \in \delta(r_i, y_{i+1}) \)

The way that we check whether a string is accepted is similar to reachability check on graphs: we create a set of all reachable states after we have examined \( y_1 \cdots y_i \). That is, we successively update

\[
R_{i+1} \leftarrow \cup_{r \in R_i} \delta(r, y_{i+1}).
\]

For example, for the NFA with two states \( Q = \{q_0, q_1\} \), \( F = \{q_0\} \), \( \Sigma = \{0, 1\} \), and transitions

\[
\begin{array}{c|cc}
Q \setminus \Sigma & 0 & 1 \\
0 & \{q_0, q_1\} & \{q_1\} \\
1 & \{q_0\} & \emptyset \\
\end{array}
\]
on the input \( y = 01101 \), we get

\[
\begin{align*}
R_0 &= \{ q_0 \} \\
R_1 &= \{ q_0, q_1 \} \\
R_2 &= \{ q_1 \} \\
R_3 &= \{ q_0 \} \\
R_4 &= \{ q_0, q_1 \} \\
R_5 &= \{ q_1 \}
\end{align*}
\]

And because \( R_5 \cap F = \emptyset \), this NFA does not accept this input.

Formally, the \( R_i \) sets is precisely how we convert this NFA into a DFA. This is because the \( R_i \)s are all members of \( P(Q) \), the power set of \( Q \). That is, the new DFA, \((Q', \Sigma, \delta', q'_0, F')\) has

\[ Q' = P(Q) \]

and transition given by

\[
\delta'(R, a) = \{ \delta(r, a) \mid r \in R \}
\]

for each \( R \in Q' \) and \( a \in \Sigma \). This needs to be modified slightly to take \( \epsilon \) transitions into account. Specifically, for each \( R \subseteq Q \), we define \( E(R) \) as the set of states reachable from \( R \) by traversing \( 0 \) or more \( \epsilon \) transitions. Then the transition that take these into account is:

\[
\delta'(R, a) = \bigcup_{r \in R} E(\delta(r, a)). \tag{1}
\]

Finally the new starting state is

\[ q'_0 = E(\{q_0\}), \]

while the accepting state is all \( R \in P(Q) \) with non-empty intersection with \( F \):

\[ F' = \{ R \in Q' \mid R \cap F \neq \emptyset \}. \]

Going with the example above, we get a DFA with states

\[ \emptyset, \{q_0\}, \{q_1\}, \{q_0, q_1\}. \]

Its starting state is \( \{q_0\} \), and its accepting states are \( \{q_1\}, \{q_0, q_1\} \). Finally, we can work out its transitions using Equation 1.

<table>
<thead>
<tr>
<th>( P(Q) \setminus \Sigma )</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( {q_0} )</td>
<td>( {q_0, q_1} )</td>
<td>( {q_1} )</td>
</tr>
<tr>
<td>( {q_1} )</td>
<td>( {q_0} )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( {q_0, q_1} )</td>
<td>( {q_0, q_1} )</td>
<td>( {q_1} )</td>
</tr>
</tbody>
</table>
2 Regular Expressions

Note that the strings that contain $ab$ can be written as the concatenation of:

1. any arbitrary string,
2. followed by the string $ab$,
3. followed by any arbitrary string.

Each of these building blocks is much easier to build an automaton for.

This then brings us to our next point, which is to decompose regular languages into simpler building blocks. We want to give a way to build a new regular language from two existing ones, $A$, and $B$. The operations that we devise are akin to addition/multiplication/powering.

1. Union: $A \cup B = \{x : x \in A \text{ or } x \in B\}$.
2. Concatenation: $A \circ B = \{xy | x \in A \text{ and } y \in B\}$
3. Star: $A^* = \{x_1 x_2 \ldots x_k | k \geq 0 \text{ and } x_i \in A \forall 1 \leq i \leq k\}$.

In particular, the language described above, $L$ that contains $ab$ as a substring, can be written as

$$\{a,b\}^* \circ \{ab\} \circ \{a,b\}^*.$$ 

Note that the $^*$, or star operation, has priority over the $\circ$ operation.

Furthermore, the $^*$ operation denotes all strings (with possibly 0 length) built from things in this set. Here because we started with $\{a,b\}$, it’s the entire set of strings.

A more interesting example is

$$\{a,ab\}^* = \{\epsilon, a, aa, ab, aab, aba, aaaa, aab, aba, abaa, abab \ldots \}.$$ 

We also need to define the base cases: single character, $\epsilon$, and $\emptyset$. Then a regular expression is formally one of:

1. $a$ for some $a \in \Sigma$.
2. $\epsilon$, the empty string.
3. $\emptyset$, the empty language. Note that this is not the same as the empty string.
4. $(R_1 \cup R_2)$ where $R_1$ and $R_2$ are regular expressions.
5. $(R_1 \circ R_2)$ where $R_1$ and $R_2$ are regular expressions.
6. $(R_1^*)$ where $R_1$ is a regular expression.

In the first three cases, the corresponding languages are $\{a\}$, $\{\epsilon\}$, and $\emptyset$ respectively.

In cases 4 - 6, which are inductive, the languages are formed by performing regular operations on the languages corresponding to $R_1$ and $R_2$ (if $R_2$ is needed).
3 Regular Operations Keep Languages Regular

We will show that the class of regular languages is precisely the languages composable from regular operations using the elements from $\Sigma$.

As a sanity check, a single string can be written as the concatenation of its characters. For example

$$\{abc\} = \{a\} \circ \{b\} \circ \{c\},$$

and a set of strings is simply the union of languages corresponding to single strings.

Formally, we want to verify that the union/concatenation of two regular languages, or the star of a regular language is also regular. For concatenation / star, a major challenge is figuring out, or guessing, where one word ends and the next begins.

Concretely, we will show that every language described by a regular expression can be described by an NFA.

The equivalence between NFA and DFAs (see e.g. Theorem 1.39 and Example 1.41 of textbook) then gives that such languages are also describable by DFAs, and are thus regular.

The overall proof is by induction on the length of the regular expression. Note that the recursive operations $\cup$, $\circ$ and $^*$ all put together shorter expressions. So we can assume that those expressions already correspond to regular languages, and in turn have NFAs that accept them.

The base cases are the three base cases for regular expressions. For these we give explicit constructions:

1. $a$: two states, with one transition from the start to the accepting state corresponding to $a$.
2. $\epsilon$: one starting state that’s also accepting.
3. $\emptyset$: one starting state that’s not accepting.

The inductive case is similar to how we applied regular operations to regular languages. We will work with $N_1$ and $N_2$, the NFAs accepting $L_1$ and $L_2$ respectively.

1. $\cup$: a new ‘super starting state’ with $\epsilon$ transitions to both starting states of $M_1$ and $M_2$.
2. $\circ$: start at $N_1$’s starting state, and add $\epsilon$ transitions from each accepting state of $L_1$ to the starting state of $N_2$.
3. $x^*$: we need to have an ‘extra’ starting state to accept $\epsilon$. For this we create an extra starting state that’s accepting, and has an $\epsilon$ transition to the starting state of $N_1$.

The inductive nature of this proof means we can (and also need to) compose these constructions together. We will work through the example (1.58) in the textbook of $(a \cup b)^*aba$. Example 1.56 is also worth a look as well.