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 $L_{01} = \{0^n1^n \mid n \geq 1\}$ 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:

\[
\begin{array}{cc}
\epsilon & p_0 \\
0 & p_1 \\
00 & p_2 \\
\vdots & \vdots \\
0^k & p_k \\
\end{array}
\]

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

Now you can **fool** $A$:
- if $\hat{\delta}(q, 1^i) \not\in F$, then the machine will foolishly reject $0^i1^i$
- if $\hat{\delta}(q, 1^i) \in F$, then the machine will foolishly accept $0^i1^i$

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

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

- $y \neq \epsilon$
- $|xy| \leq n$
- $\forall k \geq 0$, $xy^kz \in L$
Pumping lemma for regular languages

Proof

Suppose $L$ is a regular language

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

Let $w = a_1a_2\cdots a_m \in L$ with $m \geq n$

Let $p_i = \hat{\delta}(q_0, a_1a_2\cdots a_i)$, for each $i = 0, 1, \ldots, n$

There exists $i < j \leq n$ such that $p_i = p_j$
Pumping lemma for regular languages

Let us write \( w = xyz \), where

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

Evidently, \( xy^k z \in L \), for any \( k \geq 0 \)
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 $w = 0^n1^n \in L(A)$.

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

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

and state that, for each $k \geq 0$, we have $xy^kz \in L(A)$.

\[
w = \underbrace{000 \cdots 00}_{x} \underbrace{\cdots 0111 \cdots 11}_{z}
\]
Example

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

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

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

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
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 $|w| \geq n$
- P2 picks a factorization $w = xyz$, with $|xy| \leq n$, $y \neq \epsilon$ 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 $p \geq n + 2$ be some prime number. Thus $1^p \in L_{pr}$.

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

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

such that, for each $k \geq 0$, we have $xy^kz \in L(A)$.
Example

Let $|y| = m \geq 1$

$$w = \underbrace{111 \cdots}_{x} \underbrace{\cdots 1}_{y} \underbrace{1111 \cdots 11}_{z}$$

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 $1 + m > 1$
- $m = |y| \leq |xy| \leq n$, $p \geq n + 2$, thus $p - m \geq n + 2 - m \geq n + 2 - n = 2$

We have derived a **contradiction**
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 $L$ and $M$, $L \cup M$ 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.
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
**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 $L(B) = \overline{L}$

□
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, $L \cap M = \overline{L} \cup \overline{M}$

We already know that regular languages are closed under complement and union
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 Diagram]

- **Start**
- **Input $a$**
- **$A_L$**
- **$A_M$**
- **AND**
- **Accept**
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

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

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
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.
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
Closure under reverse operator

**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 $a$, 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 \( \exists k : w = w_1w_2 \cdots w_k, w_i \in L(F) \)
  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 \( L(E^R) = (L(E))^R \)
State whether the following claims hold true, and motivate your answer

- the intersection of a non-regular language and a finite language is always a regular language
- the intersection of a non-regular language $L_1$ and an infinite regular language $L_2$ is never a regular language
- every subset of a non-regular language is a non-regular language
Superset and subset

Assume $L$ is a regular language. We cannot say anything about languages $L'$ and $L''$ with $L' \subseteq L$ and $L'' \supseteq L$

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 $h : \Sigma \rightarrow \Delta^*$

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 $10^*1$. Then $h(L)$ is the language associated with the regular expression $(ab)^*$
Closure under homomorphism

**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 $a$ with $a_1a_2\cdots a_k$, under the assumption that

- $a \in \Sigma$
- $h(a) = a_1a_2\cdots a_k$, $k \geq 0$

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

$E = a$ with $a \in \Sigma$. Let $h(a) = a_1a_2\cdots a_k$, $k \geq 0$. Then $L(a) = \{a\}$ and thus $h(L(a)) = \{a_1a_2\cdots a_k\}$

The regular expression $h(a)$ is $a_1a_2\cdots a_k$. Then $L(h(a)) = \{a_1a_2\cdots a_k\} = h(L(a))$
Closure under homomorphism

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

\[
L(h(E)) = L(h(F + G)) \\
= L(h(F) + h(G)) \quad \text{\( h \) defined over regex} \\
= L(h(F)) \cup L(h(G)) \quad \text{\(+ definition\)} \\
= h(L(F)) \cup h(L(G)) \quad \text{inductive hypothesis for \( F, G \)} \\
= h(L(F) \cup L(G)) \quad \text{\( h \) defined over languages} \\
= h(L(F + G)) \quad \text{\(+ definition\)} \\
= h(L(E))
\]
Closure under homomorphism

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

\[
L(h(E)) = L(h(F.G))
\]
\[
= L(h(F).h(G)) \quad h \text{ defined over regex}
\]
\[
= L(h(F)).L(h(G)) \quad \text{. definition}
\]
\[
= h(L(F)).h(L(G)) \quad \text{inductive hypothesis for } F, G
\]
\[
= h(L(F).L(G)) \quad h \text{ defined over languages}
\]
\[
= h(L(F.G)) \quad \text{. definition}
\]
\[
= h(L(E))
\]
Closure under homomorphism

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

$$L(h(E)) = L(h(F^*)) = L([h(F)]^*) = \bigcup_{k \geq 0} [L(h(F))]^k = \bigcup_{k \geq 0} h([L(F)]^k) = h(L(F^*)) = h(L(E))$$

$h$ defined over regex

$*$ definition

Inductive hypothesis for $F$

$h$ definition over languages

$h$ definition over languages

$*$ definition
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
From $\epsilon$-NFA to DFA

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

The resulting DFA has $2^n$ states. For each state $S$ and each $a \in \Sigma$ we compute $\delta(S, a)$ in time $O(n^3)$. In total, the computation takes $O(n^3 \cdot 2^n)$ 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 $O(n^3 \cdot s)$ 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 $O(n)$
- At each node in the tree, we build new nodes and arcs in time $O(1)$ and use pointers to previously built structure, avoiding copying
- Grand total time is $O(n)$, 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

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

$L(A) \neq \emptyset$ for FA $A$ if and only if at least one final state is \textbf{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 $O(n^2)$

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

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

**Base**

- $E = \epsilon$ or else $E = a$. 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
- $E = F \cdot G$. 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 $\epsilon \in L(E)$
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 $O(n \cdot s^3)$ steps.

Alternatively, we can pre-process $A$ by calculating $ECLOSE(p)$ for $s$ states, in time $O(s^3)$. 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 $O(s^2)$
- compute the $\epsilon$-closure for the successor states in time $O(s^2)$

This takes time $O(n \cdot s^2)$.
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 $O(n \cdot s^3)$ steps.
Language membership

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

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

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

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

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

\[ \hat{\delta}(C, \epsilon) \in \mathcal{F}, \hat{\delta}(G, \epsilon) \notin \mathcal{F} \Rightarrow C \neq G \quad (\mathcal{F} \text{ finale states}) \]

\[ \hat{\delta}(A, 01) = C \in \mathcal{F}, \hat{\delta}(G, 01) = E \notin \mathcal{F} \Rightarrow A \neq G \]
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 $\hat{\delta}(A, 01x) = \hat{\delta}(E, 01x) = \hat{\delta}(C, x)$, $\forall x \in \{0, 1\}^*$
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 $\exists a \in \Sigma : \delta(p, a) \neq \delta(q, a)$, 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

\[ \exists a \in \Sigma : \delta(p, a) \neq \delta(q, a) \]
\[ \Rightarrow p \neq q \]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

Automata, Languages and Computation

Chapter 4
Correctness

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

**Proof**
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 $w = a_1a_2\cdots a_n$
Correctness

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

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 $L \equiv M$:
- 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

Automata, Languages and Computation

Chapter 4
The state equivalence algorithm produces the table

\[
\begin{array}{c|cccc}
\text{A} & \text{B} & \text{C} & \text{D} \\
\hline
\text{B} & x & & & \\
\text{C} & & x & & \\
\text{D} & & & x & \\
\text{E} & x & x & x & \\
\end{array}
\]

We have \( A \equiv C \), thus the two DFAs are equivalent.

Both DFAs recognize language \( L(\epsilon + (0 + 1)^*0) \)
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
State partition based on the equivalence relation:
\[
\{|\{A, E\}, \{B, H\}, \{C\}, \{D, F\}, \{G\}\}
\]
Example

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**
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 $p \equiv r$  

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 $q_0$
- $\gamma(p/\equiv, a) = \delta(p, a)/\equiv$
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 $\delta(p, a) \neq \delta(q, a)$, then the equivalence algorithm would conclude that $p \not\equiv q$. Thus $B$ is well defined.
Example

Minimize

\[
\begin{array}{c}
\text{Start} \\
A \\
B \\
C \\
D \\
E \\
F \\
G \\
H
\end{array}
\]

0 1 0 1 0 0 1 0
0 0 1 0 1 0 0 1
1 0 0 0 1 1 0 0
1 1 0 1 0 0 1 0
0 1 1 0 1 0 0 1

Automata, Languages and Computation

Chapter 4
Example

We obtain

![Automata diagram with transitions labeled 0 and 1]

Start

A,E

G

D,F

B,H C

A,E

G

D,F

B,H C

0

1

0

1

0

1

0

1

0

1
We **cannot** apply the algorithm to NFAs

**Example**: To minimize

![Diagram](image)

we simply remove state $C$. However, $A \neq C$