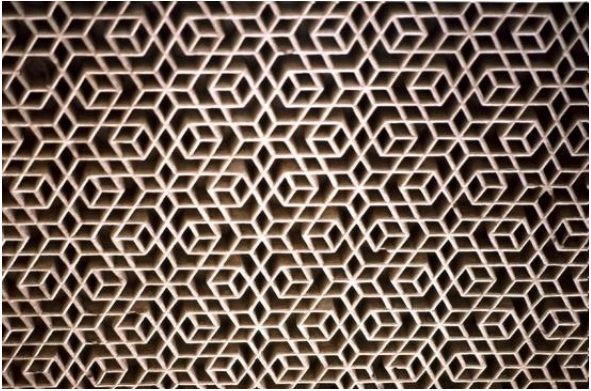
# Automata, Languages and Computation

## Chapter 4 : Properties of Regular Languages

Master Degree in Computer Engineering
University of Padua
Lecturer : Giorgio Satta

Lecture based on material originally developed by :
Gösta Grahne, Concordia University

# Properties of regular languages



1. Pumping Lemma : every regular language satisfies this property; useful to show that some languages are not regular
2. Closure properties : how to combine automata using specific operations
3. Decision problems : algorithms for the solution of problems based on automata/regex and their complexity
4. Automata minimization : reduce number of states to a minimum

# Introduction to pumping lemma

Suppose $L_{01} = \{0^n 1^n \mid n \geqslant 1\}$ were a regular language

Then $L_{01}$ must be recognized by some DFA $A$; let $k$ be the number of states of $A$

Assume $A$ reads $0^k$. Then $A$ must go through the following transitions :

fence notation
$0^{k+1}$
(because of $\mathcal{E}$)

| | |
|---|---|
| $\epsilon$ | $p_0$ |
| $0$ | $p_1$ |
| $00$ | $p_2$ |
| ... | ... |
| $0^k$ | $p_k$ |

By the **pigeonhole principle**, there must exist a pair $i, j$ with $i < j \leqslant k$ such that $p_i = p_j$. Let us call $q$ this state

# Introduction to pumping lemma

q is the state we see twice
(after reading $0^i$ we are in q and for $0^j$ $(i \neq j)$ we are again in q for pigeonhole)

Now you can **fool** $A$ :

- if $\hat{\delta}(q, 1^i) \notin F$, then the machine will foolishly reject $0^i 1^i$
- if $\hat{\delta}(q, 1^i) \in F$, then the machine will foolishly accept $0^j 1^i$

⚠ these are different and we can only do one of the two correctly

In other words: state $q$ would represent inconsistent information about the count of occurrences of 0 in the string read so far

Therefore $A$ does not exists, and $L_{01}$ is not a regular language

# Pumping lemma for regular languages

n is different for each language

**Theorem** Let $L$ be any regular language. Then $\exists n \in \mathbb{N}$ depending on $L$, $\forall w \in L$ with $|w| \geqslant n$, we can factorize $w = xyz$ with :

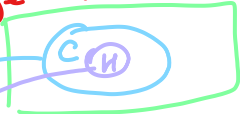
in the example below $0^i$ → 1 ← 0's

- $y \neq \epsilon$
- $|xy| \leqslant n$ ⟷ the cuts are in the "beginning of the string"
- $\forall k \geqslant 0,\ xy^k z \in L$

all languages

there are no such strings if the language is finite

not in the book  if H then C  H ⇒ C ≡ H ⊆ C




p. lemma

being regular

# Pumping lemma for regular languages

We will do exercises like: Show that L is not regular proof: show that L does not have the p. lemma property Then L cannot be regular □

**Proof**

Suppose $L$ is a regular language

Then $L$ is recognized by some DFA $A$ with, say, $n$ states

Let $w = a_1 a_2 \cdots a_m \in L$ with $m \geqslant n$

Let $p_i = \hat{\delta}(q_0, a_1 a_2 \cdots a_i)$, for each $i = 0, 1, \ldots, n$

There exists $i < j \leqslant n$ such that $p_i = p_j$

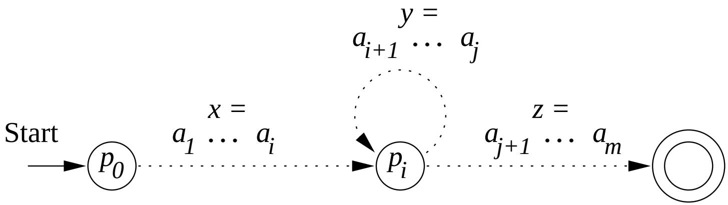
Be careful

P does NOT imply C, showing that p. lemma hold does not mean that it's a regular language

# Pumping lemma for regular languages

Let us write $w = xyz$, where

- $x = a_1 a_2 \cdots a_i$
- $y = a_{i+1} a_{i+2} \cdots a_j$
- $z = a_{j+1} a_{j+2} \ldots a_m$




Evidently, $xy^k z \in L$, for any $k \geqslant 0$ $\square$

# Example

Let $\Sigma$ be some alphabet, and let $w \in \Sigma^*$, $a \in \Sigma$. We write $\#_a(w)$ to denote the **number of occurrences** of $a$ in $w$

We define

$$L_{eq} = \{w \mid w \in \{0,1\}^*,\ \#_0(w) = \#_1(w)\}$$

In words, $L_{eq}$ is the language whose strings have an equal number of 0's and 1's

Use the pumping lemma to show that $L$ is not regular

# Example

**Proof** Suppose $L_{eq}$ were regular. Then $L(A) = L_{eq}$ for some DFA $A$

Let $n$ be the number of states of $A$ and let $w = 0^n 1^n \in L(A)$

By the pumping lemma we can factorize $w = xyz$ with

- $|xy| \leqslant n$,
- $y \neq \epsilon$

and state that, for each $k \geqslant 0$, we have $xy^k z \in L(A)$

we cannot cut where there is a 1 because we would have $|xy| > n$

$$w = \underbrace{000\cdots}_{x}\underbrace{\cdots 00}_{y}\underbrace{\cdots 0111\cdots 11}_{z}$$

⚠ It is not enough to show there exists a cut where the p.lemma is not true, we need to show the p.lemma is false for every cut (because the lemma says ∃)

# Example

For $k = 0$ we have $xz \in L(A)$

This is a **contradiction**, since $|y| \geqslant 1$ and then $xz$ has fewer 0's than 1's

We therefore conclude that $L(A) \neq L_{eq}$ $\square$

Comment of the if-then formulation of the pumping lemma: many students wrongly state that if the pumping lemma holds, then the language must be regular

if we choose $w = 101010\ldots$ it's impossible to disprove the p.lemma, it's important to choose the right $w$!

# Example

**Proof** (alternative) We can see the application of the pumping lemma as a game between two players

Player P2 states that $L_{eq}$ is regular, and player P1 wants to establish a **contradiction**

- P2 picks $n$ (number of states of DFA, if it exists)
- P1 picks string $w = 0^n1^n \in L_{eq}$, with $|w| \geqslant n$
- P2 picks a factorization $w = xyz$, with $|xy| \leqslant n$, $y \neq \epsilon$ and $xy^kz \in L_{eq}$ (assuming $L_{eq}$ is regular)
- P1 picks $k$ such that $xy^kz \notin L$, which is a violation of the pumping lemma. Specifically, P1 picks $k = 0$: $xz \notin L_{eq}$, since $y$ contains just 0's, $y \neq \epsilon$, and thus $\#_0(xz) < \#_1(xz) = n$
- P1 concludes that $L_{eq}$ cannot be regular □

# Example

Let $L_{pr} = \{1^p \mid p \text{ prime}\}$. Using the pumping lemma, show that $L_{pr}$ is not regular

**Proof** Let $n$ be as in the pumping lemma, and let $p \geqslant n + 2$ be some prime number. Thus $1^p \in L_{pr}$

By the pumping lemma we can write $w = xyz$ with

- $|xy| \leqslant n$,
- $y \neq \epsilon$

such that, for each $k \geqslant 0$, we have $xy^k z \in L(A)$

# Example

Let $|y| = m \geqslant 1$

$$w = \overbrace{\underbrace{111\cdots}_{x}\ \underbrace{\cdots 1}_{\substack{y \\ |y|=m\geqslant 1}}\ \underbrace{1111\cdots 11}_{z}}^{p}$$

Choose $k = p - m$, so that $xy^{p-m}z \in L_{pr}$ and then $|xy^{p-m}z|$ is a prime number

# Example

We can write $|xy^{p-m}z| = |xz| + (p-m)|y| = p - m + (p-m)m = (1+m)(p-m)$

Let us verify that none of the two factors is a 1 :

- $y \neq \epsilon$, thus $1 + m > 1$
- $m = |y| \leqslant |xy| \leqslant n$, $p \geqslant n + 2$, thus $p - m \geqslant n + 2 - m \geqslant n + 2 - n = 2$

We have derived a **contradiction** $\square$

# Exercise

For a string $w$, we write $w^R$ to denote the **reverse** of $w$. Example: $01011^R = 11010$ and $(w^R)^R = w$

Consider the language

$$L = \{ww^R \mid w \in \{0,1\}^*\}$$

Using the pumping lemma, show that $L$ is not regular

# Closure properties of regular languages

Let $L$ and $M$ be regular languages over $\Sigma$. Then the following languages are all regular

- Union: $L \cup M$
- Intersection: $L \cap M$
- Complement: $\overline{L} = \Sigma^* \setminus L$
- Difference: $L \setminus M$
- Reversal: $L^R = \{w^R \mid w \in L\}$
- Kleene closure: $L^*$
- Concatenation: $L.M$
- Homomorphism: $h(L) = \{h(w) \mid w \in L\}$
- Inverse homomorphism: $h^{-1}(L) = \{w \in \Sigma^* \mid h(w) \in L\}$

# Closure under union

**Theorem** For any regular languages $L$ e $M$, $L \cup M$ is regular

**Proof** Let $E$ and $F$ be regular expressions such that $L = L(E)$ and $M = L(F)$. Then $L \cup M$ is generated by $E + F$, and is regular by definition $\square$

# Closure under concatenation and Kleene

The proof of closure under union is rather **immediate**, since regular expressions use the union operator

Similarly, we can immediately prove the closure under

- concatenation
- Kleene operator

# Closure under complement

**Theorem** If $L$ is a regular language over $\Sigma$, then so is $\overline{L} = \Sigma^* \setminus L$

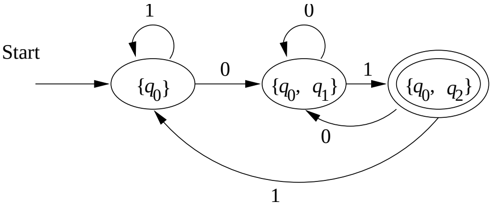
**Proof** Let $L$ be recognized by a DFA

$$A = (Q, \Sigma, \delta, q_0, F).$$

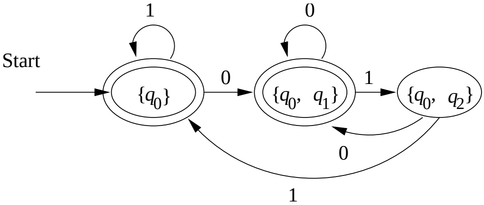
Let $B = (Q, \Sigma, \delta, q_0, Q \setminus F)$. Now $L(B) = \overline{L}$ $\square$

# Example

Let $L$ be recognized by the DFA




Then $\overline{L}$ is recognized by the DFA



# Closure under intersection

**Theorem** If $L$ and $M$ are regular, then so is $L \cap M$

**Proof** By De Morgan's law, $L \cap M = \overline{\overline{L} \cup \overline{M}}$

We already know that regular languages are closed under complement and union $\square$

# Intersection automaton

**Proof** (alternative) Let $L = L(A_L)$ and $M = L(A_M)$ for automata $A_L$ and $A_M$ with

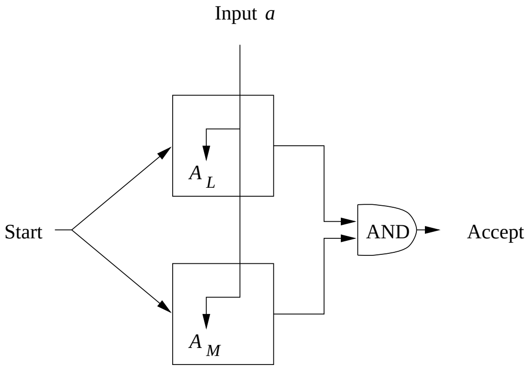
$$A_L = (Q_L, \Sigma, \delta_L, q_L, F_L)$$
$$A_M = (Q_M, \Sigma, \delta_M, q_M, F_M)$$

Without any loss of generality, we assume that both automata are deterministic

We shall construct an automaton that simulates $A_L$ and $A_M$ in parallel, and accepts if and only if both $A_L$ and $A_M$ accept

# Intersection automaton

**Idea** : If $A_L$ goes from state $p$ to state $s$ upon reading $a$, and $A_M$ goes from state $q$ to state $t$ upon reading $a$, then $A_{L \cap M}$ will go from state $(p, q)$ to state $(s, t)$ upon reading a



# Intersection automaton

Formally

$$A_{L\cap M} = (Q_L \times Q_M, \Sigma, \delta_{L\cap M}, (q_{L,0}, q_{M,0}), F_L \times F_M),$$

where

$$\delta_{L\cap M}((p,q),a) = (\delta_L(p,a), \delta_M(q,a))$$

the book skips the next part

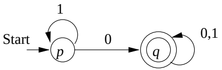
We can show by induction on $|w|$ that

$$\hat{\delta}_{L\cap M}((q_{L,0}, q_{M,0}), w) = \left(\hat{\delta}_L(q_{L,0}, w), \hat{\delta}_M(q_{M,0}, w)\right)$$

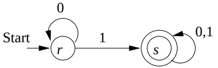
Then $A_{L\cap M}$ accepts if and only if $A_L$ and $A_M$ accept $\square$

# Exercise

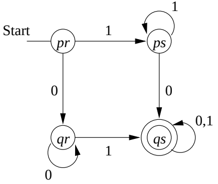
Build an automaton that accepts strings with at least one 0 and at least one 1. Let's build **simpler** automata and take the intersection




(a)




(b)




(c)

# Closure under set difference

**Theorem** If $L$ and $M$ are regular languages, so is $L \setminus M$

**Proof** Observe that $L \setminus M = L \cap \overline{M}$

We already know that regular languages are closed under complement and intersection $\square$

# Closure under reverse operator

$a \in \Sigma$

$x \in \Sigma^*$

$$w^R = \begin{cases} \text{if } w = \varepsilon \text{ then } w^R = \varepsilon \\ \text{otherwise } w^R = a x^R \text{ s.t. } xa = w \end{cases}$$

$cia\cancel{o}$ $\quad o(cia)^R$

**Theorem** If $L$ is regular, so is $L^R$

**Proof** Let $L$ be recognized by FA $A$. Turn $A$ into an FA for $L^R$ by

- reversing all arcs
- make the old start state the new sole accepting state
- create a new start state $p_0$ such that $\delta(p_0, \epsilon) = F$, $F$ the set of accepting states of old $A$ □

# Closure under reverse operator

**Proof** (alternative) Let $E$ be a regular expression. We shall construct a regular expression $E^R$ such that $L(E^R) = (L(E))^R$

We proceed by structural induction on E

**Base** If $E$ is $\epsilon$, $\emptyset$, or $\boldsymbol{a}$, then $E^R = E$ (easy to verify)

# Closure under reverse operator

**Induction**

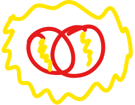
- $E = F + G$ : We need to reverse the two languages. Then $E^R = F^R + G^R$
- $E = F.G$ : We need to reverse the two languages and also reverse the order of their concatenation. Then $E^R = G^R.F^R$
- $E = F^*$ :
  $w \in L(F^*)$ means $\exists k : w = w_1 w_2 \cdots w_k,\ w_i \in L(F)$
  then $w^R = w_k^R w_{k-1}^R \cdots w_1^R,\ w_i^R \in L(F^R)$
  then $w^R \in L(F^R)^*$
  Same reasoning for the inverse direction. Then $E^R = (F^R)^*$

Thus $L(E^R) = (L(E))^R$ □

# Test

State whether the following claims hold true, and motivate your answer

1. the intersection of a non-regular language and a finite language is always a regular language
2. the intersection of a non-regular language $L_1$ and an infinite regular language $L_2$ is never a regular language false
3. every subset of a non-regular language is a non-regular language false



1. $|R| < +\infty \Longrightarrow |R \cap N| < +\infty$

2. $\underbrace{L(a^* b^*)}_{\text{regular}} \cap \underbrace{L_>}_{\text{not reg}} = \emptyset$ ↯

3. Sol: take a finite subset (also $\emptyset$)

# Superset and subset

Assume $L$ is a regular language. We **cannot say anything** about languages $L'$ and $L''$ with $L' \subset L$ and $L'' \supset L$

More precisely

- $L'$ could be regular or non-regular
- $L''$ could be regular or non-regular

Often student gets confused about this, thinking that adding strings to $L$ makes it 'more difficult' and removing strings from $L$ makes it 'less difficult'. But this is **not true in general**

# Homomorphisms

Let $\Sigma$ and $\Delta$ be two alphabets. A **homomorphisms** over $\Sigma$ is a function $h : \Sigma \to \Delta^*$

Informally, a homomorphism is a function which replaces each symbol with a string

**Example** : Let $\Sigma = \{0, 1\}$ and define $h(0) = ab$, $h(1) = \epsilon$; $h$ is a homomorphism over $\Sigma$

# Homomorphisms

We extend $h$ to $\Sigma^*$ : if $w = a_1 a_2 \cdots a_n$ then

$$h(w) = h(a_1)h(a_2)\cdots h(a_n)$$

Equivalently, we can use a **recursive** definition :

$$h(w) = \begin{cases} \epsilon, & \text{if } w = \epsilon; \\ h(x)h(a) & \text{if } w = xa,\ x \in \Sigma^*,\ a \in \Sigma. \end{cases}$$

**Example** : Using $h$ from previous example on string 01001 results in *ababab*

# Homomorphisms

For a language $L \subseteq \Sigma^*$

$$h(L) = \{h(w) \mid w \in L\}$$

**Example** : Let $L$ be the language associated with the regular expression $\mathbf{10^*1}$. Then $h(L)$ is the language associated with the regular expression $(\boldsymbol{ab})^*$

h is overloaded: it works on symbols, strings and now languages

## Closure under homomorphism NO DIM ALL'ESAME

**Theorem** Let $L \subseteq \Sigma^*$ be a regular language and let $h$ be a homomorphisms over $\Sigma$. Then $h(L)$ is a regular language

**Proof** Let $E$ be a regular expression generating $L$. We define $h(E)$ as the regular expression obtained by substituting in $E$ each symbol $\boldsymbol{a}$ with $\boldsymbol{a_1 a_2 \cdots a_k}$, under the assumption that

- $a \in \Sigma$
- $h(a) = a_1 a_2 \cdots a_k$, $k \geqslant 0$

We now prove the statement

$$L(h(E)) = h(L(E)),$$

using structural induction on $E$

# Closure under homomorphism

**Base** $E = \epsilon$ or else $E = \emptyset$. Then $h(E) = E$, and $L(h(E)) = L(E) = h(L(E))$

$E = \boldsymbol{a}$ with $a \in \Sigma$. Let $h(a) = a_1 a_2 \cdots a_k$, $k \geqslant 0$. Then $L(\boldsymbol{a}) = \{a\}$ and thus $h(L(\boldsymbol{a})) = \{a_1 a_2 \cdots a_k\}$

The regular expression $h(\boldsymbol{a})$ is $\boldsymbol{a_1 a_2} \cdots \boldsymbol{a_k}$. Then $L(h(\boldsymbol{a})) = \{a_1 a_2 \cdots a_k\} = h(L(\boldsymbol{a}))$

# Closure under homomorphism

**Induction** Let $E = F + G$. We can write

$$
\begin{array}{rcll}
L(h(E)) & = & L(h(F + G)) & \\
& = & L(h(F) + h(G)) & h \text{ defined over regex} \\
& = & L(h(F)) \cup L(h(G)) & + \text{ definition} \\
& = & h(L(F)) \cup h(L(G)) & \text{inductive hypothesis for } F, G \\
& = & h(L(F) \cup L(G)) & h \text{ defined over languages} \\
& = & h(L(F + G)) & + \text{ definition} \\
& = & h(L(E)) &
\end{array}
$$

Salta nodi nella ricorsione fino alle foglie

# Closure under homomorphism

Let $E = F.G$. We can write

$$\begin{array}{rcll} L(h(E)) & = & L(h(F.G)) & \\ & = & L(h(F).h(G)) & h \text{ defined over regex} \\ & = & L(h(F)).L(h(G)) & . \text{ definition} \\ & = & h(L(F)).h(L(G)) & \text{inductive hypothesis for } F, G \\ & = & h(L(F).L(G)) & h \text{ defined over languages} \\ & = & h(L(F.G)) & . \text{ definition} \\ & = & h(L(E)) & \end{array}$$

# Closure under homomorphism

Let $E = F^*$. We can write

$$
\begin{array}{rcll}
L(h(E)) & = & L(h(F^*)) & \\
 & = & L([h(F)]^*) & h \text{ defined over regex} \\
 & = & \bigcup_{k \geqslant 0} [L(h(F))]^k & * \text{ definition} \\
 & = & \bigcup_{k \geqslant 0} [h(L(F))]^k & \text{inductive hypothesis for } F \\
 & = & \bigcup_{k \geqslant 0} h([L(F)]^k) & h \text{ definition over languages} \\
 & = & h(\bigcup_{k \geqslant 0} [L(F)]^k) & h \text{ definition over languages} \\
 & = & h(L(F^*)) & * \text{ definition} \\
 & = & h(L(E)) &
\end{array}
$$

□

# Conversion complexity

We can convert among DFA, NFA, $\epsilon$-NFA, and regular expressions

What is the **computational complexity** of these conversions?

We investigate the computational complexity as a function of

- number of states $n$ for an FA
- number of operators $n$ for a regular expressions
- we assume $|\Sigma|$ is a constant

the proofs and the algorithms are
not required for these theorems

# From $\epsilon$-NFA to DFA (subset construction)

Suppose an $\epsilon$-NFA has $n$ states. To compute ECLOSE$(p)$ we visit at most $n^2$ arcs. We do this for $n$ states, resulting in time $\mathcal{O}(n^3)$

The resulting DFA has $2^n$ states. For each state $S$ and each $a \in \Sigma$ we compute $\delta(S, a)$ in time $\mathcal{O}(n^3)$. In total, the computation takes $\mathcal{O}(n^3 \cdot 2^n)$ steps, that is, **exponential time**

If we compute $\delta$ just for the **reachable** states

- we need to compute $\delta(S, a)$ $s$ times only, with $s$ the number of reachable states
- in total the computation takes $\mathcal{O}(n^3 \cdot s)$ steps

# Other conversions

From NFA to DFA : computation takes **exponential time**

From DFA to NFA :

- put set brackets around the states
- computation takes time $\mathcal{O}(n)$, that is, **linear time**

From FA to regular expression via state elimination construction: computation takes **exponential time**

# Other conversions

From regular expression to $\epsilon$-NFA :

- construct a tree representing the structure of the regular expression in time $\mathcal{O}(n)$
- at each node in the tree, we build new nodes and arcs in time $\mathcal{O}(1)$ and use **pointers** to previously built structure, avoiding copying
- grand total time is $\mathcal{O}(n)$, that is, **linear time**

# Decision problems

In the problem instances below, languages $L$ and $M$ are expressed in any of the four representations introduced for regular languages

- $L = \emptyset$ ?
- $w \in L$ ?
- $L = M$ ?

Are these problems solvable? How?

No proofs

# Empty language

$L(A) \neq \emptyset$ for FA $A$ if and only if at least one final state is **reachable** from the initial state of $A$

**Algorithm** for computing reachable states :

**Base** The initial state is reachable

**Induction** If $q$ is reachable and there exists a transition from $q$ to $p$, then $p$ is reachable

Computation takes time proportional to the number of arcs in $A$, thus $\mathcal{O}(n^2)$

We already saw this idea in the lazy evaluation for translating NFA into DFA

we find all reachable states, if any of them is final it's not empty

# Empty language

Given a regular expression $E$, we can decide $L(E) \stackrel{?}{=} \emptyset$ by structural induction

**Base**

- $E = \epsilon$ or else $E = \boldsymbol{a}$. Then $L(E)$ is non-empty
- $E = \emptyset$. Then $L(E)$ is empty

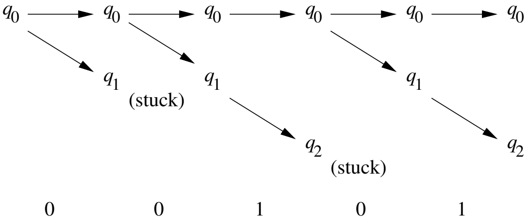
**Induction**

- $E = F + G$. Then $L(E)$ is empty if and only if both $L(F)$ and $L(G)$ are empty
- $E = F.G$. Then $L(E)$ is empty if and only if either $L(F)$ or $L(G)$ are empty
- $E = F^*$. Then $L(E)$ is not empty, since $\epsilon \in L(E)$

# Language membership

We can test $w \in L(A)$ for DFA $A$ by simulating $A$ on $w$. If $|w| = n$ this takes $\mathcal{O}(n)$ steps

If $A$ is an NFA with $s$ states, simulating $A$ on $w$ requires $\mathcal{O}(n \cdot s^2)$ steps



# Language membership

If $A$ is an $\epsilon$-NFA with $s$ states, simulating $A$ on $w$ requires $\mathcal{O}(n \cdot s^3)$ steps

Alternatively, we can pre-process $A$ by calculating ECLOSE($p$) for $s$ states, in time $\mathcal{O}(s^3)$. Afterwards, the simulation of each symbol $a$ from $w$ is carried out as follows

- from the current states, find the successor states under $a$ in time $\mathcal{O}(s^2)$
- compute the $\epsilon$-closure for the successor states in time $\mathcal{O}(s^2)$

This takes time $\mathcal{O}(n \cdot s^2)$

# Language membership

If $L = L(E)$, for some regular expression $E$ of length $s$, we first convert $E$ into an $\epsilon$-NFA with $2s$ states. Then we simulate $w$ on this automaton, in $\mathcal{O}(n \cdot s^3)$ steps

# Language membership

We can convert an NFA or an $\epsilon$-NFA into a DFA, and then simulate the input string in time $\mathcal{O}(n)$

The time required by the conversion could be **exponential** in the size of the input FA

This method is used

- when the FA has small size
- when one needs to process several strings for membership with the same FA

# Equivalent states

Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA, and let $p, q \in Q$. We define

$$p \equiv q \;\Leftrightarrow\; \forall w \in \Sigma^* \;:\; \hat{\delta}(p, w) \in F \text{ if and only if } \hat{\delta}(q, w) \in F$$

In words, we require $p, q$ to have equal response to input strings, with respect to acceptance

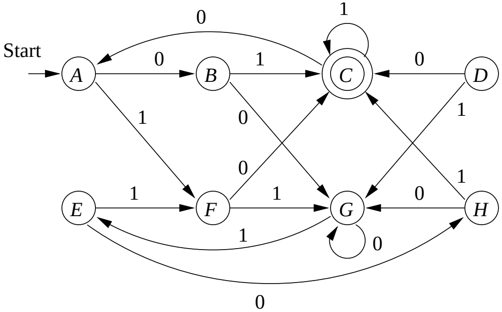
If $p \equiv q$ we say that $p$ and $q$ are **equivalent** states

If $p \not\equiv q$ we say that $p$ and $q$ are **distinguishable** states

Equivalently : $p$ and $q$ are distinguishable if and only if

$$\exists w \;:\; \hat{\delta}(p, w) \in F \text{ and } \hat{\delta}(q, w) \notin F, \text{ or the other way around}$$

# Example




$\hat{\delta}(C,\epsilon) \in \mathcal{F},\ \hat{\delta}(G,\epsilon) \notin \mathcal{F}\ \Rightarrow\ C \not\equiv G$ ($\mathcal{F}$ finale states)

$\hat{\delta}(A,01) = C \in \mathcal{F},\ \hat{\delta}(G,01) = E \notin \mathcal{F}\ \Rightarrow\ A \not\equiv G$

# Example

We prove $A \equiv E$

$\hat{\delta}(A, 1) = F = \hat{\delta}(E, 1)$. Thus $\hat{\delta}(A, 1x) = \hat{\delta}(E, 1x) = \hat{\delta}(F, x)$, $\forall x \in \{0, 1\}^*$

$\hat{\delta}(A, 00) = G = \hat{\delta}(E, 00)$. Thus $\hat{\delta}(A, 00x) = \hat{\delta}(E, 00x) = \hat{\delta}(G, x)$, $\forall x \in \{0, 1\}^*$

$\hat{\delta}(A, 01) = C = \hat{\delta}(E, 01)$. Thus $\hat{\delta}(A, 01x) = \hat{\delta}(E, 01x) = \hat{\delta}(C, x)$, $\forall x \in \{0, 1\}^*$

this works only for this automata, it's not a general algorithm

# State equivalence algorithm

We can compute distinguishable state pairs using the following recursive relation

**Base** If $p \in F$ and $q \notin F$, then $p \not\equiv q$

**Induction** If $\exists a \in \Sigma : \delta(p, a) \not\equiv \delta(q, a)$, then $p \not\equiv q$

We compute distinguishable states by backward propagation

# State equivalence algorithm

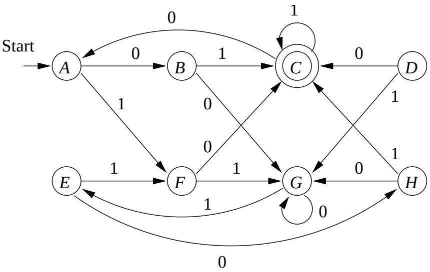
Apply the recursive relation using an **adjacency table** and the following dynamic programming algorithm

- initialize table with pairs that are distinguishable by string $\epsilon$
- for all not yet visited pairs, try to distinguish them using one symbol string: if you reach a pair of **already** distinguishable states, then update table
- iterate until no new pair can be distinguished

# Example

$$\exists a \in \Sigma \ : \ \delta(p, a) \not\equiv \delta(q, a)$$
$$\Rightarrow p \not\equiv q$$

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| B | x | | | | | | |
| C | x | x | | | | | |
| D | x | x | x | | | | |
| E | | x | x | x | | | |
| F | x | x | x | | x | | |
| G | x | x | x | x | x | x | |
| H | x | | x | x | x | x | x |



# Correctness

**Theorem** If $p$ and $q$ are not distinguished by the algorithm, then $p \equiv q$

**Proof** proof is not in the exam

Suppose to the contrary that there is a *bad pair* $\{p, q\}$ such that

- $\exists w \, : \, \hat{\delta}(p, w) \in F, \; \hat{\delta}(q, w) \notin F$, or the other way around
- the algorithm does not distinguish between $p$ and $q$

Each bad pair can be distinguished by some string $w$

We choose the bad pair $p, q$ with the shortest distinguishing string $w$. Let $w = a_1 a_2 \cdots a_n$

# Correctness

Now $w \neq \epsilon$, since otherwise the algorithm would distinguish $p$ from $q$ at the basis step. Thus $n \geqslant 1$

Let us consider states $r = \delta(p, a_1)$ and $s = \delta(q, a_1)$

$r, s$ cannot be a bad pair, otherwise $r, s$ would be identified by a string shorter than $w$

therefore the algorithm must have correctly discovered that $r$ and $s$ are distinguishable. But then the algorithm would distinguish $p$ from $q$ in the inductive part

We conclude that there are no bad pairs, and the theorem holds true $\square$

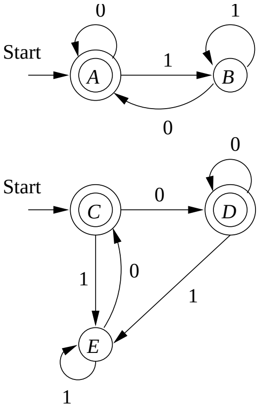
# Regular language equivalence

Let $L$ and $M$ be regular languages (specified by means of some representation)

To test $L \stackrel{?}{=} M$ :

- convert $L$ and $M$ representations into DFAs
- construct the union DFA (never mind if there are two start states)
- apply state equivalence algorithm
- if the two start states are distinguishable, then $L \neq M$, otherwise $L = M$

# Example



# Example

The state equivalence algorithm produces the table

| | $A$ | $B$ | $C$ | $D$ |
|---|---|---|---|---|
| $B$ | $x$ | | | |
| $C$ | | $x$ | | |
| $D$ | | $x$ | | |
| $E$ | $x$ | | $x$ | $x$ |

We have $A \equiv C$, thus the two DFAs are equivalent

Both DFAs recognize language $L(\epsilon + (\mathbf{0} + \mathbf{1})^*\mathbf{0})$

# DFA minimization

Important application of the equivalence algorithm : given DFA as input, produces equivalent DFA with **minimum number of states**

Minimal DFA is **unique**, up to renaming of the states

**Idea** :

- eliminate states that are unreachable from the initial state
- merge equivalent states into an individual state

# Example

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $B$ | $x$ | | | | | | |
| $C$ | $x$ | $x$ | | | | | |
| $D$ | $x$ | $x$ | $x$ | | | | |
| $E$ | | $x$ | $x$ | $x$ | | | |
| $F$ | $x$ | $x$ | $x$ | | $x$ | | |
| $G$ | $x$ | $x$ | $x$ | $x$ | $x$ | $x$ | |
| $H$ | $x$ | | $x$ | $x$ | $x$ | $x$ | $x$ |
| | $A$ | $B$ | $C$ | $D$ | $E$ | $F$ | $G$ |

State partition based on the equivalence relation :
$\{\{A, E\}, \{B, H\}, \{C\}, \{D, F\}, \{G\}\}$

# Example

| | $A$ | $B$ | $C$ | $D$ |
|---|---|---|---|---|
| $B$ | $x$ | | | |
| $C$ | | $x$ | | |
| $D$ | | $x$ | | |
| $E$ | $x$ | | $x$ | $x$ |

State partition based on the equivalence relation :
$\{\{A, C, D\}, \{B, E\}\}$

## Transitivity

**Theorem** If $p \equiv q$ and $q \equiv r$, then $p \equiv r$

**Proof** skip

Suppose to the contrary that $p \not\equiv r$

- Then $\exists w$ such that $\hat{\delta}(p, w) \in F$ and $\hat{\delta}(r, w) \notin F$ or the other way around
- *Case* 1 : $\hat{\delta}(q, w)$ is accepting. Then $q \not\equiv r$
- *Case* 2 : $\hat{\delta}(q, w)$ is not accepting. Then $p \not\equiv q$

Therefore it must be that $p \equiv r$ $\square$

Relation $\equiv$ is reflexive, symmetric and transitive : thus $\equiv$ is an **equivalence relation**

We can talk about equivalence classes

# DFA minimization

To minimize DFA $A = (Q, \Sigma, \delta, q_0, F)$, construct DFA $B = (Q/_\equiv, \Sigma, \gamma, q_0/_\equiv, F/_\equiv)$, where

- elements of $Q/_\equiv$ are the equivalence classes of $\equiv$
- elements of $F/_\equiv$ are the equivalence classes of $\equiv$ composed by states from $F$
- $q_0/_\equiv$ is the set of states that are equivalent to $q_0$
- $\gamma(p/_\equiv, a) = \delta(p, a)/_\equiv$

# DFA minimization

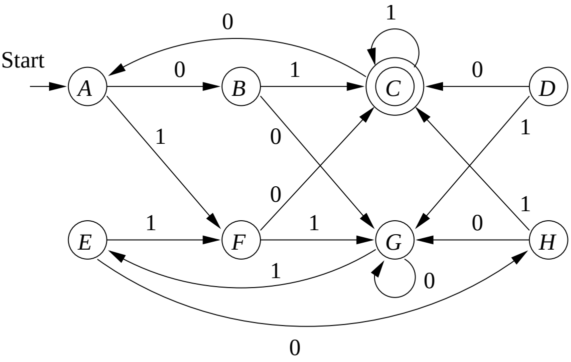
In order for $B$ to be well defined we have to show that

$$\text{If } p \equiv q \text{ then } \delta(p, a) \equiv \delta(q, a)$$

If $\delta(p, a) \not\equiv \delta(q, a)$, then the equivalence algorithm would conclude that $p \not\equiv q$. Thus $B$ is well defined

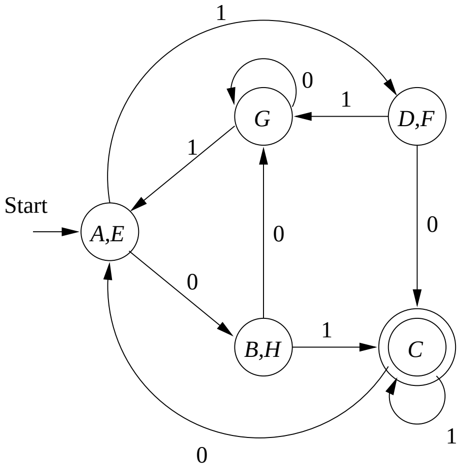
# Example

Minimize



# Example

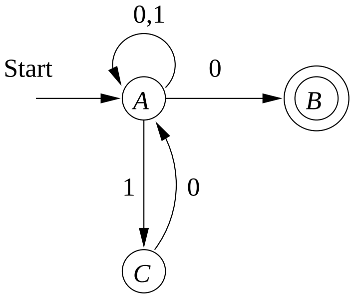
We obtain



# Automata minimization

We **cannot** apply the algorithm to NFAs

**Example** : To minimize




we simply remove state $C$. However, $A \not\equiv C$