- Chapter 3

REGULAR EXPRESSIONS

REGULAR EXPRESSIONS

- an algebraic description
- regular expressions serve as the input language for many systems that process strings
- Examples
  - Search commands such as the UNIX *grep* or equivalent commands for finding strings that one sees in Web browsers or text-formatting systems.
    - These systems use a regular-expression-like notation for describing patterns that the user wants to find in a life.
    - Different search systems convert the regular expression into either a DFA or an NFA and simulate that automaton on the file being searched.
  - Lexical-analyzer generators such as Lex or Flex.
    - a lexical analyzer is the component of a compiler that breaks the source program into logical units (called tokens) of one or more characters that have a shared significance.
    - Examples of tokens include keywords (eg. while), identifiers (eg. any letter followed by zero or more letters and/or digits) and signs such as + or <=.
    - A lexical-analyzer generator accepts descriptions of the forms of tokens which are essentially regular expressions, and produces a DFA that recognizes which token appears next on the input

The Operators of Regular Expressions

- Regular expressions denote languages.
  - the regular expression $01^* + 10^*$ denotes the language consisting of all strings that are either a single 0 followed by any number of 1's or a single 1 followed by any number of 0's

The Operators of Regular Expressions

- Three operations on languages that the operators of regular expressions represent
  - The *union* of two languages L and M
    - is denoted $L \cup M$
    - is the set of strings that are in either L or M, or both
    - if $L = \{001, 10, 111\}$ and $M = \{\varepsilon, 001\}$, then $L \cup M = \{\varepsilon, 10, 001, 111\}$
  - The *concatenation* of languages L and M
    - is denoted $L.M$ or LM
    - is the set of strings that can be formed by taking any string in L and concatenating it with any string in M.
    - if $L = \{001, 10, 111\}$ and $M = \{\varepsilon, 001\}$, then LM = ???????
  - The *closure* (or *star*) of a language L
    - is denoted L^*
    - represents the set of those strings that can be formed by taking any number of strings from L
    - $L^* = \bigcup_{i \geq 0} L^i$, where $L^0 = \{\varepsilon\}, L^1 = L, L^i = L.L \dots \dots L$, for i > 1

Building Regular Expressions

- The algebra of regular expressions
  - using constants and variables that denote languages
  - operators: union, dot, and star
  - we can describe the regular expressions recursively
  - for each regular expression E, we describe the language it represents, which we denote $L(E)$

Building Regular Expressions

- BASIS: The basis consists of three parts:
  - The constants ε and ∅ are regular expressions
    - denotes the languages {ε} and ∅
    - $L(\varepsilon) = \{\varepsilon\}$, and $L(\emptyset) = \emptyset$
  - If **a** is any symbol, then **a** is a regular expression
    - denotes the language {a}
    - $L(\mathbf{a}) = \{a\}$
    - we use boldface font to denote an expression corresponding to a symbol
  - A variable, usually capitalized and italic such as *L* is a variable representing any language
    S

Building Regular Expressions

- INDUCTION: There are four parts to the inductive step
  - If *E* and *F* are regular expressions then
    - *E* + *F* is a regular expression denoting the union of L(E) and L(F)
      - $L(E + F) = L(E) \cup L(F)$
    - *EF* is a regular expression denoting the concatenation of L(E) and L(F)
      - $L(EF) = L(E)\, L(F)$
    - *E** is a regular expression denoting the closure of L(E)
      - $L(E^*) = (L(E))^*$
    - *(E)*, a parenthesized *E*, is also a regular expression denoting the same language as *E*
      - $L((E)) = L(E)$

Example 1: Write a regular expression for the set of strings that consist of alternating 0's and 1's.

- First, develop a regular expression for the language consisting of the single string *01*
- We can then use the star operator to get an expression for all strings of the form *0101.....01*
- The basis rule for regular expressions tells us that **0** and **1** are expressions denoting the languages {0}and {1}, respectively.
- If we concatenate the two expressions, we get a regular expression for the language {01}; this expression is **01**
- To get all strings consisting of zero or more occurrences of 01, we use the regular expression **(01)***
- However, L((**01**)*) is not exactly the language that we want.
  - It includes only those strings of alternating 0's and 1's that begin with 0 and end with 1.
  - We also need to consider the possibility that there is a 1 at the beginning and/or a 0 at the end
  - (10)* represents those alternating strings that begin with 1 and end with 0
  - **0**(10)*can be used for strings that both begin and end with 0
  - **1**(01)* serves for strings that begin and end with 1
- The entire regular expression is: **(01)* + (10)* + 0(10)* + 1(01)***

Example 1: Write a regular expression for the set of strings that consist of alternating 0's and 1's.

- There is another approach that yields a regular expression that looks rather different and is also somewhat more succinct.

  $(\varepsilon + \mathbf{1})(\mathbf{01})^*((\varepsilon + \mathbf{0})$

Precedence of RegularExpression Operators

1. The star operator (*) is of highest precedence.
2. Next comes the concatenation or “dot” operator
3. Finally, union operator.

Example 2

- Determine the language the expression **01*** + **1**
  - is grouped (**0**(**1***)) + **1**
  - the language of the given expression
    - is the string 1 plus all strings consisting of a 0 followed by any number of 1's (including none)
    - L(**01*** + **1**) = {1, 0, 01, 011, 0111, …..}

Example 3

- Determine the language the expression $0(1^* + 1)$

Exercises

<u>Exercise 1</u>: Write regular expressions for the following languages

a) The set of strings over alphabet {a, b, c} containing at least one *a* and at least one *b.*

b) The set of strings of 0's and 1's whose third symbol from the right end is 1.

c) The set of strings of 0's and 1's with at most one pair of consecutive 1's

BÀI 1. Xây dựng BTCQ biểu diễn (chỉ định) các ngôn ngữ sau:

a) Tập hợp các xâu trên {a,b} chứa ít nhất xâu aba
b) Tập hợp các xâu trên {a,b} có độ dài chia hết cho 3
c) Tập hợp các xâu trên {a, b, c} chỉ chứa 1 ký hiệu a, còn lại là các ký hiệu b và c
d) Tập hợp các số nhị phân có tận cùng là 11
e) Tập hợp các số nhị phân có giá trị là các số chẵn từ 2 đến 16.
f) Tập hợp các số nguyên không dấu chia hết cho 5
g) Tập hợp các xâu trên {0, 1} bắt đầu và kết thúc với kí tự giống nhau
h) Tập hợp các xâu trên {0, 1}có ít nhất 3 kí tự, kí tự thứ ba là 0

Exercises

<u>Exercise 2</u>: Write regular expressions for the following languages

a) The set of strings of 0’s and 1’s whose number of 0’s is divisible by five

b) The set of all strings of 0’s and 1’s not containing 101 as a substring

c) The set of strings of 0’s and 1’s whose number of 0’s is divisible by Five and whose number of 1’s is even.

Exercises

Exercise 3: Give English descriptions of the languages of the following regular expressions

a) $(0^*1^*)^*000(0+1)^*$

b) $(0+10)^*1^*$

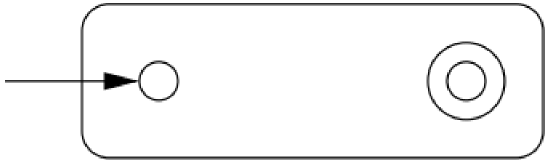
c) $(1+\varepsilon)(00^*1)^*0^*$

Converting Regular Expressions to Automata

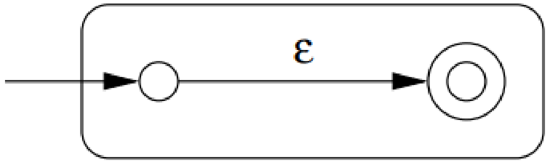
- A regular expression that gives a "picture" of the pattern we want to recognize
- A regular expression is the medium of choice for applications that search for patterns in text.
- The regular expressions are then compiled, behind the scenes, into deterministic or nondeterministic automata, which are then simulated to produce a program that recognizes patterns in text

Converting Regular Expressions to Automata

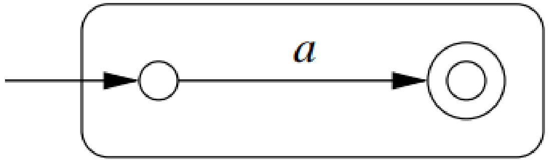
- **Theorem 3.1**. Every language defined by a regular expression is also defined by a fnite automator
  - For regular expression ∅:



  - For regular expression ε:

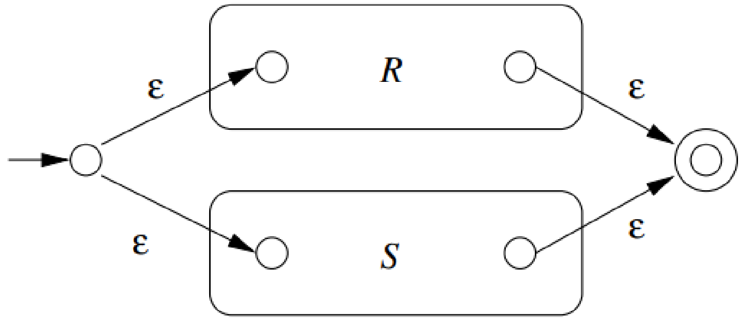


  - For regular expression **a**:

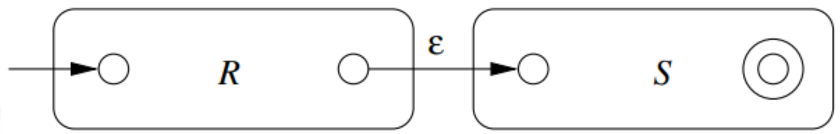


Converting Regular Expressions to Automata

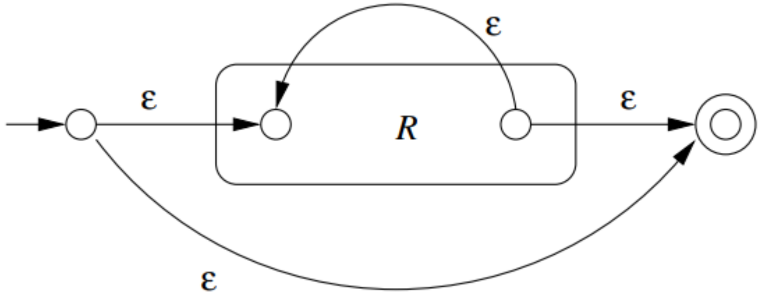
- Where R, S are regular expressions
  - For R + S:



  - For RS:



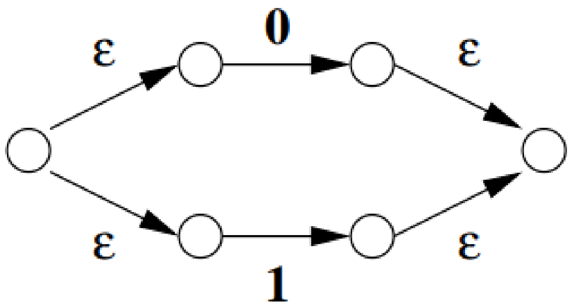
  - For R*:



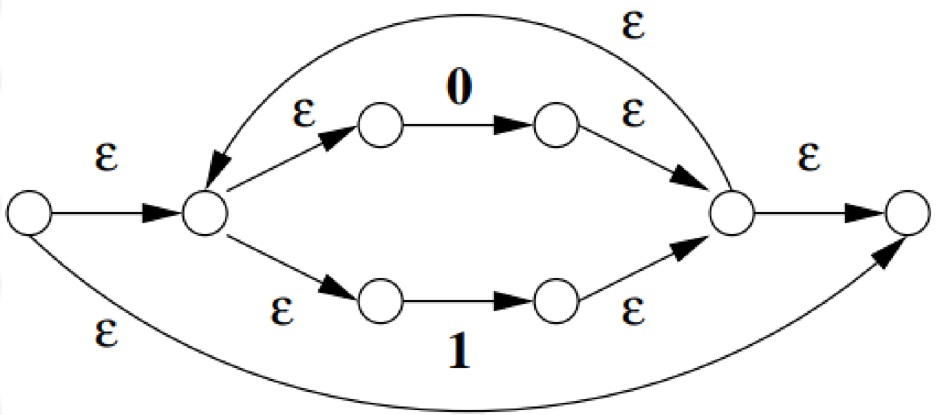
Example 1.

Convert the regular expression $(0+1)^*1(0+1)$ an ε-NFA

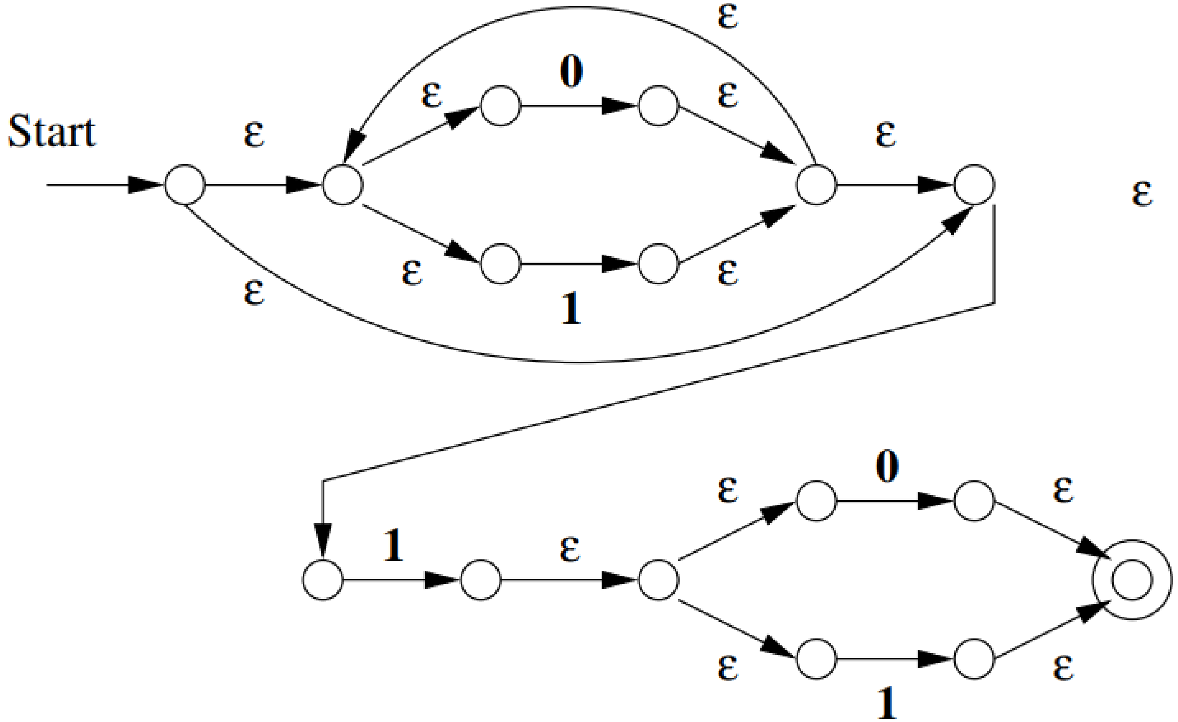
- For 0+1



- For $(0+1)^*$



- For $(0+1)^*1(0+1)$



Example 1.

Convert the regular expression $(0+1)^*1(0+1)$ an ε-NFA

- After removing ε-transitions

Exercises

- **<u>Exercise 1</u>**. Convert the following regular expressions to NFA's with ε-transitions.
  - a) (0+1)01
  - b) $00(0+1)^*$
  - c) 01^*

Exercises

- **Exercise 2**. Eliminate ε–transitions from your ε–NFA's of **Exercise 1**.

Applications of Regular Expressions

- A regular expression that gives a "picture" of the pattern we want to recognize
- A regular expression is the medium of choice for applications that search for patterns in text.
- The regular expressions are then compiled, behind the scenes, into deterministic or nondeterministic automata, which are then simulated to produce a program that recognizes patterns in text

Regular Expressions in UNIX

- UNIX notation for extended regular expressions
  - The symbol . (dot) stands for “any character”.
  - The sequence [$a_1a_2\ldots.a_k$] stands for the regular expression $a_1+a_2+\ldots.+a_k$
  - Between the square braces we can put a range of the form *x-y* to mean all the characters from x to y in the ASCII sequence
    - [0-9]: the digits
    - [A-Z]: the upper-case letters
    - [A-Za-z0-9]: the set of all letters and digits
    - [-+.0-9]: the set of digits, plus the dots, plus and minus signs
  - There are special notations for several of the most common classes of characters
    - [:digits:] is the set of ten digits the same as [0-9]
    - [:alpha:] stands for any alphabetic character as does [A-Za-z]
    - [:alnum:] stands for the digits and letters (alphabetic and numeric characters), as does [A-Za-z0-9]

Regular Expressions in UNIX

- In addition, there are several operators that are used in UNIX regular expressions that we have not encountered previously
  - The operator | is used in place of + to denote union
  - The operator ? means "zero or one of".
    - R? in UNIX is the same as ε+R
  - The operator $^+$ means "one or more of".
    - R^+ in UNIX is shorthand for RR^* in our notation.
  - The operator {n} means "n copies of".
    - R{5} in UNIX is shorthand for RRRRR

Lexical Analysis

- One of the oldest applications of regular expressions was in specifying the component of a compiler called a “lexical analyzer”.
- This component scans the source program and recognizes all *tokens*, those substrings of consecutive char acters that belong together logically
  - Keywords and identifiers are common examples of tokens, but there are many others
- The UNIX command lex and its GNU version flex, accept as input a list of regular expressions in the UNIX style, each followed by a bracketed section of code that indicates what the lexical analyzer is to do when it finds an instance of that token.

Lexical Analysis

```
else                 {return(ELSE);}

[A-Za-z][A-Za-z0-9]*    {code to enter the found identifier
                      in the symbol table;
                      return(ID);
                     }

>=                   {return(GE);}

=                    {return(ASGN);}

...
```

- The UNIX command **lex** and its GNU version **flex**, accept as input a list of regular expressions in the UNIX style, each followed by a bracketed section of code that indicates what the lexical analyzer is to do when it finds an instance of that token.

Lexical Analysis

- Commands such as **lex** and **flex** have been found extremely useful because the regular-expression notation is exactly as powerful as we need to describe tokens.
- These commands are able to use the regular-expression-to-DFA conversion process to generate an efficient function that breaks source programs into tokens.
- Further, if we need to modify the lexical analyzer for any reason, it is often a simple matter to change a regular expression or two, instead of having to go into mysterious code to fix a bug.

Finding Patterns in Text

- Automata could be used to search efficiently for a set of words in a large repository such as the Web.
- The general problem for which regular-expression technology has been found useful is the description of a vaguely defined class of patterns in text.
- By using regular expression notation it becomes easy to describe the patterns at a high level, with little effort, and to modify the description quickly when things go wrong.
- A “compiler” for regular expressions is useful to turn the expressions we write into executable code.

Finding Patterns in Text

- Suppose that we want to scan a very large number of Web pages and detect addresses.
- We might simply want to create a mailing list.
- Or, perhaps we are trying to classify businesses by their location so that we can answer queries like “find me a restaurant within 10 minutes drive of where I am now”.

Finding Patterns in Text

- We shall focus on recognizing street addresses in particular.
- What is a street address?
  - A street address will probably end in “Street” or its abbreviation “St”
  - However some people live on “Avenues” or “Roads,” and these might be abbreviated in the address as well.

=> we might use as the ending for our regular expression something like:

`Street|St\.|Avenue|Ave\.|Road|Rd|.`

Finding Patterns in Text

- The designation such as Street must be preceded by the name of the street.
- The name is a capital letter followed by some lower-case letters
  - We can describe this pattern by the UNIX expression $[A\text{-}Z][a\text{-}z]^*$
- However, some streets have a name consisting of more than one word, such as Rhode Island Avenue in Washington DC
  - ‘$[A\text{-}Z][a\text{-}z]^*(\ \ [A\text{-}Z][a\text{-}z]^*)^*$’

Finding Patterns in Text

- Now we need to include the house number as part of the address.
  - Most house numbers are a string of digits.
  - However some will have a letter following, as in “123A Main St”
    - the expression we use for numbers has an optional capital letter following:
      `[0-9]+[A-Z]?`

- The entire expression we have developed for street addresses is:

```
'[0-9]+[A-Z]?  [A-Z][a-z]*(  [A-Z][a-z]*)*
(Street|St\.|Avenue|Ave\.|Road|Rd|.)'
```

Finding Patterns in Text

- The entire expression we have developed for street addresses is:

  ‘[0-9]⁺[A-Z]? [A-Z][a-z]*([A-Z][a-z]*)*
  (Street|St\.|Avenue|Ave\.|Road|Rd|.)’

- If we work with this expression we shall do fairly well. However we shall eventually discover that we are missing:
  - Streets that are called something other than a street, avenue, or road.
    - For example, we shall miss “Boulevard”, “Place”, “Way,” and their abbreviations
  - Street names that are numbers or partially numbers, like “42nd Street”
  - Post-Office boxes and rural-delivery routes
  - Street names that don’t end in anything like “Street”.
    - An example is El Camino Real in Silicon Valley

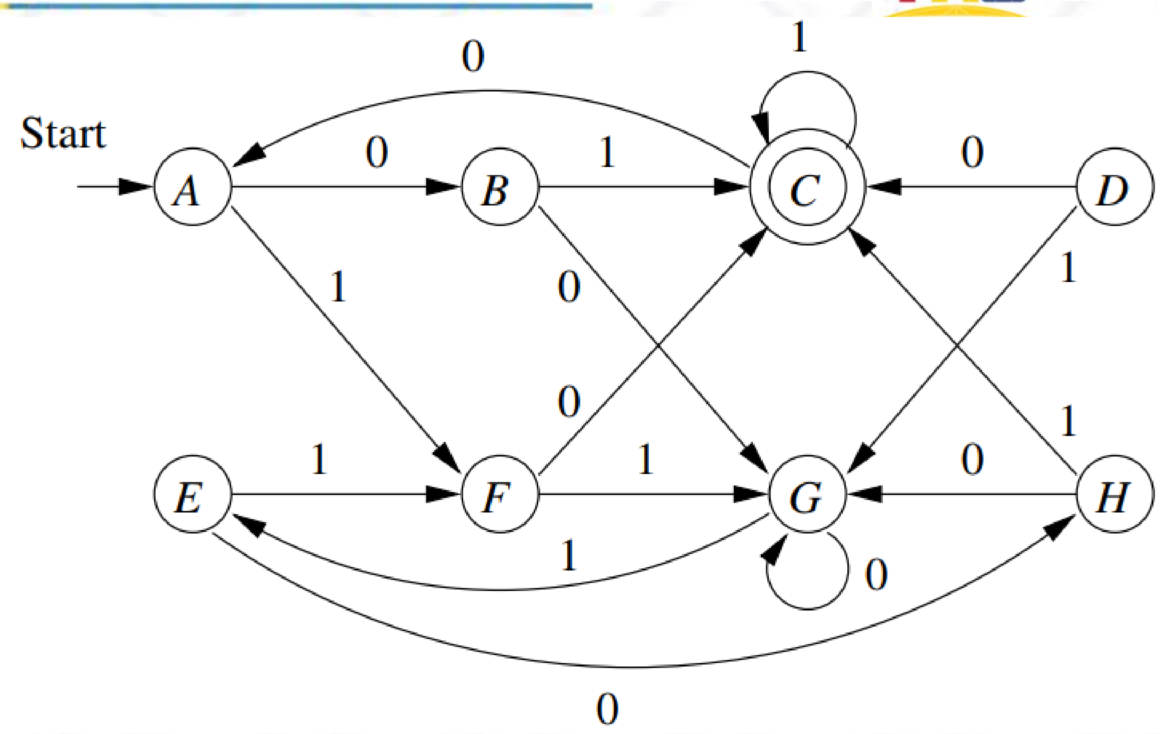
Minimization of DFA’s

- For each DFA we can find an equivalent DFA that has as few states as any DFA accepting the same language.

Minimization of DFA's

- Equivalence of States
  - We say that states p and q are *equivalent* if:
    - For all input strings w, $\hat{\delta}(p, w)$ is an accepting state if and only if $\hat{\delta}(q, w)$ is an accepting state
  - If two states are not *equivalent* then we say they are *distinguishable*.
    - State p is *distinguishable* from state q if there is at least one string w such that one of $\hat{\delta}(p, w)$ and $\hat{\delta}(q, w)$ is accepting, and the other is not accepting.

Minimization of DFA's



- Consider states A and G
  - $\hat{\delta}(A, 01)$ = C - accepting state
    $\hat{\delta}(G, 01)$ = E – not accepting state

  ⇒A and G are not equivalent.
- Consider states A and E
  - $\hat{\delta}(A, 1)$ = F, $\hat{\delta}(E, 1)$ = F
  - $\hat{\delta}(A, 1x) = \hat{\delta}(E, 1x)$
  - $\hat{\delta}(A, 0)$ = B, $\hat{\delta}(E, 1)$ = H
  - $\hat{\delta}(A, 01)$ = C, $\hat{\delta}(E, 01)$ = C

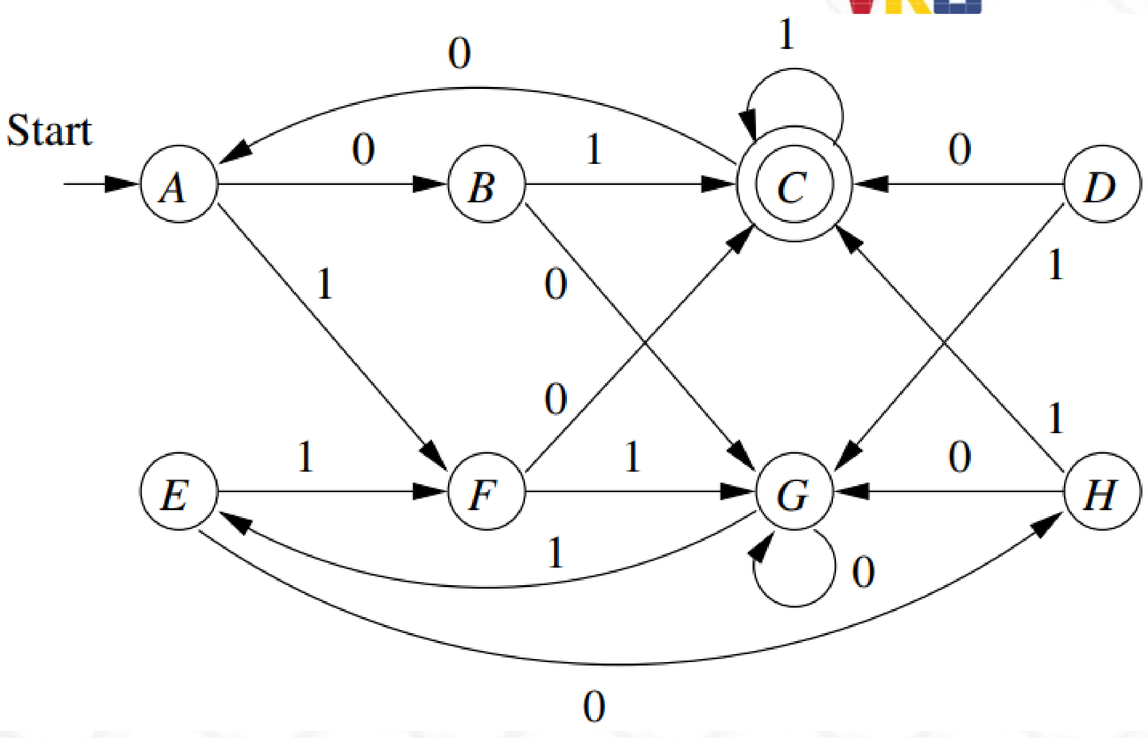
  => A and E are equivalent.

Minimization of DFA’s

- Equivalence of States
  - To find states that are equivalent, we make our best efforts to find pairs of states that are distinguishable.
    - then any pair of states that we do not find distinguishable are equivalent.
  - Table-filling algorithm - a recursive discovery of distinguishable pairs in a DFA $A=\{Q, \Sigma, \delta, q_0, F\}$
    - If *p* is an accepting state and *q* is nonaccepting then the pair *{p, q}* is distinguishable.
    - Let *p* and *q* be states such that for some input symbol *a*, $r = \delta(p, a)$ and $s = \delta(q, a)$ are a pair of states known to be distinguishable. Then *{p, q}* is a pair of distinguishable states.
  - If two states are not distinguished by the table-filling algorithm then the states are equivalent.

Minimization of DFA's

- Execute the table-filling algorithm on the DFA



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*

- *x* indicates pairs of distinguishable states
- the blank squares indicate those pairs that have been found equivalent

Minimization of DFA’s

- The algorithm is as follows:
  1. First, eliminate any state that cannot be reached from the start state.
  2. Then, partition the remaining states into blocks, so that:
     1. all states in the same block are equivalent, and
     2. no pair of states from different blocks are equivalent
     3. Theorem 3.2, below, shows that we can always make such a partition

Minimization of DFA's

- **<u>Theorem 3.2</u>:** The equivalence of states is transitive. That is, if in some DFA A={Q, ∑, δ, q_0, F} we find that states *p* and *q* are equivalent, and we also find that *q* and *r* are equivalent, then it must be that *p* and *r* are equivalent.

- **<u>Theorem 3.3</u>:** If we create for each state *q* of a DFA a *block* consisting of *q* and all the states equivalent to *q*, then the different blocks of states form a *partition* of the set of states.

That is, each state is in exactly one block. All members of a block are equivalent, and no pair of states chosen from different blocks are equivalent.

Minimization of DFA's

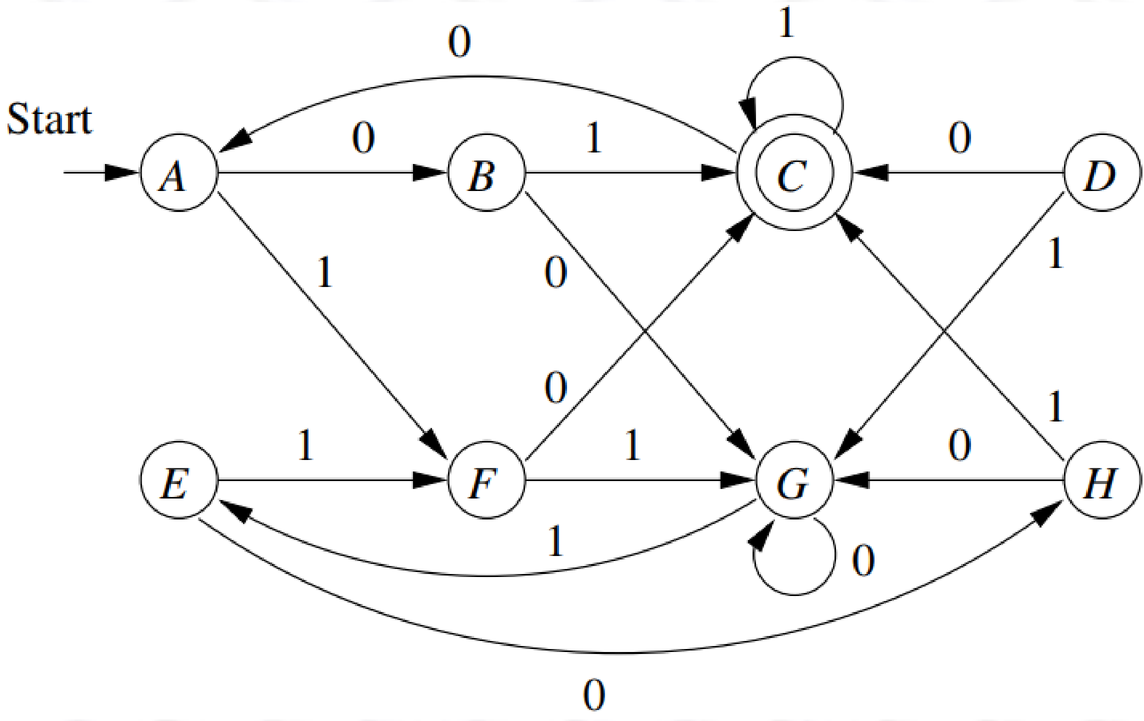
Algorithm for minimizing a DFA $A=(Q, \Sigma, \delta, q_0, F)$

1. Use the table-filling algorithm to find all the pairs of equivalent states.
2. Partition the set of states Q into blocks of mutually equivalent states by the method described above.
3. Construct the minimum-state equivalent DFA B by using the blocks as its states.

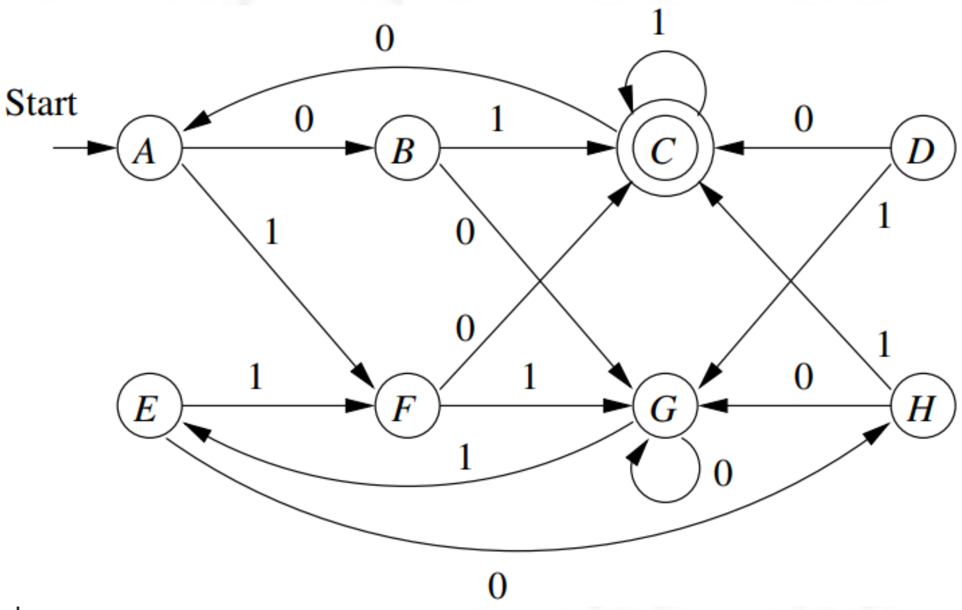
Example: Minimize the DFA

1. Use the table-filling algorithm to find all the pairs of equivalent states.

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



Example: Minimize the DFA



2. Partition the set of states Q into blocks of mutually equivalent states.

   {A, E}

   {B, H}

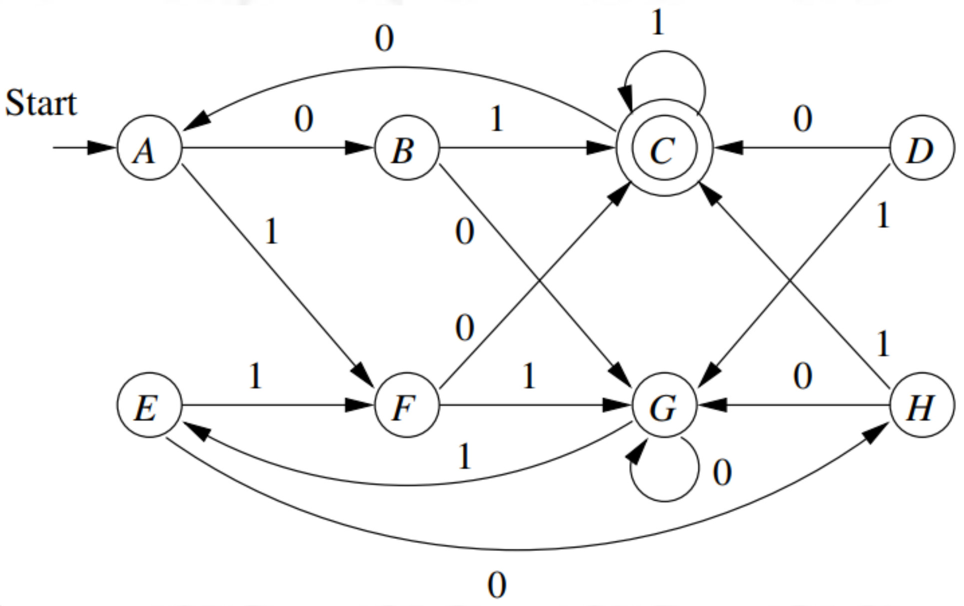
   {D, F}

   {C}

   {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*
	A	*B*	*C*	*D*	*E*	*F*	*G*

Example: Minimize the DFA



3. Construct the minimum-state equivalent DFA B by using the blocks as its states.

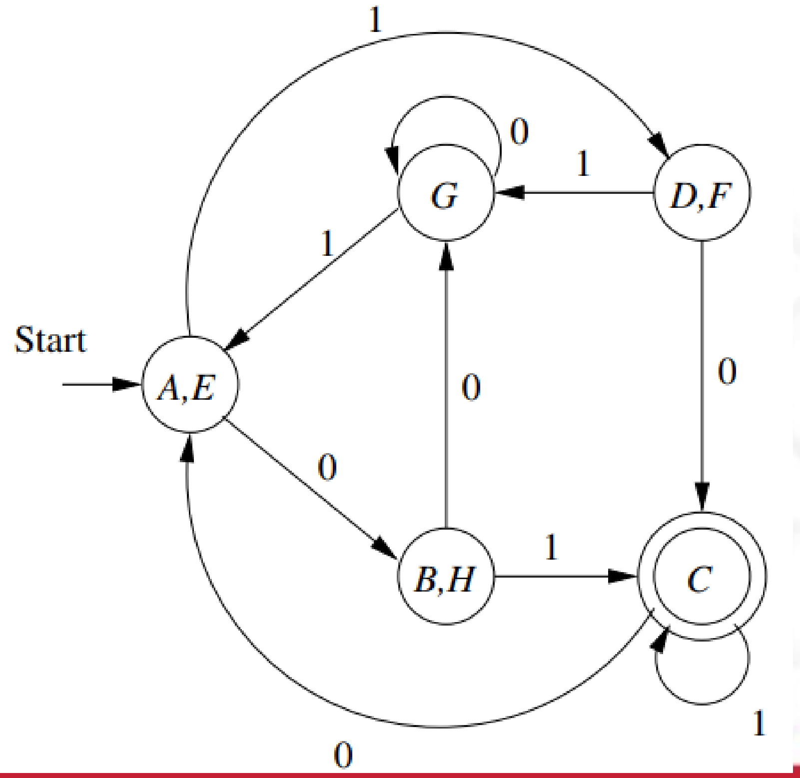
{A, E}

{B, H}

{D, F}

{C}

{G}



Exercises

<u>Exercise 1</u>: Given a DFA

1. Draw the table of distinguishabilities for this automaton.
2. Construct the minimum state equivalent DFA.

	0	1
$\rightarrow A$	B	A
B	A	C
C	D	B
$*D$	D	A
E	D	F
F	G	E
G	F	G
H	G	D

Exercises

Exercise 2: Given another DFA

1. Draw the table of distinguishabilities for this automaton.
2. Construct the minimum state equivalent DFA.

	0	1
$\rightarrow A$	B	E
B	C	F
$*C$	D	H
D	E	H
E	F	I
$*F$	G	B
G	H	B
H	I	C
$*I$	A	E

<u>BÀI 1.</u> Cực tiểu hóa DFA sau:

δ	0	1
→A		B
B	C	D
*C	C	C
*D	D	D