Chapter 4

Properties of Regular Languages

(2015/10/15)

Pasbag, Turkey

Outline

4.1 Proving Languages Not to Be Regular

4.2 Closure Properties of Regular Languages

4.3 Decision Properties of Regular Languages

4.4 Equivalence and Minimization of Automata

4.1 Proving Languages Not to Be Regular

4.1.1 Pumping Lemma for Regular Languages

- It is first noted that *not* every language is regular.
  - For example, $L_{01} = \{0^n1^n \mid n \geq 1\}$ is not a regular language.

- How to prove that a language is not regular?
  - Answer: use the *pumping lemma*.

- In the sequel, we abbreviate "regular language" as "RL."

- **Theorem 4.1** (*Pumping lemma for RL's*) ---

  Let L be an RL. Then, there exists an integer constant n (depending on L) such that for every string w in L with $|w| \geq n$, we can break w into three substrings, $w = xyz$, such that:

  - $y \neq \varepsilon$ (i.e., y has at least one symbol);
  - $|xy| \leq n$; and
  - for all $k \geq 0$, the "pumped" string xy^kz is also in L.

  Proof. See the textbook.

- An alternative way of describing the pumping lemma (supplemental) ---

  The pumping lemma may be rewritten more precisely by mathematical notations as

  $$(\forall L)(\exists n)(\forall w)(w \in L, |w| \geq n \Rightarrow (\exists x, y, z)(w = xyz, |xy| \leq n, |y| \geq 1, (\forall k)(xy^kz \in L))).$$

4.1.2 Applications of Pumping Lemma

- The pumping lemma may be used for proving "a given language is *not* an RL," instead of proving "*is* an RL."

- **Example 4.2** ---

  Prove that the language $L_{\text{eq}} = \{w \mid w$ has equal numbers of 0's and 1's$\}$ is *not* an RL.

  Proof (by contradiction):
  - Assume that L_{eq} is an RL.
  - The *pumping lemma* says that there exists an integer n such that for every string w in L with length $|w| \geq n$, w can be broken into 3 pieces, $w = xyz$, such that the three conditions mentioned in Theorem 4.1 hold.
  - In particular, pick the string $w = 0^n1^n$ whose length is $2n$.
  - We know that w is in L_{eq}.
  - Because $|w| = 2n > n$, by Theorem 4.1, *string w* can be broken into 3 pieces, $w = xyz$, so that
    - $y \neq \varepsilon$;
    - $|xy| \leq n$;
    - for all $k \geq 0$, $xy^kz \in L$.
  - Also, the inequality $|xy| \leq n$ says that xy consists of all 0's because $w = 0^n1^n$.
  - Furthermore, $y \neq \varepsilon$ says y has at least one 0. (A)

- ♦ Now, take k to be 0 and the *pumping* in the 3rd condition says that the following is true:

$$xy^0z = x\varepsilon z = xz \in L. \qquad \text{(B)}$$

- ♦ However, by (A) at least one 0 disappears when y was "pumped" out.
- ♦ This means that the resulting string *xz cannot have equal numbers of 0's and 1's*, *i.e.,* $xz \notin L$. *Contradictive to* (B) *above!*
- ♦ So, the original assumption "L_{eq} is an RL" is false (according to principle of "proof by contradiction."). Done!

4.2 Closure Properties of RL's

- ■ The term "closure" means "being closed" in the same type of language domain, such as RL's.
- ■ We will prove a set of "closure" theorems of the form ---

  "if certain languages are regular, and a language L is formed from them by certain operations, then L is also regular."

- ■ Language operations for the above statement to be true include:
  - ♦ *Union*
  - ♦ *Concatenation*
  - ♦ *Closure (star)*

  - ♦ *Intersection*
  - ♦ *Complementation*
  - ♦ *Difference*
  - ♦ *Reversal*

  - ♦ *Homomorphism*
  - ♦ *Inverse homomorphism*

<u>4.2.1~4.2.4 (for proofs, see the textbook)</u>

- ■ Let L and M be two RL's over alphabet Σ. We have the following theorems.
  - ♦ **Theorem 4.4** --- the union $L \cup M$ is an RL.
  - ♦ **Theorem 4.5** --- the complement $\overline{L} = \Sigma^* - L$ is an RL (Σ^* is the universal language)
  - ♦ **Theorem 4.8** --- the intersection $L \cap M$ is an RL.
  - ♦ **Theorem 4.10** --- the difference $L - M$ is an RL.
  - ♦ **Theorems** --- the concatenation LM and the closure L^* are RL's (no theorem numbers; mentioned in a frame on the top of p. 135).

- ■ Definitions of string and language reversals and a related theorem ---
  - ♦ The *reversal* w^R of a string $w = a_1a_2, \ldots a_n$ is $w^R = a_na_{n-1}\ldots a_2a_1$.
  - ♦ The *reversal* L^R of a language L is the language consisting of the reversals of all its strings.
  - ♦ **Theorem 4.11** --- the reversal L^R of an RL L is also an RL.

- ■ Definitions of homomorphism and inverse homomorphism, examples, and related theorems ---
  - ♦ Definition of *homomorphism* ---

A *(string) homomorphism* is a function h which substitutes a particular string for each symbol. That is, $h(a) = x$, where a is a symbol and x is a string.

- Given $w = a_1a_2\ldots a_n$, we define $h(w) = h(a_1)h(a_2)\ldots h(a_n)$.
- Given a language, we define $h(L) = \{h(w) \mid w \in L\}$.

♦ **Example 4.13 ---**

Let function h be defined as $h(0)= ab$ and $h(1) = \varepsilon$, then h is a string homomorphism. For examples,

- $h(0011) = h(0)h(0)h(1)h(1) = abab\varepsilon\varepsilon = abab$.
- If RE $r = \mathbf{10}^*\mathbf{1}$, then $h(L(r)) = L((\mathbf{ab})^*)$.

♦ **Theorem 4.14** --- If L is an RL, then $h(L)$ is also an RL where h is a homomorphism.

♦ Definition of *inverse homomorphism* ---

Let h be a homomorphism from some alphabet Σ to strings in another alphabet T. Let L be an RL over T. Then $h^{-1}(L)$ is the set of strings w such that $h(w)$ is in L.

- $h^{-1}(L)$ is read "h inverse of L."

♦ **Example 4.15 ---**

Let $L = L((\mathbf{00} + \mathbf{1})^*)$. Let (string) homomorphism h be defined as $h(a) = 01$, $h(b) = 10$. It can be proved that $h^{-1}(L) = L((\mathbf{ba})^*)$

(For the proof, see the textbook).

♦ **Theorem 4.16** --- If h is a homomorphism from alphabet Σ to alphabet T, and L is an RL, then $h^{-1}(L)$ is also an RL.

4.3 Decision Properties of RL's

■ A *decision problem* about a certain property P of a computational model means a question about whether P can be determined to be true or not in reasonable time.

4.3.1 Resources Requirements for Conversions among Representations

■ Given an FA, assume:
- ♦ #symbols = constant;
- ♦ #states = n.

■ The following facts can be proven to be true (for proofs, see the textbook):
- ♦ conversion from an ε-NFA to a DFA --- requiring time of the order $O(n^3 2^n)$ in the worse cases;
- ♦ conversion from a DFA to an NFA --- requiring time of the order $O(n)$;
- ♦ conversion from an automaton (DFA) to an RE --- requiring time of the order $O(n^3 4^n)$;
- ♦ conversion from an RE to an automaton (ε-NFA) --- requiring linear time proportional to the size of the RE.

4.3.2 Testing Emptiness of RL's

■ **Emptiness problem of RL's** --- determining if a regular language generated by an

automaton or represented by a regular expression is *empty*.

- **Decidability** --- a problem is said to be *decidable* if there exists an algorithm to answer the problem in reasonable time.

- **Problem 1** --- testing if a regular language generated by an *automaton* is empty.
  - Equivalent to testing if there exists *no* path from the start state to an accepting state.
  - Requiring $O(n^2)$ time in the worse case.
  - Why? Time proportional to #arcs ⇒ each state has at most n arcs (to the n states) ⇒ at most n^2 arcs ⇒ at most $O(n^2)$ time

- **Problem 2** --- testing if a language represented by an *RE* is empty
  - Two methods for the testing ---
    - A direct method: easy; see p. 154 of the textbook.
    - A 2-step method described next.

  - A 2-step method for testing if a language generated by an RE is empty:
    - Step 1 --- convert the RE to an ε-NFA, requiring time $O(s)$ as said previously, where $s = |\text{RE}|$ (length of RE);
    - Step 2 --- test if the language of the ε-NFA is empty, requiring time $O(n^2)$ as said above.

    Therefore, the overall time requirement is $O(s) + O(n^2)$.

- **Conclusion** --- The problem of testing emptiness of RL's is *decidable* (i.e., there exists an algorithm executable in reasonable time to answer the problem).
  - Note: RL's may be accepted by various automata (DFA's, NFA's, ε-NFA's) or represented by RE's.

4.3.3 <u>Testing Membership in an RL</u>

- **Membership Problem of RL's** --- determining if a given string w is in a given RL L.

- **Problem 1** --- The RL L is represented by a *DFA*.
  - The algorithm to answer the problem requires $O(n)$ time, where $n = |w|$ (# symbols in the *string* instead of #states of the automaton).
  - Why? Just processing input symbols one by one to see if an accepting state is reached.

- **Problem 2** --- The RL L is represented by an NFA without ε-transitions.
  - The algorithm requires $O(ns^2)$ time, where
    - $n = |w|$ (# symbols in string w instead of #states);
    - s = #states.
  - Why? Just processing input symbols one by one to see if an accepting state is reached, and at each state there are at most s^2 choices of the next states.

- **Problem 3** --- The RL L is represented by an ε-NFA.
  - The algorithm has to compute the ε-closures at first before processing the symbols.
  - Computing ε-closures requires time $O(s^2) + O(s^2) = O(s^2)$ (see the textbook for the proof).
  - Processing the input string of symbols needs time $n \times O(s^2) = O(ns^2)$.
  - The overall required time is so $O(s^2) + O(ns^2) = O(ns^2)$.

■ **Problem 4** --- The RL L is represented by an RE of size s.
- ♦ The algorithm first transforms it to an ε-NFA with at most $2s$ states in time $O(s)$.
- ♦ Then do as for Problem 3.
- ♦ The overall required time is so $O(s) + O(ns^2) = O(ns^2)$.

■ **Conclusion** --- The problem of testing the membership of an RL is *decidable*.

4.4 Equivalence & Minimization of Automata

■ **What we want to show in this section:**
- ♦ Testing whether two descriptions of RL's define the same languages.
- ♦ Minimization of DFA's --- good for implementations of DFA's with less resources (like space, time, IC areas, …)

4.4.1 Testing Equivalence of States

■ Goal --- want to understand when two distinct states p and q can be replaced by a single state that behaves like both p and q.

■ Definition --- two states are said *equivalent* if for all strings w, $\hat{\delta}(p, w)$ is an accepting state if and only if $\hat{\delta}(q, w)$ is an accepting state.

■ Note:
- ♦ It is *not* necessary to enter the same accepting state for the above definition to be met.
- ♦ We only require that either *both states are accepting or both states are non-accepting*.

■ Definition --- non-equivalent states are said to be *distinguishable*.
- ♦ That is, state p is said to be distinguishable from q if there is at least a string w such that one of $\hat{\delta}(p, w)$ and $\hat{\delta}(q, w)$ is accepting, and the other is not accepting.

■ A systematic way to find distinguishable states --- use a *table-filling algorithm*.

■ **Table-filling algorithm** ---
- ♦ **Basis**.

  If p is an accepting state and q is not, then the pair $\{p, q\}$ is distinguishable.

- ♦ **Induction**.

  Let p and q be states such that for some input symbol a, the next states $r = \hat{\delta}(p, a)$ and $s = \hat{\delta}(q, a)$ are known to be distinguishable. Then, $\{p, q\}$ are distinguishable.

  That is, the precedents (前行者) p, q of a distinguishable pair r, s are also a distinguishable pair.

  - Why?

    $r = \hat{\delta}(p, a)$, $s = \hat{\delta}(q, a)$ are distinguishable

    ⇒ there exists a string w such that *only* one of $\hat{\delta}(r, w)$ and $\hat{\delta}(s, w)$ is accepting

    ⇒ but $\hat{\delta}(p, aw) = \hat{\delta}(r, w)$, $\hat{\delta}(q, aw) = \hat{\delta}(s, w)$

    ⇒ there exists a string aw such that *only* one of $\hat{\delta}(p, aw)$ and $\hat{\delta}(q, aw)$ is accepting.

$\Rightarrow p$ and q are distinguishable.

■ **Example 4.19** ---

Apply the above algorithm to the DFA shown in Fig. 4.1 (Fig. 4.18 in the textbook).

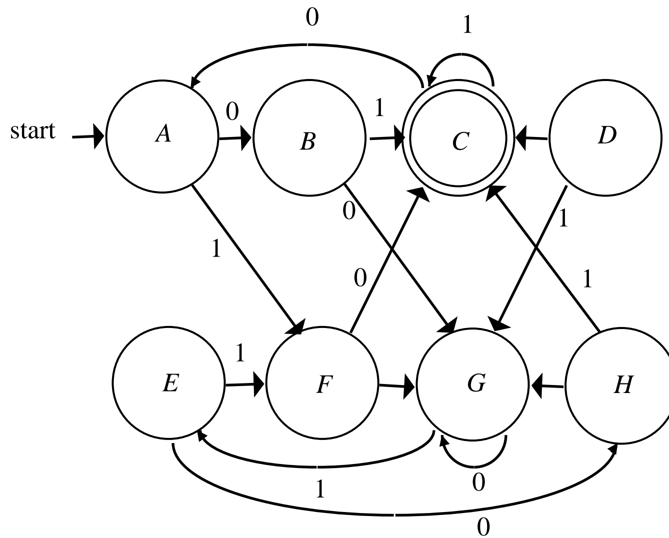


Fig. 4.1 DFA of Example 4.19.

♦ *Basis*:

- Since C is the only accepting state, we put an "×" into the pairs of $\{A, C\}$, $\{B, C\}$, $\{C, D\}$, $\{C, E\}$, $\{C, F\}$, $\{C, G\}$, $\{C, H\}$, with × meaning "distinguishable".
- The result is shown in Table 4.1.

Table 4.1 The result of the basis step of Example 4.19.

	A	B	C	D	E	F	G
B							
C	×	×					
D			×				
E			×				
F			×				
G			×				
H			×				

♦ Induction:

- Step 1: for the pair $\{C, H\}$, input 0 brings pair $\{E, F\}$ to pair $\{C, H\}$, so $\{E, F\}$ are distinguishable and the pair is marked. The result is shown in Table 4.2 with the pair marked in red.

- Step 2: find other pairs using the *existing* pairs, and the result is shown in Table 4.3 (blue pairs are found from the bold red pair as “triggers” and subscripts are inputs).

Table 4.2 The result of Step 1 of the induction process of Example 4.19.

	A	B	C	D	E	F	G
B							
C	×	×					
D			×				
E			×				
F			×		×		
G			×				
H			×				

Table 4.3 The result of Step 2 of the induction process of Example 4.19.

	A	B	C	D	E	F	G
B							
C	×	×					
D		$\times_{0}$	×				
E			×	×			
F		$\times_{0,1}$	×				
G			×	$\times_{0}$		$\times_{0}$	
H			×	$\times_{0}$		$\times_{0,1}$	

- Step 3: do this for all pairs *in order recursively* until no more pair can be marked. The final result is shown in Table 4.4.

Table 4.4 The final result of the induction process of Example 4.19.

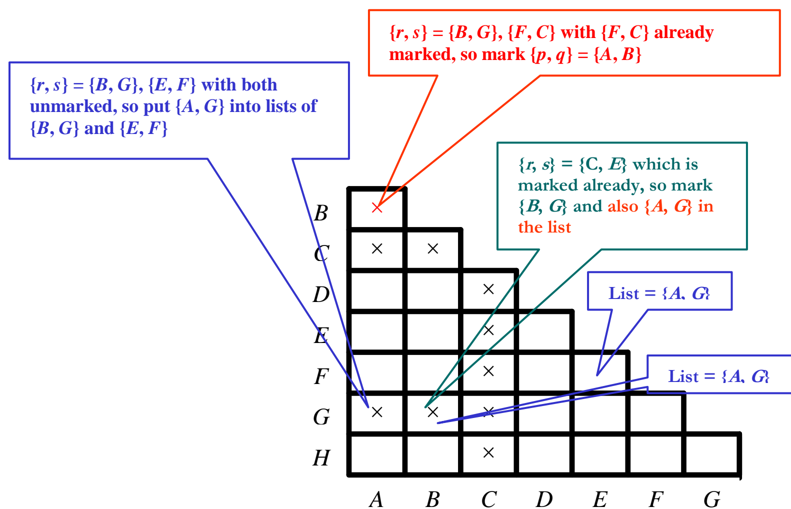
	A	B	C	D	E	F	G
B	×						
C	×	×					
D	×	×	×				
E		×	×	×			
F	×	×	×		×		
G	×	×	×	×	×	×	
H	×		×	x	×	×	×

- A note: the above method described in the textbook "wastes" some intermediate results. A better way is given next.

- **A better way** (see the 3rd paragraph, p. 160 in textbook) ---
  - ♦ After finding distinguishable pairs by final states and marking them by "×" in the table, perform the following steps:
    - Set up a ***list*** for each pair in the table, initially empty.
    - For each unmarked pair $\{p, q\}$, do:
      (1) For each symbol a, compute $r = \delta(p, a)$, $s = \delta(q, a)$.
      (2) *If* any pair $\{r, s\}$ is marked,
      then also mark the pair $\{p, q\}$ *as well as all the pairs in the list* of the pair $\{p, q\}$, and *also*, *recursively*, all the pairs in the lists of just-marked pairs;
      else put the pair $\{p, q\}$ into the list of each pair of $\{r, s\}$.
    - Repeat the last step until no more pair in the table can be marked.

  - ♦ An example of the intermediate results is shown in Table 4.5. The Final result is the same as shown in Table 4.4.

Table 4.5 An intermediate result of applying the better way of induction to Example 4.19.



  - ♦ Then, what?

- **Theorem 4.20** ---

  If two states are not distinguishable by the table-filling algorithm, then they are equivalent.

4.4.2 (Taught later)

4.4.3 Minimization of DFA's

- After indistinguishable states are found out as described previously, the following process is conducted:

- ♦ Group equivalent states into a block and regard each block as a new state in the minimized DFA.
- ♦ Take the block containing the old start state as the new start state.
- ♦ Take the new accepting states as those blocks which contain old accepting states.

- ■ **Example 4.25** (cont'd from Example 4.19) ---
  - ♦ The final result described by Table 4.4 says that (A, E), (B, H), (D, F) are equivalent states and can be put into 3 blocks as states of the new DFA.
  - ♦ The final new DFA is as shown in Fig.4.2.

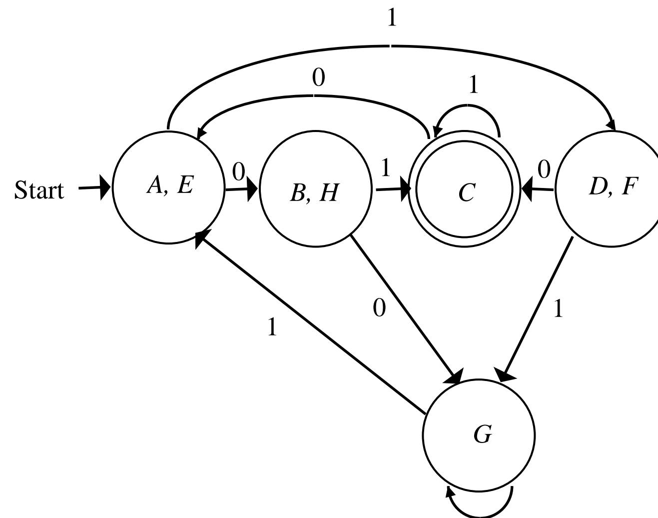


Fig. 4.2 The minimized DFA of Example 4.25.

4.4.2 Testing Equivalence of RL's

- ■ **Concept**: use the table filling algorithm
- ■ Given two DFA's A_L and A_M with start states q_L and q_M, respectively, for two RL's L and M, we can test if their languages are equivalent by:
  - ♦ imagine a third DFA A_3 whose states are union of those of A_L and A_M;
  - ♦ test if q_L and q_M are equivalent; if so, L and M are equivalent.

- ■ Why? Because they accept the same set of strings, i.e., the same set of languages.

- ■ See Example 4.21 for an example.

- ■ The table filling algorithm requires time of the order O(n^2) where n = #states.

- ■ **Conclusion**: the problem of testing the equivalence of two RL's is *decidable*.

4.4.4 Why the Minimized DFA's Can't be Beaten

- ■ The minimized DFA by the table filling algorithm is really the "minimal," having the *fewest* states in all DFA's which accept the same language, as guaranteed by the following theorem.

■ **Theorem 4.26** ---

If A is a DFA, and M is the DFA constructed from A by the table filling algorithm, then M has as few states as any DFA equivalent to A.