# Regular expressions and Kleene's theorem

## Informatics 2A: Lecture 5

Mary Cryan

School of Informatics
University of Edinburgh
mcryan@inf.ed.ac.uk

26 September 2018

Finishing DFA minimization

- An algorithm for minimization

More closure properties of regular languages

- Operations on languages
- $\epsilon$-NFAs
- Closure under concatenation and Kleene star

Regular expressions

- Regular expressions

# An algorithm for minimization

First eliminate any unreachable states (easy).

Then create a table of all possible pairs of states $(p, q)$, initially unmarked. (E.g. a two-dimensional array of booleans, initially set to false.) We mark pairs $(p, q)$ as and when we discover that $p$ and $q$ cannot be equivalent.

1. Start by marking all pairs $(p, q)$ where $p \in F$ and $q \notin F$, or vice versa.
2. Look for unmarked pairs $(p, q)$ such that for some $u \in \Sigma$, the pair $(\delta(p, u), \delta(q, u))$ is marked. Then mark $(p, q)$.
3. Repeat step 2 until no such unmarked pairs remain.

If $(p, q)$ is still unmarked, can collapse $p, q$ to a single state.

# Why does this algorithm work?

Let's say a string $s$ separates states $p, q$ if $s$ takes us from $p$ to an accepting state and from $q$ to a rejecting state, or *vice versa*.
Such an $s$ is a reason for not merging $p, q$ into a single state.
We mark $(p, q)$ when we find that there's a string separating $p, q$:

- If $p \in F$ and $q \notin F$, or *vice versa*, then $\epsilon$ separates $p, q$.
- Suppose we mark $(p, q)$ because we've found a previously marked pair $(p', q')$ where $p \xrightarrow{a} p'$ and $q \xrightarrow{a} q'$ for some $a$.
  If $s'$ is a separating string for $p', q'$, then $as'$ separates $p, q$.

We stop when there are no more pairs we can mark.
If $(p, q)$ remains unmarked, why are $p, q$ equivalent?

- If $s = a_1 \ldots a_n$ were a string separating $p, q$, we'd have

$$\begin{aligned} p &= p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} \cdots p_{n-1} \xrightarrow{a_n} p_n \,, \\ q &= q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots q_{n-1} \xrightarrow{a_n} q_n \end{aligned}$$

  with just one of $p_n, q_n$ accepting. So we'd have marked $(p_n, q_n)$ in Round 0, $(p_{n-1}, q_{n-1})$ by Round 1, ... and $(p, q)$ by Round $n$.

# Alternative: Brzozowski's minimization algorithm

There's a surprising alternative algorithm for minimizing a DFA $M = (Q, \delta, s, F)$ for a language $L$. Assume no unreachable states.

- Reverse the machine $M$: flip all the arrows, make $F$ the set of start states, and make $s$ the only accepting state.
  This gives an NFA $N$ (*not* typically a DFA) which accepts $L^{rev} = \{rev(s) \mid s \in L\}$.
- Apply the subset construction to $N$, omitting unreachable states, to get a DFA $P$.
  It turns out that $P$ is minimal for $L^{rev}$ (clever)!
- Now apply the same two steps again, starting from $P$.
  The result is a minimal DFA for $(L^{rev})^{rev} = L$.

# Comparing Brzozowski and marking algorithms

- Both algorithms result in the same minimal DFA for a given DFA $M$ (recall that there's a unique minimal DFA up to isomorphism.)
- In the worst case, Brzozowski's algorithm can take time $O(2^n)$ for a DFA with $n$ states. The marking algorithm, as presented, runs within time $O(kn^4)$, where $k = |\Sigma|$. (Can be improved further.)
- There are some practical cases where Brzozowski does better.
- Marking algorithm is probably easier to understand, and illustrates a common pattern (more examples later in course).

# Improving determinization

Now we have a minimization algorithm, the following improved determinization procedure is possible.

To determinize an NFA $M$ with $n$ states:

1. Perform the subset construction on $M$ to produce an equivalent DFA $N$ with $2^n$ states.
2. Perform the minimization algorithm on $N$ to produce a DFA $\mathrm{Min}(N)$ with $\leq 2^n$ states.

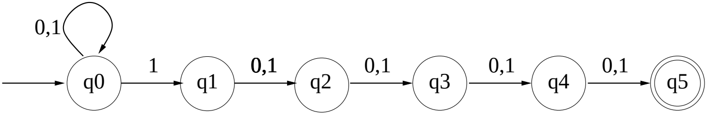
Using this method we are guaranteed to produce the smallest possible DFA equivalent to $M$.

In many cases this avoids the exponential state-space blow-up.

In some cases, however, an exponential blow-up is unavoidable.

# Question from lecture 4

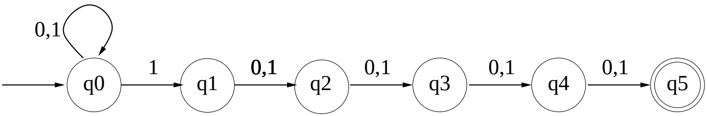
Consider our example NFA over $\{0, 1\}$:




What is the number of states of the smallest DFA that recognises the same language?

# Question from lecture 4

Consider our example NFA over $\{0, 1\}$:

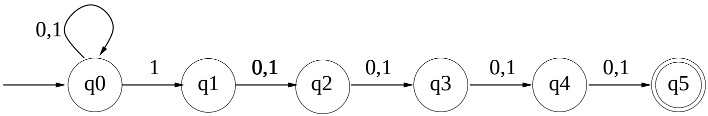



What is the number of states of the smallest DFA that recognises the same language?

Answer: The smallest DFA has 32 states.

# Question from lecture 4

Consider our example NFA over $\{0, 1\}$:




What is the number of states of the smallest DFA that recognises the same language?

Answer: The smallest DFA has 32 states.

More generally, the smallest DFA for the language:

$$\{x \in \Sigma^* \mid \text{the } n\text{-th symbol from the end of } x \text{ is } 1\}$$

has $2^n$ states. Whereas, there is an NFA with $n + 1$ states.

# Concatenation

We write $L_1.L_2$ for the concatenation of languages $L_1$ and $L_2$, defined by:

$$L_1.L_2 \;=\; \{xy \mid x \in L_1, y \in L_2\}$$

For example, if $L_1 = \{aaa\}$ and $L_2 = \{b, c\}$ then $L_1.L_2$ is the language $\{aaab,\ aaac\}$.

Later we will prove the following closure property.

*If $L_1$ and $L_2$ are regular languages then so is $L_1.L_2$.*

# Kleene star

We write $L^*$ for the Kleene star of the language $L$, defined by:

$$L^* \ = \ \{\epsilon\} \cup L \cup L.L \cup L.L.L \cup \dots$$

For example, if $L_3 = \{aaa, b\}$ then $L_3^*$ contains strings like *aaaaaa*, *bbbbb*, *baaaaaabbaaa*, etc.

More precisely, $L_3^*$ contains all strings over $\{a, b\}$ in which the letter $a$ always appears in sequences of length some multiple of 3

Later we will prove the following closure property.

*If $L$ is a regular language then so is $L^*$.*

# Exercise

Consider the language over the alphabet $\{a, b, c\}$

$$L = \{x \mid x \text{ starts with } a \text{ and ends with } c\}$$

Which of the following strings are valid for the language $L.L$ ?

1. *abcabc*
2. *acacac*
3. *abcbcac*
4. *abcbacbc*

# Exercise

Consider the language over the alphabet $\{a, b, c\}$

$$L = \{x \mid x \text{ starts with } a \text{ and ends with } c\}$$

Which of the following strings are valid for the language $L.L$ ?

1. *abcabc*
2. *acacac*
3. *abcbcac*
4. *abcbacbc*

Answer: 1,2,3 are valid, but 4 isn't. (To split the string into two $L$-strings, we'd need $c$ followed by $a$.)

# Another exercise

Consider the (same) language over the alphabet $\{a, b, c\}$

$$L = \{x \mid x \text{ starts with } a \text{ and ends with } c\}$$

Which of the following strings are valid for the language $L^*$ ?

1. $\epsilon$
2. *acaca*
3. *abcbc*
4. *acacacacac*

# Another exercise

Consider the (same) language over the alphabet $\{a, b, c\}$

$$L = \{x \mid x \text{ starts with } a \text{ and ends with } c\}$$

Which of the following strings are valid for the language $L^*$ ?

1. $\epsilon$
2. *acaca*
3. *abcbc*
4. *acacacacac*

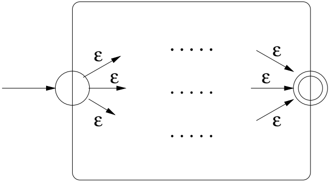
Answer: 1,3,4 are valid, but not 2. (In this particular case, it so happens that $L^* = L + \{\epsilon\}$, but this won't be true in general.)

# NFAs with $\epsilon$-transitions

We can vary the definition of NFA by also allowing transitions labelled with the special symbol $\epsilon$ (*not* a symbol in $\Sigma$).

The automaton may (but doesn't have to) perform a spontaneous $\epsilon$-transition at any time, without reading an input symbol.

This is quite convenient: for instance, we can turn any NFA into an $\epsilon$-NFA with just one start state and one accepting state:




(Add $\epsilon$-transitions from new start state to each state in $S$, and from each state in $F$ to new accepting state.)

# Equivalence to ordinary NFAs

Allowing $\epsilon$-transitions is just a convenience: it doesn't fundamentally change the power of NFAs.

If $N = (Q, \Delta, S, F)$ is an $\epsilon$-NFA, we can convert $N$ to an ordinary NFA with the same associated language, by simply 'expanding' $\Delta$ and $S$ to allow for silent $\epsilon$-transitions.

To achieve this, perform the following steps on $N$.

- For every pair of transitions $q \xrightarrow{a} q'$ (where $a \in \Sigma$) and $q' \xrightarrow{\epsilon} q''$, add a new transition $q \xrightarrow{a} q''$.
- For every transition $q \xrightarrow{\epsilon} q'$, where $q$ is a start state, make $q'$ a start state too.

Repeat the two steps above until no further new transitions or new start states can be added.

Finally, remove all $\epsilon$-transitions from the $\epsilon$-NFA resulting from the above process. This produces the desired NFA.

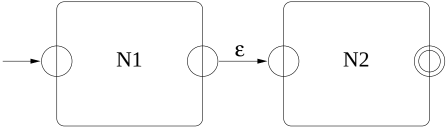
# Closure under concatenation

We use $\epsilon$-NFAs to show, as promised, that regular languages are closed under the concatenation operation:

$$L_1.L_2 \;=\; \{xy \mid x \in L_1, y \in L_2\}$$

If $L_1, L_2$ are any regular languages, choose $\epsilon$-NFAs $N_1, N_2$ that define them. As noted earlier, we can pick $N_1$ and $N_2$ to have just one start state and one accepting state.

Now hook up $N_1$ and $N_2$ like this:



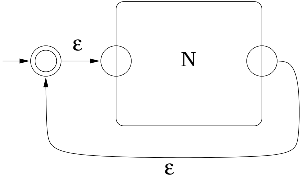

Clearly, this NFA corresponds to the language $L_1.L_2$.

# Closure under Kleene star

Similarly, we can now show that regular languages are closed under the Kleene star operation:

$$L^* \;=\; \{\epsilon\} \cup L \cup L.L \cup L.L.L \cup \ldots$$

For suppose $L$ is represented by an $\epsilon$-NFA $N$ with one start state and one accepting state. Consider the following $\epsilon$-NFA:




Clearly, this $\epsilon$-NFA corresponds to the language $L^*$.

# Regular expressions

We've been looking at ways of specifying regular languages via machines (often presented as pictures). But it's very useful for applications to have more textual ways of defining languages.

A regular expression is a written mathematical expression that defines a language over a given alphabet $\Sigma$.

- The basic regular expressions are

$$\emptyset \qquad \epsilon \qquad a \ \ (\text{for } a \in \Sigma)$$

- From these, more complicated regular expressions can be built up by (repeatedly) applying the two binary operations $+$, $.$ and the unary operation $^*$ . Example: $(a.b + \epsilon)^* + a$

We use brackets to indicate precedence. In the absence of brackets, $^*$ binds more tightly than ., which itself binds more tightly than $+$.

$$\text{So} \quad a + b.a^* \quad \text{means} \quad a + (b.(a^*))$$

Also the dot is often omitted: $ab$ means $a.b$

# Reading

Relevant reading:

- DFA minimization: Kozen Chapters 13 & 14.
- Regular expressions: Kozen chapters 7,8; J & M chapter 2.1. (Both texts actually discuss more general 'patterns' — see next lecture.)
- From regular expressions to NFAs: Kozen chapter 8; J & M chapter 2.3.

Next two lectures: Some applications of all this theory.

- String and pattern matching
- Lexical analysis
- Model checking