Summary

Regular expressions are useful for describing tokens of programming languages. Convert reg expr to DFA; DFA is easy to implement. We have noted: sometimes the DFA obtained is much larger than necessary. (Sometimes a reg expr necessarily has a much larger equiv DFA, often it doesn’t need to be much larger.) Given a DFA, want to simplify it. It turns out that we can (efficiently) find a simplest possible DFA equivalent to a given DFA.

Identifying a DFA as being too complicated

"over-complicated" means more states than you need to accept the language. Example:

DFA on left has an inaccessible state that can be removed to get equiv DFA on right.

Example

Certainly removal of inaccessible states is necessary for simplification. Not so obvious which states are inaccessible. (try to find them!) Need a general method for removing inaccessible states.
A set of states is inaccessible if

- the initial state is not included
- there are no arcs coming into the set from outside it.

We don’t want to test all sets. Build up collection of accessible states as follows.

**Algorithm**

\[ S = \{ i \} \]

repeat

\[ S = S \cup \{ \text{all states that are reached by an arc coming from a member of } S \} \]

until no extra states were added in last iteration

Claim: All accessible states are found by this procedure.

---

### Merging Indistinguishable States

We are not done when we get rid of inaccessible states. There may be pairs of states that are “indistinguishable”.

The above machine accepts \{ba, aa\}.

All states that are found in this way are certainly accessible. We also need: every accessible state is found. To prove this, prove that:

The new states found in the \( j \)-th iteration are those states reached for the first time with \( j \)-letter words. (and any accessible state is reached via some finite word)

If a state needs (say) a 10-letter word to reach it (from state \( i \)), it must be reached via a state that needed a 9-letter word to be reached.

So, if the 10-th iteration finds a new state, then the 9-th will have found a new state, and so on.

---

**Claim:** states 2 and 3 are really “the same” and can be merged:

This smaller machine accepts the same language.
Another example:

Three indistinguishable states can be replaced by one state:

Two states are indistinguishable if the set of words labelling accepting paths from one state is the same as the set of words labelling accepting paths from the other state. When we have two indistinguishable states, you can remove one and direct its arcs to/from the other. The resulting machine accepts the same language.

Before we show how to check for indistinguishable states, we give the following result (in the handout):

**Theorem**

*If a DFA has no inaccessible states, and no indistinguishable states, then it has a minimal number of states.*

So there’s nothing left to do when you have dealt with inaccessible and indistinguishable states.

So we will continue by first proving the theorem then showing the procedure for merging indistinguishable states, and give an algorithm that uses both of these procedures to minimise automata.
Proof: Suppose $M$ has $n$ states that are accessible and distinguishable.
Suppose $M'$ is equivalent to $M$ but has only $n - 1$ states.
We can find $n$ words that reach each of the $n$ states of $M$
use accessibility here
Two of these words must reach the same state of $M'$.
We can find a suffix for these two words such that $M$ would accept
one but not the other
here we use distinguishability
But both words, with any suffix attached, must reach the same
state in $M'$. So $M'$ cannot accept the same set of words.
\[\square\]

"Heuristic" analysis

Let $L_i$ denote language accepted if you start at state $i$. We can see:

\[
L_{10} = \emptyset
\]
\[
L_9 = \{a, b\}^*\]
(“no $c$’s”)
\[
L_8 = \{a, b\}^*c\{a, b\}^*\]
(strings containing exactly one $c$)
\[
L_6, L_7 = \{a, b, c\}^*\]
(any string)
\[
L_3, L_4 = \{a, b\}^*c\{a, b, c\}^*\]
(strings containing at least one $c$)
so far we have found 2 equivalences...

\[
L_2 = cL_3 \cup \{a, b\}L_8
\]
\[
L_5 = \{a, b\}^*cL_3
\]
\[
L_1 = aL_2 \cup bL_5 \cup cL_3
\]

Are any of the above equivalent?...
General method to determine whether pairs of states are distinguishable

Algorithm

Start with two equivalence classes, $T$ and $Q \setminus T$.

Repeatedly subdivide as follows:

For each equivalence class, separate any pair of states for which some letter of the alphabet labels transitions to distinct equivalence classes.

We may stop when this process gives no more subdivisions.

(Notice that if an iteration of the above doesn’t subdivide an equivalence class, then on further iterations you will be doing the same thing; no point in continuing...)

Equivalence Classes (1)

(To begin with, 2 states are equivalent if the empty string cannot "tell them apart")

$Q \setminus T = \{1, 2, 3, 4, 5, 8, 10\}$ $T = \{6, 7, 9\}$

Look at states $s \in Q \setminus T$. Qn: Can any string of length 1 tell them apart?

<table>
<thead>
<tr>
<th>$s$</th>
<th>$\phi(s, a)$</th>
<th>$\phi(s, b)$</th>
<th>$\phi(s, c)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$Q \setminus T$</td>
<td>$Q \setminus T$</td>
<td>$Q \setminus T$</td>
</tr>
<tr>
<td>2</td>
<td>$Q \setminus T$</td>
<td>$Q \setminus T$</td>
<td>$Q \setminus T$</td>
</tr>
<tr>
<td>3</td>
<td>$Q \setminus T$</td>
<td>$Q \setminus T$</td>
<td>$T$</td>
</tr>
<tr>
<td>4</td>
<td>$Q \setminus T$</td>
<td>$Q \setminus T$</td>
<td>$T$</td>
</tr>
<tr>
<td>5</td>
<td>$Q \setminus T$</td>
<td>$Q \setminus T$</td>
<td>$Q \setminus T$</td>
</tr>
<tr>
<td>8</td>
<td>$Q \setminus T$</td>
<td>$Q \setminus T$</td>
<td>$T$</td>
</tr>
<tr>
<td>10</td>
<td>$Q \setminus T$</td>
<td>$Q \setminus T$</td>
<td>$Q \setminus T$</td>
</tr>
</tbody>
</table>

You can distinguish $\{3, 4, 8\}$ from states $\{1, 2, 5, 10\}$.

Round 1

Now look at members of $T$.

<table>
<thead>
<tr>
<th>$s$</th>
<th>$\phi(s, a)$</th>
<th>$\phi(s, b)$</th>
<th>$\phi(s, c)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
<tr>
<td>7</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
<tr>
<td>9</td>
<td>$T$</td>
<td>$T$</td>
<td>${1, 2, 5, 10}$</td>
</tr>
</tbody>
</table>

$\{9\}$ is distinguished from $\{6, 7\}$.
Equivalence Classes (2)

\{9\}, \{6, 7\}, \{3, 4, 8\}, \{1, 2, 5, 10\}

Try all transitions for members of each class:

<table>
<thead>
<tr>
<th>s</th>
<th>(\phi(s, a))</th>
<th>(\phi(s, b))</th>
<th>(\phi(s, c))</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>{6, 7}</td>
<td>{6, 7}</td>
<td>{6, 7}</td>
</tr>
<tr>
<td>7</td>
<td>{6, 7}</td>
<td>{6, 7}</td>
<td>{6, 7}</td>
</tr>
</tbody>
</table>

– we find nothing new.

\{3, 4, 8\} splits into \{3, 4\}, \{8\}.

Class splits into \{1\}, \{5\}, \{2\}, \{10\} (distinguishable with strings of length 2)
Equivalence Classes (3)

\{1, 5\}, \{2\}, \{3, 4\}, \{6, 7\}, \{8\}, \{9\}, \{10\}

Same again:

\[
\begin{array}{c|ccc}
   s & \phi(s, a) & \phi(s, b) & \phi(s, c) \\
\hline
   1 & \{2\} & \{1, 5\} & \{3, 4\} \\
   5 & \{1, 5\} & \{1, 5\} & \{3, 4\} \\
   3 & \{3, 4\} & \{3, 4\} & \{6, 7\} \\
   4 & \{3, 4\} & \{3, 4\} & \{6, 7\} \\
   6 & \{6, 7\} & \{6, 7\} & \{6, 7\} \\
   7 & \{6, 7\} & \{6, 7\} & \{6, 7\} \\
\end{array}
\]

and we now have:

\{1\}, \{2\}, \{3, 4\}, \{5\}, \{6, 7\}, \{8\}, \{9\}, \{10\}

Another iteration fails to distinguish 3 from 4 or 6 from 7.

Proving that the algorithm for finding distinguishable states is correct:

Any pair of states in distinct equivalence classes are distinguishable. (the algorithm essentially finds a distinguishing string.)

Any pair of states in the same class are indistinguishable. To prove this, prove that at the \(j\)-th iteration, we find pairs of distinguishable states that require a string of length \(j\) to distinguish them. If 2 states need a string of length \(j\) to distinguish them, there must be 2 states that need a string of length \(j - 1\) to distinguish them. Hence the algorithm doesn’t stop prematurely.