Spring 2015
Lecture 23: State minimization and NFAs

When a user takes a photo, the app should check whether they’re in a national park...
Sure, easy GIS lookup. Gimme a few hours.
...and check whether the photo is of a bird.
I’ll need a research team and five years.

In CS, it can be hard to explain the difference between the easy and the virtually impossible.
Prove that the following grammar generates all strings with matched ():

\[ S \rightarrow SS \mid (S) \mid \varepsilon. \]

Base: \( P(0) \) \( T = \varepsilon \) \( \vdash \)

IH: For some \( k \geq 0 \) and any \( 0 \leq j \leq k \), \( P(j) \) holds.

IS: Want to prove \( P(k+1) \).

Fix \( T \) len \( (T) = k+1 \) has matched ().

\[ A \quad (\text{-}j) \quad B \]

\[ \text{len} \quad (T) = k+1 \quad \text{by IH} \]

\[ s \rightarrow SS \rightarrow \quad \text{by IH} \]

\[ \text{by IH} \]

\[ \rightarrow (A)S \rightarrow \quad \text{by IH} \]

\[ \quad (A)B \]
state minimization

- Many different FSMs (DFAs) for the same problem
- Take a given FSM and try to reduce its state set by combining states
  - Algorithm will always produce the unique minimal equivalent machine (up to renaming of states) but we won’t prove this
1. Put states into groups based on their outputs (or whether they are final states or not)
2. Repeat the following until no change happens
   a. If there is a symbol $s$ so that not all states in a group $G$ agree on which group $s$ leads to, split $G$ into smaller groups based on which group the states go to on $s$
state minimization example

Put states into groups based on their outputs (or whether they are final states or not)
state minimization example

state transition table

<table>
<thead>
<tr>
<th>present state</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>S0</td>
<td>S1</td>
<td>S2</td>
<td>S3</td>
<td>1</td>
</tr>
<tr>
<td>S1</td>
<td>S0</td>
<td>S3</td>
<td>S1</td>
<td>S5</td>
<td>0</td>
</tr>
<tr>
<td>S2</td>
<td>S1</td>
<td>S3</td>
<td>S2</td>
<td>S4</td>
<td>1</td>
</tr>
<tr>
<td>S3</td>
<td>S1</td>
<td>S0</td>
<td>S4</td>
<td>S5</td>
<td>0</td>
</tr>
<tr>
<td>S4</td>
<td>S0</td>
<td>S1</td>
<td>S2</td>
<td>S5</td>
<td>1</td>
</tr>
<tr>
<td>S5</td>
<td>S1</td>
<td>S4</td>
<td>S0</td>
<td>S5</td>
<td>0</td>
</tr>
</tbody>
</table>

Put states into groups based on their outputs (or whether they are final states or not)
Put states into groups based on their outputs (or whether they are final states or not)

If there is a symbol $s$ so that not all states in a group $G$ agree on which group $s$ leads to, split $G$ based on which group the states go to on $s$
Put states into groups based on their outputs (or whether they are final states or not)

If there is a symbol \( s \) so that not all states in a group \( G \) agree on which group \( s \) leads to, split \( G \) based on which group the states go to on \( s \)
Put states into groups based on their outputs (or whether they are final states or not)

If there is a symbol $s$ so that not all states in a group $G$ agree on which group $s$ leads to, split $G$ based on which group the states go to on $s$
state minimization example

Put states into groups based on their outputs (or whether they are final states or not)

If there is a symbol $s$ so that not all states in a group $G$ agree on which group $s$ leads to, split $G$ based on which group the states go to on $s$
Put states into groups based on their outputs (or whether they are final states or not)

If there is a symbol \( s \) so that not all states in a group \( G \) agree on which group \( s \) leads to, split \( G \) based on which group the states go to on \( s \)
state minimization example

Can combine states S0-S4 and S3-S5.

In table replace all S4 with S0 and all S5 with S3.
minimized machine

<table>
<thead>
<tr>
<th>present state</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>S0</td>
<td>S1</td>
<td>S2</td>
<td>S3</td>
<td>1</td>
</tr>
<tr>
<td>S1</td>
<td>S0</td>
<td>S3</td>
<td>S1</td>
<td>S3</td>
<td>0</td>
</tr>
<tr>
<td>S2</td>
<td>S1</td>
<td>S3</td>
<td>S2</td>
<td>S0</td>
<td>1</td>
</tr>
<tr>
<td>S3</td>
<td>S1</td>
<td>S0</td>
<td>S0</td>
<td>S3</td>
<td>0</td>
</tr>
</tbody>
</table>

state transition table
another way to look at DFAs

Definition: The label of a path in a DFA is the concatenation of all the labels on its edges in order.

Lemma: $x$ is in the language recognized by a DFA iff $x$ labels a path from the start state to some final state.
nondeterministic finite automaton (NFA)

- Graph with start state, final states, edges labeled by symbols (like DFA) but
  - Not required to have exactly 1 edge out of each state labeled by each symbol--- can have 0 or >1
  - Also can have edges labeled by empty string $\varepsilon$

- **Definition:** $x$ is in the language recognized by an NFA if and only if $x$ labels a path from the start state to some final state
binary strings that have even # of 1’s or contain the substring 111
three ways of thinking about NFAs

• Outside observer: Is there a path labeled by $x$ from the start state to some final state?

• Perfect guesser: The NFA has input $x$ and whenever there is a choice of what to do it magically guesses a good one (if one exists)

• Parallel exploration: The NFA computation runs all possible computations on $x$ step-by-step at the same time in parallel
Theorem: For any set of strings (language) $A$ described by a regular expression, there is an NFA that recognizes $A$.

Proof idea: Structural induction based on the recursive definition of regular expressions...
regular expressions over $\Sigma$

- **Basis:**
  - $\emptyset, \varepsilon$ are regular expressions
  - $a$ is a regular expression for any $a \in \Sigma$

- **Recursive step:**
  - If $A$ and $B$ are regular expressions then so are:
    - $(A \cup B)$
    - $(AB)$
    - $A^*$
• Case $\emptyset$:

• Case $\varepsilon$:

• Case $a$:
- Case $\emptyset$:
  - Case $\varepsilon$:
    - Case $a$:
• Suppose that for some regular expressions $A$ and $B$ there exist NFAs $N_A$ and $N_B$ such that $N_A$ recognizes the language given by $A$ and $N_B$ recognizes the language given by $B$
Case \((A \cup B)\):
Case \((A \cup B)\):
Case \((AB)\):

\[ N_A \rightarrow \epsilon \rightarrow N_B \]
Case (AB):
Case A*
Case A*