Nondeterministic Finite Automata
COMP1600 / COMP6260

Victor Rivera    Dirk Pattinson
Australian National University

Semester 2, 2020
DFA Minimisation

Elimination of equivalent states.
- if two states are equivalent, one can be eliminated

Elimination of Unreachable States
- if a state cannot be reached from the initial state then it can also be eliminated.

Example. $S_3$ not reachable
The Standard Minimisation Algorithm

Main Idea.

- aggregate states into groups (of possibly equivalent states)
- initially, \textit{all} states are possibly equivalent
- \textit{split} a group of possibly equivalent states if we have \textit{evidence} that they are not equivalent.
  - a non-final state is never equivalent to a final state
  - two states are non-equivalent if the transition function takes them into different groups (with the same letter)
- repeat until no more groups can be split.

Realisation.

- The working data structure for the algorithm is a list of lists ("groups") of states
- On each iteration, we test one of the groups with a symbol from the alphabet.
- If we notice differing behaviour, we \textit{split the group}. 
The Algorithm Details

- **Input:** A list containing two “groups”. (a group is represented as a list of states). One group consists of the Final states and the other consists of the non-final states.

- **Data:** The working data structure, $WDS : [[\text{State}]]$, is a list of groups of states. When two states are in different groups, we know they are not equivalent.

- **Loop:** Pick a group, $\{s_1, \ldots s_j\}$ and a symbol, $x$.
  - If the states $\{N(s_i, x) \mid i = 1, \ldots, j\}$ are all in the same group, then the group $\{s_1, \ldots s_j\}$ is not split.
  - If the states $\{N(s_i, x) \mid i = 1, \ldots, j\}$ belong to different groups of $WDS$, then the group $\{s_1, \ldots s_j\}$ should be split accordingly.

- **Continue until** we cannot, by any choice of letter, split any group.
Our Previous Example

Our running example is trivial. The initial split is it.

\[
\begin{align*}
A &: \quad \begin{array}{c}
S_0 \\
S_1 \\
S_2 \\
S_3
\end{array} \\
\begin{array}{c}
\downarrow 0 \\
\downarrow 1 \\
\downarrow 0 \\
\downarrow 1
\end{array} & \rightarrow & \begin{array}{c}
\downarrow 0 \\
\downarrow 1 \\
\downarrow 0 \\
\downarrow 1
\end{array} \\
\Rightarrow & \begin{array}{c}
[[s_0, s_2], [s_1, s_3]] \\
[[s_0, s_2], [s_1, s_3]] \\
[[s_0, s_2], [s_1, s_3]] \\
[[s_0, s_2], [s_1, s_3]]
\end{array}
\end{align*}
\]
Q. What is the language of this automaton? Can you find a simpler automaton with the same language?
Minimisation Step by Step

- initial split: \( \{0, 4\}, \{1, 2, 3\} \)
  - check \( \{0, 4\} \): don’t split
  - check \( \{1, 2, 3\} \):
    - \( S_1 \xrightarrow{a} S_3 \) and \( S_2 \xrightarrow{a} S_4 \) in different group, so split
    - \( S_1 \xrightarrow{b} S_0 \) and \( S_3 \xrightarrow{b} S_3 \) in different group, so split
    - \( S_2 \xrightarrow{a} S_4 \) and \( S_3 \xrightarrow{a} S_3 \) in different group, so split
- next split: \( \{0, 4\}, \{1\}, \{2\}, \{3\} \)
  - check \( \{0, 4\} \): don’t split
  - check \( \{1\}, \{2\} \) and \( \{3\} \): don’t split
- final split \( \{0, 4\}, \{1\}, \{2\}, \{3\} \)
  - as no more splits did occur in the last round
Consider this FSA:

![FSA diagram]

Q. Is it intuitively clear what it does?

Q. Is it a DFA in the sense of our definition?
Is it legal, i.e. a “proper” DFA?

A. It makes sense, but it is \textit{nondeterministic}: A nondeterministic finite automaton (NFA). So not a “legal” DFA, but a specimen of a different breed.

\textbf{Differences} to deterministic automata
- Multiple edges with the \textit{same label} come out of states
- For some states, there is \textit{not an edge} for every token

\textbf{Formally.} NFAs have a transition \textit{relation} rather than a transition \textit{function}.
- transition relation $R(s_1, x, s_2)$ obtains if there’s an $x$-labelled edge from $s_1$ to $s_2$
- there can be \textit{no} $x$-labelled edge between $s_1$ and \textit{any} state
- there can be \textit{many} states $s_2, s_3, \ldots$ that are connected to $s_1$ via an $x$-labelled edge.
Is it clear what it does?

Observations.

- Some states don’t have an outgoing edge with a certain letter, so the NFA can “get stuck”.
- In some states, there’s more than one possible successor state with a certain letter.

Acceptance condition for NFAs given string $\alpha$:

- can get from initial to final state, making the “right” choice of successor state
- without getting stuck

Example. $\alpha = aaabcc$

- need to “look ahead” to make the right choice
- (alternatively, try to backtrack if wrong choice has been made)
DFAs vs NFAs

Key Differences.

- For each state in a DFA and for each input symbol, there is a unique successor state.
- DFAs have a transition function.
- NFAs allow zero, one or more transitions from a state for the same input symbol.
- NFAs have a transition relation.
- An input sequence \( a_1, a_2, \ldots, a_n \) is accepted by a NFA if there exists some sequence of transitions that leads from the initial state to a final state.
Why NFAs?

**Example.** NFAs are simpler.

A NFA recognising strings of letters ending in “man”:
($\Sigma$ is the Latin alphabet)

![Diagram of NFA](image)

**Note.**

- *two* transitions from $S_0$ for the letter “m”
- *no* transition from $S_1$ for (e.g.) the letter “n”
An Equivalent DFA

**Example.** DFAs are (often) more complex.

A DFA that recognises strings of letters than end in “man”.

![DFA Diagram]

- Start state: $S_0$
- Final state: $S_3$
- Alphabet: $\Sigma = \{a, m, n\}$
- Transitions:
  - $S_0$ on $m$ to $S_1$
  - $S_1$ on $m$ to $S_3$
  - $S_1$ on $a$ to $S_2$
  - $S_2$ on $n$ to $S_3$
  - $S_0$ on $\Sigma-\{a, m\}$ to $S_0$
  - $S_1$ on $\Sigma-\{a, m\}$ to $S_1$
  - $S_2$ on $\Sigma-\{m, n\}$ to $S_2$
  - $S_3$ on $\Sigma-\{m\}$ to $S_3$
NFAs: Formal Definition

A Nondeterministic Finite State Automaton (NFA) consists of five parts:

\[ A = (\Sigma, S, s_0, F, R) \]

- an input **alphabet** \( \Sigma \), the set of tokens
- a set of **states** \( S \)
- an “**initial**” state \( s_0 \in S \) (we start here)
- a set of “**final**” states \( F \subseteq S \) (we hope to finish in one of these)
- a **transition relation** \( R \subseteq S \times \Sigma \times S \).

**Aside.** The transition *relation* is what makes the automaton nondeterministic. It can be seen as a function \( \delta : S \times \Sigma \rightarrow \mathcal{P}(S) \), where \( \mathcal{P}(S) \) is the set of subsets of \( S \).
Another Example

Transition Diagram

As a transition table.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>→</td>
<td>S₀  {S₀, S₁}</td>
<td>S₀  {S₀, S₃}</td>
</tr>
<tr>
<td></td>
<td>S₁  {S₂}</td>
<td>S₁  ∅</td>
</tr>
<tr>
<td>◯</td>
<td>S₂  {S₂}</td>
<td>S₂  {S₂}</td>
</tr>
<tr>
<td></td>
<td>S₃  ∅</td>
<td>S₃  {S₂}</td>
</tr>
</tbody>
</table>

Both convey precisely the same information. What is the language of this automaton?
Acceptance for NFAs

**Given.** An NFA $A = (\Sigma, S, F, s_0, R)$. Then $A$ **accepts** a word $w = a_1a_2 \ldots a_n$ (in symbols: $w \in L(A)$) if there exists a sequence of states

$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} s_{n-1} \xrightarrow{a_n} s_n$$

where $s_0$ is the starting state, $s_n \in F$ is an accepting state, and $s \xrightarrow{a} t$ if $(s, a, t) \in R$.

**Aside.** This is like for deterministic automata, the only difference is that for

- **non-deterministic automata** we have $s \xrightarrow{a} t$ if $(s, a, t) \in R$ (that is, the automaton can make a transition)
- **deterministic automata** we have $s \xrightarrow{a} t$ if $N(s, a) = t$ (that is, the automaton makes the transition)
Eventual State Relation for NFAs

Basic Idea. The eventual state relation $R^*(s, w, s')$ is true if $s'$ is a state that the NFA can reach, starting in state $s$ and reading string $w$.

Formal Definition. The eventual state relation has type

$$R^* \subseteq S \times \Sigma^* \times S$$

or

$$R^*: S \times \Sigma^* \times S \rightarrow \text{Bool}$$

and is defined inductively as follows:

$$R^*(s, \epsilon, s)$$

$$R^*(s, x\alpha, s') = \exists s''. R(s, x, s'') \land R^*(s'', \alpha, s')$$
Eventual State Relation: Example

The “double digits” automaton

Eventual State Relation.

- \((S_0, \epsilon, S_0) \in R^*\) by definition
- \(S_0 \xrightarrow{0} S_0 \xrightarrow{0} S_0 \xrightarrow{1} S_0\), hence \((S_0, \text{"001"}, S_0) \in R^*\).
- \(S_0 \xrightarrow{0} S_1 \xrightarrow{0} S_2 \xrightarrow{1} S_2\), hence \((S_0, \text{"001"}, S_2) \in R^*\).
- \(S_1 \xrightarrow{0} S_2 \xrightarrow{0} S_2 \xrightarrow{1} S_2\), hence \((S_1, \text{"001"}, S_2) \in R^*\).
An Important (but Unsurprising) Theorem about $R^*$

For all states $s, s'$ and for all strings $\alpha, \beta \in \Sigma^*$

$$R^*(s, \alpha \beta, s') \text{ if and only if } \exists s''. R^*(s, \alpha, s'') \land R^*(s'', \beta, s')$$

The proof is similar to the corresponding result for $N^*$ in DFAs.
Language of a NFA

Let \( A = (\Sigma, S, s_0, F, R) \) be a NFA.

**Theorem.** A string \( w \) is *accepted* by \( A \) if

\[
\exists s \in F. \ R^*(s_0, w, s)
\]

(Compare with the definition of acceptance for NFAs earlier)

**Language of an NFA.**

The *language* accepted by \( A \) is the set of all strings accepted by \( A \)

\[
L(A) = \{ w \in \Sigma^* \mid \exists s \in F. \ R^*(s_0, w, s) \}
\]

**Informally.** That is, \( w \in L(A) \) iff *there exists* a path through the diagram for \( A \), from \( s_0 \) to a final state \( s \) (\( s \in F \)), such that the symbols on the path match the symbols in \( w \)
Power of Nondeterminism?

Q. Is there a language that is accepted by an NFA for which we cannot find a DFA that (also) accepts it?
   - it seems easier to construct NFAs
   - but in examples, DFAs did also exist

A. A simple “no”.

Theorem. If language $L$ is accepted by a NFA, then there is some DFA which accepts the same language.

Moreover, this DFA can be computed using an algorithm.
   - just like the minimal automaton can be computed using state equivalence

Drawback. The resulting DFA may have exponentially many states
   - Have to record a set of states that the NFA could be in.
Constructing the Equivalent DFA from an NFA

**Assumption.** We have an NFA with state set \( \{ q_0, \ldots, q_n \} \).

**Basic Idea.**
- consider all possible runs of the NFA in parallel
- as a consequence, can be in a *set* of states

**Construction.**
- A *state* of the DFA is a *set of states* of the NFA
- e.g. \( \{ q_3, q_7 \} \) or \( \emptyset \)
- signifies the states that the NFA can be in after reading some input
- transition function: records possible next states
- e.g. from \( \{ q_3, q_7 \} \) with letter \( x \), take union of transitions (with \( x \)) from \( q_3 \) and \( q_7 \)
- *final states* are state sets that contain a final state.
Subset Construction: The Finer Points

**Given.** NFA $A = (\Sigma, S, s_0, F, R)$.

**Subset Construction.**

- states are *subsets* of $S$ but each subset plays the role of a single state!
- transitions: for a state $Q \subseteq S$ and a letter $a \in \Sigma$:

$$N(Q, a) = \{ s_1 \in S \mid s \xrightarrow{a} s_1 \text{ for some } s \in Q \}$$

$$= \{ s_1 \in S \mid (s, a, s_1) \in R \text{ for some } s \in Q \}$$
Determinisation: Example

The “double digits” automaton

Subset Construction: transition table

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{S_0}</td>
<td>{S_0, S_1}</td>
</tr>
<tr>
<td></td>
<td>{S_0, S_1}</td>
<td>{S_0, S_3}</td>
</tr>
<tr>
<td></td>
<td>{S_0, S_1}</td>
<td>{S_0, S_2, S_3}</td>
</tr>
<tr>
<td></td>
<td>{S_0, S_2}</td>
<td>{S_0, S_2}</td>
</tr>
<tr>
<td></td>
<td>{S_0, S_2}</td>
<td>{S_0, S_2, S_3}</td>
</tr>
<tr>
<td></td>
<td>{S_0, S_3}</td>
<td>{S_0, S_3}</td>
</tr>
</tbody>
</table>

Note.

- don’t have transition for all states, just those that are reachable from \( \{S_0\} \)
- all others are not relevant (cf. elimination of unreachable states)
- having all states would require \(2^4 = 16\) entries.
Determinisation Example, as Diagrams

Double Digits, as NFA.

Double Digits as DFA.
**Q.** Can there be a simpler DFA (with fewer states) that recognises the same language?

- **initial split:** \( \{S_0, S_{01}, S_{03}\} \), \( \{S_{012}, S_{02}, S_{023}\} \)
- **next split:** \( \{S_0\} \), \( \{S_{01}\} \), \( \{S_{03}\} \), \( \{S_{012}, S_{02}, S_{023}\} \)
- no more splits, so \( S_{012}, S_{02} \) and \( S_{023} \) can be merged.
More Expressive Power: $\epsilon$-transitions

**Extra Ingredient:** Spontaneous transitions that don’t “eat” a letter
- NFAs that may change state *without* consuming a symbol.
- NFAs of this kind are called *NFAs with $\epsilon$-transitions*
- can convert NFAs with $\epsilon$-transitions to (standard) NFAs

**Formal Definition.** An NFA with $\epsilon$-transitions is an NFA, but the transition relation has the form

$$ R \subseteq S \times \Sigma \cup \{\epsilon\} \times S $$

- cf. NFAs with transition relation $R \subseteq S \times \Sigma \times S$
- $R(s, \epsilon, s')$ is a spontaneous transition (without reading input symbol)
- $\epsilon$ is *not* an element of the alphabet!
\(\epsilon\)-NFA: Example

**General Pattern.** \(\epsilon\)-transitions say “or”

![Diagram of \(\epsilon\)-NFA]

**Interpretation.**

- “top” automaton (with start state \(s_1\)) requires even number of 0’s
- “bottom” automaton (with start state \(s_3\)) requires even number of 1’s
- entire automaton (with start state \(s_0\)) accepts *either* an even number of 1’s *or* an even number of 0’s
Example and Acceptance

Language of this Automaton?

![Diagram of an automaton with states S0, S1, and S2, labeled with transitions for a, b, and c, and an accepting state S2.]

**Acceptance.** An \( \epsilon \)-NFA \( A \) *accepts* a word \( w = a_1 \ldots a_n \) if there is a sequence of states

\[
s_0 \xrightarrow{\epsilon^*} r_1 \xrightarrow{a_1} r_1' \xrightarrow{\epsilon^*} r_2 \xrightarrow{a_2} r_2' \ldots r_n \xrightarrow{a_n} r_n' \xrightarrow{\epsilon^*} f
\]

where \( s_0 \) is the starting state, \( f \in F \) is an accepting state and

- \( s \xrightarrow{a} t \) if there is an \( a \)-transition from \( s \) to \( t \), i.e \((s, a, t) \in R\)
- \( s \xrightarrow{\epsilon^*} t \) if there is a sequence of \( \epsilon \)-transitions (only!) from \( s \) to \( t \).

In particular: the empty string \( \epsilon \in L(A) \) if \( s_0 \xrightarrow{\epsilon^*} f \) for a final state \( f \in F \).
Eventual State Relation for $\epsilon$-NFAs

Given. An $\epsilon$-NFA $(\Sigma, S, s_0, F, R)$ (i.e. $R \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$) then the $\epsilon$-closure of a state $s \in S$ is given by

$$\text{eclose}(s) = \{s' \in S \mid \text{there is a sequence of } \epsilon\text{-transitions from } s \text{ to } s'\}$$

and the eventual state relation is given by

$$R^*(s, \epsilon, s') \iff s' \in \text{eclose}(s)$$

$$R^*(s, aw, s') \iff \text{there are } s_0 \text{ and } s_1 \text{ such that }$$

$$s_0 \in \text{eclose}(s), (s_0, a, s_1) \in R, (s_1, w, s') \in R^*$$

As for DFAs / NFAs:

A string $w$ is accepted by an $\epsilon$-NFA $A$ (in symbols: $w \in L(A)$) if $(s_0, w, f) \in R^*$ for some final state $f \in F$, that is

$$L(A) = \{w \in \Sigma^* \mid \exists f \in F. (s_0, w, f) \in R^*\}$$

Q. How does this relate to the notion of acceptance earlier?
Relationship Between NFAs and $\epsilon$-NFAs

Q. Are there languages only accepted by $\epsilon$-NFAs?

A. No. Every $\epsilon$-NFA $A = (\Sigma, S, s_0, F, R)$ can be converted to an NFA $A'$ without $\epsilon$-transitions so that $L(A) = L(A')$.

Construction. Put $A' = (\Sigma, S, s_0, F', R')$ where

- Make $s \in S$ an accepting state in $A'$ if $s$ can reach an accepting state in $A$ by $\epsilon$-transitions:

$$F' = \{ s \in S \mid \text{eclose}(s) \cap F \neq \emptyset \}$$

- Put an arc $s \xrightarrow{a} t$ into $A'$ if there is a transition $s' \xrightarrow{a} t$ in $A$ with $s' \in \text{eclose}(s)$:

$$R' = \{ (s, a, t) \mid (s', a, t) \in R \text{ for some } s' \in \text{eclose}(s) \}$$

(and convince yourself that $A$ and $A'$ accept the same strings!)
Regular Expressions

Challenge. Understand the computational power of DFAs / NFAs.

Approach. Characterise the languages that can be accepted by an NFA in a different form.

One Characterisation. Regular expressions (cf. Perl, Ruby, grep)

Basic Operators used to construct new expressions from old:
- vertical bar (pipe): choose either the left or right expression
- Kleene star: repeat strings from an expression
- $\epsilon$, the empty string, and every letter of the alphabet
- concatenation, for sequencing expressions
- parentheses, for grouping

Example.
- $a^*$ indicates 0 or more $a$s.
- $yes \mid no$ is the language with just the 2 given strings.
- $(0 \mid 1)^*$ indicates the set of binary numerals.
Regular Expressions — More Examples

- $0|\,(1(0|1)^*)$ is the set of binary numerals with no leading zeros.
- $(a \mid b)^*c(a \mid b)^*$ is the set of strings over $\{a, b, c\}$ with just one $c$.
- $(0*10*10^*)^*$ is the language of bit-strings that have an even number of ones. (Alternatively $0^*(10*10^*)^*$)
- $(z^*(x^* \mid y^*) z))^*$ is the set of strings over $\{x, y, z\}$ with no $x$ and $y$ adjacent.
- $1 \mid (0\, (\epsilon \mid(.(0 \mid 1)^*1))))$ is binary fractional numerals between 0 and 1 with no trailing zeroes. (e.g. 0.1, 0.110011 but not .1 or 0.10)
The Definition of Regular Expressions

Key Concept.
- regular expressions are purely *syntactical* – just like formulae
- *but*: every expression denotes a set of strings – this is the meaning.

Definition. The regular expressions over alphabet Σ and the sets that they denote are:
- ∅ is a regular expression and denotes the empty set ∅
- ε is a regular expression and denotes the set {ε}
- for each \( a \in \Sigma \), \( a \) is a regular expression and denotes the set \( \{a\} \)

If \( \alpha \) and \( \beta \) are regular expressions denoting languages \( R \) and \( S \) respectively, then:
- \( \alpha | \beta \) denotes \( R \cup S \)
- \( \alpha \beta \) denotes \( RS \) which is \( \{xy \mid x \in R \land y \in S\} \)
- \( \alpha^* \) denotes \( R^* \), ie, the set of *finitely* many \( r_i \in R \), concatenated \( R^* \) is (inductively) defined as \( \{\epsilon\} \cup RR^* \)
Regular Expressions and DFAs

Key Insight.

Regular expressions and NFAs / DFAs are equivalent.

- for every DFA $A$, have regular expression $r$ with $L(A) = L(r)$
- for every regular expression $r$, have DFA $A$ with $L(r) = L(A)$
- so the “power” of NFAs / DFAs are completely described by regular expressions.

Q. Can we “compute” more than what can be described by regular expressions?
Regular Expressions to $\epsilon$-NFAs

**Key Insight.**
- regular expressions are an *inductively defined structure*
- e.g. representable by an inductive data type in Haskell
- as a consequence, we can give *inductive definition* of the corresponding automaton

**Construction.** (start state on left, final state on right)
- When the regular expression is a symbol $a$ of the alphabet (language is $\{a\}$) the automaton is
  
  $\overset{a}{\longrightarrow}$

- When the regular expression is $\epsilon$ (language is $\{\epsilon\}$) the automaton is
  
  $\overset{\epsilon}{\longrightarrow}$

- When the regular expression is $\emptyset$ (language is $\emptyset$) the automaton has no edges
  
  $\overset{\emptyset}{\longrightarrow}$
Regular Expressions to NFAs, ctd

Suppose the NFA corresponding to some $R$ is:

\[
\begin{array}{c}
\text{R} \\
\end{array}
\]

Then NFAs corresponding to composite regular expressions are defined as follows:

- $R_1 R_2$

\[
\begin{array}{c}
\text{R}_1 \\
\text{R}_2 \\
\end{array}
\]

- $R^*$

\[
\begin{array}{c}
\text{R} \\
\end{array}
\]

- $R_1 \mid R_2$

\[
\begin{array}{c}
\text{R}_1 \\
\text{R}_2 \\
\end{array}
\]
Example

Given the regular expression for binary numerals without leading zeros, \((0 \mid 1(0|1)^*)\), the above algorithm gives this NFA.
Closing the Loop

**Given.** A finite alphabet $\Sigma$ and a language $L \subseteq \Sigma^*$. The following are equivalent:

- $L$ can be described by a regular expression
- $L$ can be recognised by an $\epsilon$-NFA
- $L$ can be recognised by an NFA
- $L$ can be recognised by a DFA . . .

as we can convert regular expressions into $\epsilon$-NFAs into NFAs into DFAs.

**Missing Link.** Construction of regular expressions from DFAs (not covered in this course)
Summary.

Starting Point. Finite Automata
- motivated by computers having finite memory (only)
- solving simple problems: is string $s$ accepted?

Limitations of Finite Automata
- e.g. cannot recognise $L = \{a^n b^n \mid n \geq 0\}$

Characterisation of expressive power
- can go back and forth between automata and regular expressions

Q. Are finite automata a “good” model of computation?
- if yes, why?
- if not, why not? What is missing?
Literature.

- Introduction to Automata Theory, Languages, and Computation By Hopcroft, Motwani, and Ullman.
  A classic text that has been re-worked from a standard textbook.
- Introduction To The Theory Of Computation by Michael Sipser
  The part on Automata and Languages covers (more than) what we have discussed here.