Properties of Regular Languages

Yo-Sub Han
CS, Yonsei University

Overview of Unit

1. Closure properties
2. Decision properties
3. Minimization of DFAs
4. Pumping Lemma

Closure Properties of Regular Languages

Let L_1 and L_2 be regular languages. Regular languages are closed under the following operations:

- Union: $L_1 \cup L_2$
- Intersection: $L_1 \cap L_2$
- Complement: $\overline{L_1}$
- Difference: $L_1 \setminus L_2$
- Reversal: $L_1^R = \{w^R \mid w \in L_1\}$
- Kleene closure: L_1^*
- Catenation: $L_1 L_2$
- Homomorphism: $h(L_1) = \{h(w) \mid w \in L_1$ and h is a homomorphism$\}$
- Inverse homomorphism:
  $h^{-1}(L_1) = \{w \in \Delta^* \mid h(w) \in L_1, h : \Sigma \to \Delta$ is a homomorphism$\}$

Closure Properties of Regular Languages

Claim: Let L_1 and L_2 be regular languages. $L_1 \cup L_2$ is regular.

Proof: Let E be a regular expression for L_1 and F be a regular expression for L_2. By definition, $L(E + F) = L_1 \cup L_2$.

Claim: Let L be a regular language. $\overline{L} = \Sigma^* \setminus L$ is regular.

Proof: Let $A = (Q, \Sigma, \delta, s, F)$ be a DFA for L. We flip the accepting status of all states (final state $\leftrightarrow$ nonfinal state). Namely,

$$A' = (Q, \Sigma, \delta, s, Q \setminus F).$$

Then, $L(A') = \overline{(L)}$.

Closure Properties of Regular Languages

Claim: Let L_1 and L_2 be regular languages. $L_1 \cap L_2$ is regular.

Proof: $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ by DeMorgan's law. Since regular languages are closed under union and complement, $L_1 \cap L_2$ is regular.

We also look at another "direct" proof based on the Cartesian product.

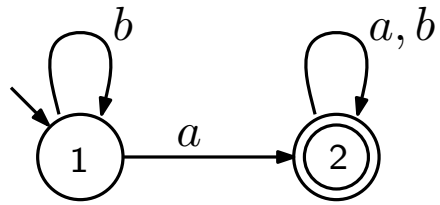
Let $A_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$ be DFAs for L_1 and L_2, respectively.
We construct a new FA that simulates A_1 and A_2 in parallel, and accepts a string if and only if both A_1 and A_2 accept it.
Let $A = (Q, \Sigma, \delta, s, F)$, where

1. $Q = Q_1 \times Q_2$; namely, Q is a set of state pairs
2. $\delta(q, \sigma) = (\delta(q_1, \sigma), \delta(q_2, \sigma))$, where $q = (q_1, q_2)$ and $\sigma \in \Sigma$
3. $s = (s_1, s_2)$
4. $F = F_1 \times F_2$

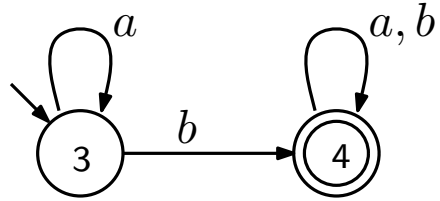
Then, $L(A) = L_1 \cap L_2$.

Cartesian Product Example

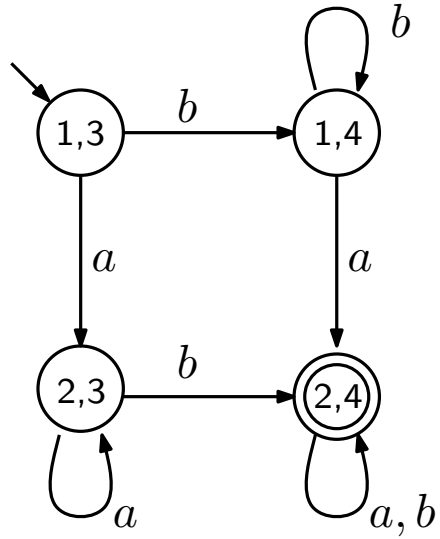
$L_1 = L(b^*a(a+b)^*)$



$L_2 = L(a^*b(a+b)^*)$



$L_1 \cap L_2$



Closure Properties of Regular Languages

Claim: Let L_1 and L_2 be regular languages. $L_1 \setminus L_2$ is regular.

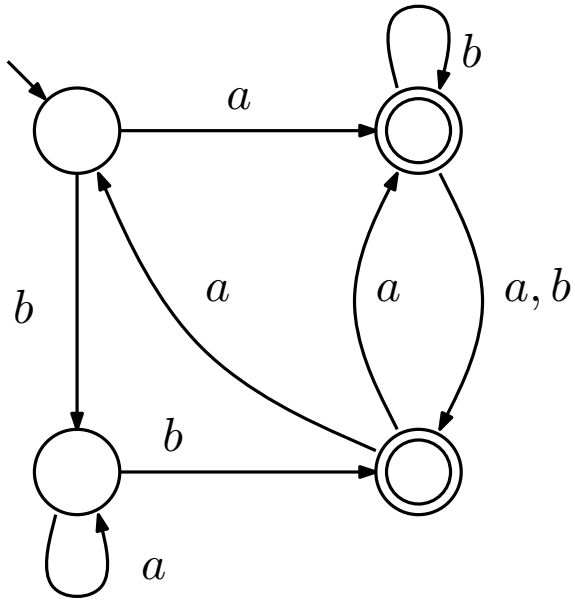
Proof: $L_1 \setminus L_2 = L_1 \cap \overline{L_2}$.

Claim: Let L be a regular language. L^R is regular.

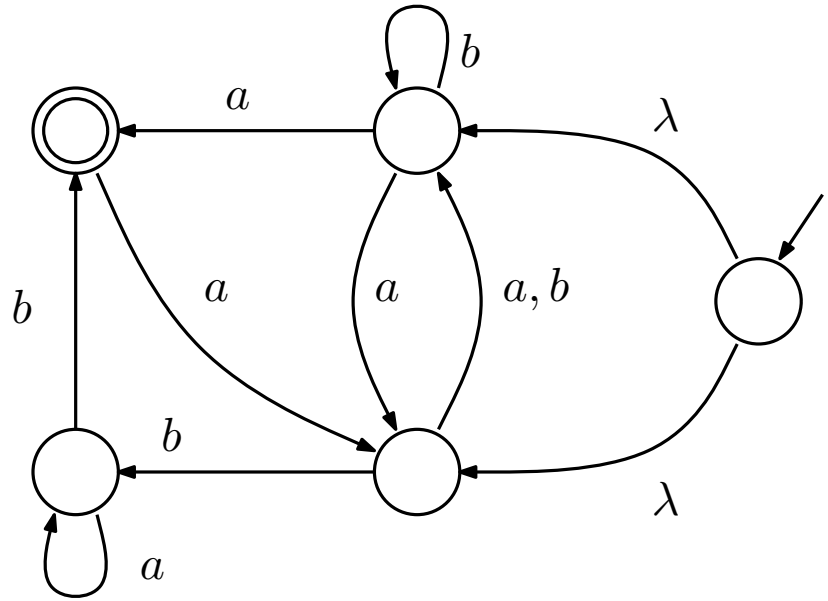
Proof: Let $A = (Q, \Sigma, \delta, s, F)$ be a DFA for L. From A, we construct $A' = (Q, \Sigma, \delta^R, s', s)$, where

1. $(q, \sigma, p) \in \delta^R$ iff $(p, \sigma, q) \in \delta$ for all states $p, q \in Q$
2. $(s', \lambda, f) \in \delta^R$ for all $f \in F$
3. s is the new single final state in A'

Reversal FA Construction Example



$L(A)$



$L(A)^R$

Closure Properties of Regular Languages

Let Σ and Γ be alphabets. We define a homomorphism to be a function

$$h : \Sigma \to \Gamma^*.$$

Note that a homomorphism is "substituting" a symbol in Σ for a string over Γ such that

$$h(xy) = h(x)h(y).$$

Let $w = \sigma_1\sigma_2 \cdots \sigma_n \in \Sigma^*$. Then,

1. $h(w) = h(\sigma_1)h(\sigma_2) \cdots h(\sigma_n)$ and
2. $h(L) = \{h(w) \mid w \in L\}$

Example: Let $h : \{0, 1\} \to \{a, b\}^*$ be defined by $h(0) = ab$ and $h(1) = \lambda$. Then,

1. $h(0011) = abab$
2. $h(L(10^*1)) = L((ab)^*)$

Closure Properties of Regular Languages

Claim: Let L be a regular language and h be a homomorphism. $h(L)$ is regular.

Proof: Let $L = L(E)$ for a regular expression E. We prove that $L(h(E)) = h(L)$.

Basis:

1. If E is λ or $\emptyset$, then $h(E) = E$ since h does not affect the string λ and the language $\emptyset$.
   This implies that $L(h(E)) = L(E) = h(L(E))$
2. If $E = \sigma$, then $L(E) = \{\sigma\}$.
   This implies that $L(h(E)) = L(h(\sigma)) = \{h(\sigma)\} = h(L(E))$

Induction:

1. $L = L(E + F)$. Then, $L(h(E + F)) = L(h(E) + h(F)) = L(h(E)) \cup L(h(F)) = h(L(E)) \cup h(L(F)) = h(L(E) \cup L(F)) = h(L(E + F))$
2. $L = L(EF)$. Then, $L(h(EF)) = L(h(E)h(F)) = L(h(E))L(h(F)) = h(L(E))h(L(F)) = h(L(E)L(F)) = h(L(EF))$
3. $L = L(E^*)$. Then, $L(h(E^*)) = L(h(E)^*) = L(h(E))^* = h(L(E))^* = h(L(E^*))$

Decision Properties of Regular Languages

We consider the following problems:

1. Converting among representations for regular languages
2. Is $L = \emptyset$?
3. Is $w \in L$?
4. Do two descriptions define the same language?

Decision Properties

From NFAs to DFAs: Given a λ-NFA with n states,

1. We can compute an NFA without λ-transitions in $O(n^3)$ steps
2. We can convert an n-state NFA into a DFA by the subset construction. The construction requires 2^n states and, thus, it takes $O(2^n)$ steps

From DFAs to NFAs:

Decision Properties

From regular expressions to FAs: Given a regular expression whose size is n,

1. We can build an expression tree in n steps
2. We can construct the Thompson automaton in n steps
3. We can eliminate λ-transitions in $O(n^3)$ steps
4. If we want a DFA, we might need an exponential number of states; $L((a+b)^*a(a+b)^k)$

Additional remarks: Using the Thompson construction and the λ elimination technique we can obtain an NFA without λ-transitions in $O(n^3)$ time. (Step 2 and Step 3)
However, there is another old yet very simple construction called the position construction by Glushkov, and McNaughton and Yamada in 1960 that gives an NFA with n states and n transitions (on average) in linear time.

Emptiness Test

We now want to examine whether or not a given regular language L is empty; Namely, $L = \emptyset$?

1. L is given by an FA $A = (Q, \Sigma, \delta, s, F)$:
   $L(A) \neq \emptyset$ if and only if there is a path from s to $f \in F$. We can check this in $O(n^2)$ steps.
2. L is given by a regular expression E:
   (a) We can construct an FA for E and use the previous approach
   (b) We can directly inspect E
      i. $E = \emptyset$. $L(E)$ is empty
      ii. $E = \lambda$. $L(E)$ is not empty
      iii. $E = \sigma$. $L(E)$ is not empty
      iv. $E = F + G$. $L(E)$ is empty iff both $L(F)$ and $L(G)$ are empty
      v. $E = FG$. $L(E)$ is empty iff either $L(F)$ or $L(G)$ is empty
      vi. $E = F^*$. $L(E)$ is never empty since $\lambda \in L(E)$

Membership Test

We now want to determine whether or not $w \in L$ for an input string w such that $|w| = m$:

1. For a DFA with n states, we simulate A on w and it takes $O(m)$ steps
2. For an NFA with n states, it takes $O(mn^2)$ steps

Equivalence and Minimization of DFAs

Given a DFA $A = (Q, \Sigma, \delta, s, F)$ and two states $p, q \in Q$,

1. Given a state p and a string $w = w_1 w_2 \cdots w_n$, we define

$$\hat{\delta}(p, w) = q \text{ iff } (p, w) \vdash_A^n (q, \lambda)$$

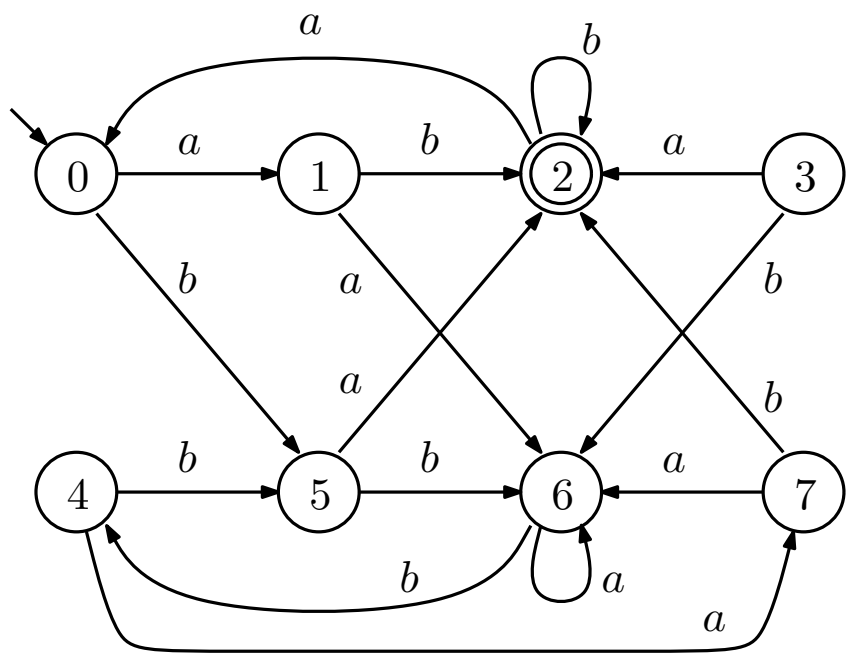
2. We say that p and q are equivalent ($\equiv$)

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

3. Otherwise, we say that p and q are distinguishable (or inequivalent)
   In other words, p and q are distinguishable iff

$$\exists w, \hat{\delta}(p, w) \in F \text{ and } \hat{\delta}(q, w) \neq F, \text{ or vice versa}$$

Equivalent States Example



$F = \{2\}$

$\hat{\delta}(2, \lambda) \in F$ but $\hat{\delta}(6, \lambda) \notin F \Longrightarrow 2 \not\equiv 6$.

$\hat{\delta}(2, \lambda) \in F$ but $\hat{\delta}(6, \lambda) \notin F \Longrightarrow 2 \not\equiv 6$.

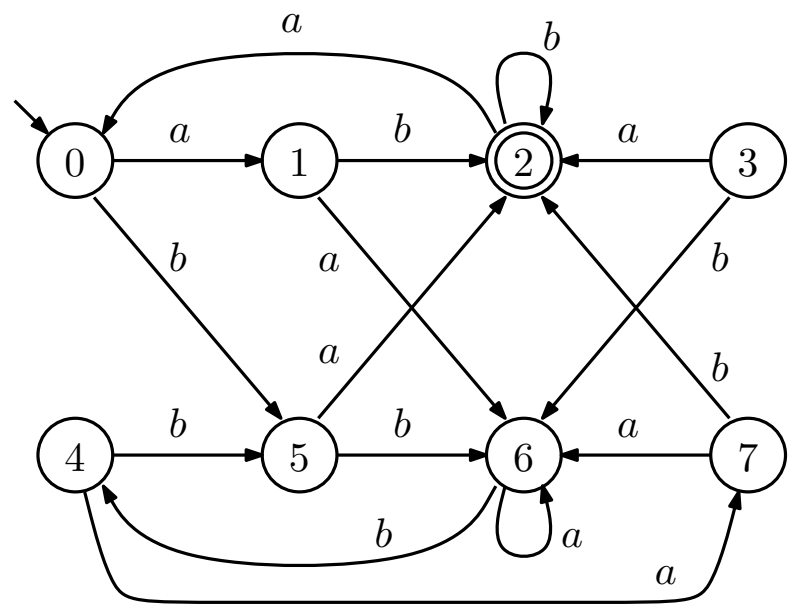
$\hat{\delta}(0, \lambda) \notin F$ and $\hat{\delta}(4, \lambda) \notin F$, and $\hat{\delta}(0, b) = \hat{\delta}(4, b) = 5 \Longrightarrow \hat{\delta}(0, bw) = \hat{\delta}(4, bw) = \hat{\delta}(5, w)$.
$\hat{\delta}(0, aa) = \hat{\delta}(4, aa) = 6$ and
$\hat{\delta}(0, ab) = \hat{\delta}(4, ab) = 2$.

$0 \equiv 4$.

Table Filling Algorithm—Distinguishable States

We identify distinguishable states using the following inductive table filling algorithm:

Basis: If $p \in F$ and $q \notin F$, then $p \not\equiv q$.
Induction: if $\exists \sigma \in \Sigma : \delta(p, \sigma) \not\equiv \delta(q, \sigma)$, then $p \not\equiv q$.



Applying the table filling algorithm example

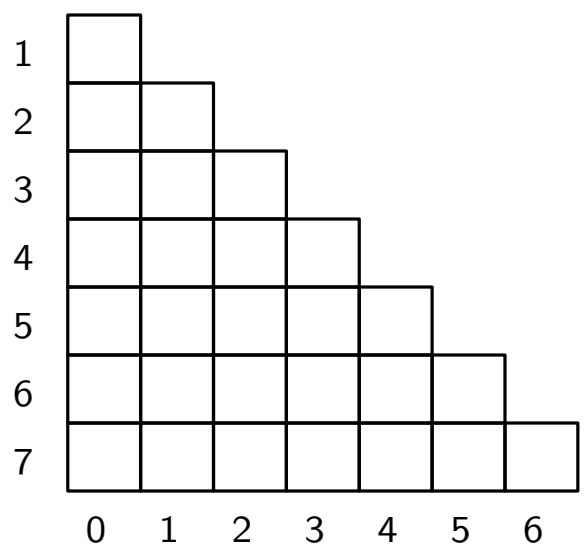


Table Filling Algorithm—Distinguishable States

Claim: If p and q are not distinguished by the TF-algorithm, then $p \equiv q$.

Proof: Suppose to the contrary that there is a pair of states (p, q) such that

1. there is a string w such that $\hat{\delta}(p, w) \in F, \hat{\delta}(q, w) \notin F$ or vice versa
2. The TF-algorithm fails to distinguish p and q

We call such a pair of states a bad pair. Let $w = \sigma_1\sigma_2 \cdots \sigma_n$ be the shortest string that identifies a bad pair (p, q). Note that $w \neq \lambda$ since if λ distinguishes a pair of states, then that pair is marked by the basis part of the algorithm. Thus $n \geq 1$.
Consider the states $p' = \delta(p, \sigma_1)$ and $q' = \delta(q, \sigma_1)$. States p' and q' are distinguished by the string $\sigma_2\sigma_3 \cdots \sigma_n$, since this string takes p' and q' to $\hat{\delta}(p, w)$ and $\hat{\delta}(q, w)$, respectively.
Now (p', q') cannot be a bad pair because we assume that (p, q) are the bad pair with the shortest string w. Therefore, the algorithm should have discovered that p' and q' are distinguishable.
This implies that the algorithm must distinguish p from q in the inductive step—a contradiction.

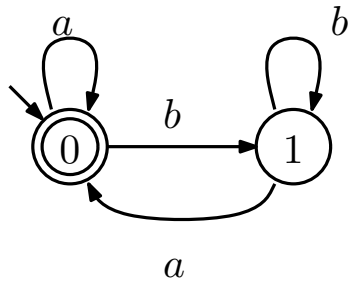
Equivalence Test for Regular Languages

Let L_1 and L_2 be two regular languages. We now determine whether or not $L_1 = L_2$ as follows:

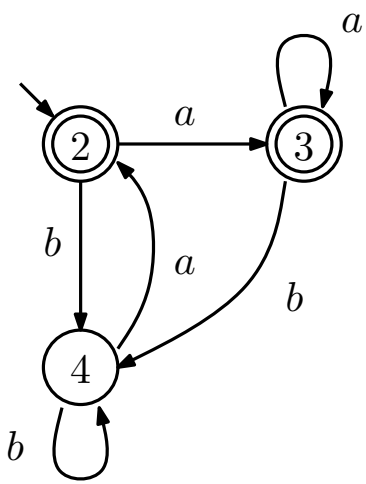
1. Obtain DFAs A_1, A_2 for L_1, L_2
2. Make a new DFA that is the union A_1 and A_2 (we do not concern about two start states)
3. If TF-algorithm says that the two start states are distinguishable, then $L_1 \neq L_2$. Otherwise, $L_1 = L_2$

Equivalence Test for Regular Languages Example

L_1



L_2



We can *obviously* see that both DFAs accept $L(\lambda + (a+b)^*a))$. Here is the TF-algorithm result:

1	X			
2		X		
3		X		
4	X		X	X
	0	1	2	3

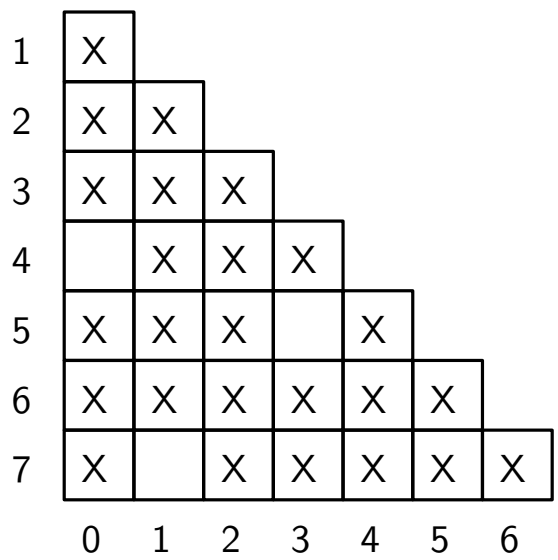
Since $0 \equiv 2$, $L_1 = L_2$.

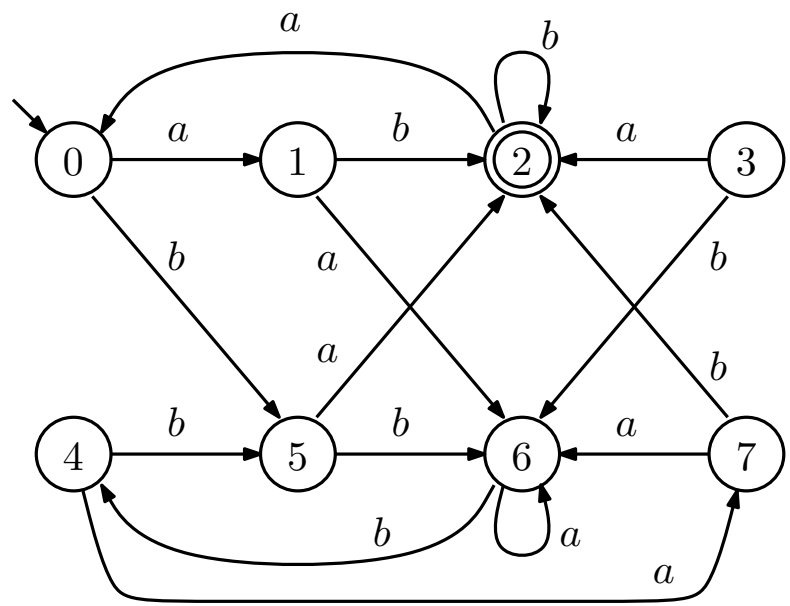
Equivalence Test for Regular Languages

The running time to fill out the table, and thus, to decide whether or not two states are equivalent is polynomial in the number of states.
Assume that there are n states. Then, there are $n(n-1)/2$ pairs of states. In one step, we consider all pairs of states to see if one of their successor pairs has been found distinguishable (marked as "X"), so a step takes no more than $O(n^2)$ time. Moreover, if on some step, no additional X's are placed in the table, then the algorithm ends. Therefore, there can be no more than $O(n^2)$ steps, and $O(n^2) \times O(n^2) = O(n^4)$ is surely an upper bound on the running time of the TF algorithm.
(The algorithm can be improved into an $O(n^2)$ algorithm using addition lists. See the text page 160 for details.)

DFA Minimization

We know that we identify equivalent states using the TF-algorithm. Now we show how to minimize a DFA by merging equivalent states. First, we remove all unreachable states from the start state and, then, we replace each state p by its equivalent class $[p]$.





Equivalent Class $= \{\{0,4\},\{1,7\},\{2\},\{3,5\},\{6\}\}$

If $[p]$ is an equivalence class, then the relation $\equiv$ has to be an equivalence relation (reflexive, symmetric and transitive).

DFA Minimization

Claim: If $p \equiv q$ and $q \equiv r$, then $p \equiv r$.

Proof: Suppose to the contrary that $p \not\equiv r$. This implies that there is a string w such that $\hat{\delta}(p, w) \in F$ but $\hat{\delta}(r, w) \notin F$ (or vice versa). Note that $\hat{\delta}(q, w)$ is either accepting or not since $p \equiv q$ and $q \equiv r$.

1. $\hat{\delta}(q, w) \in F$. This implies that $q \not\equiv r$—a contradiction
2. $\hat{\delta}(q, w) \notin F$. This implies that $q \not\equiv p$—a contradiction

The other case is proved symmetrically.
Therefore, $p \equiv r$.

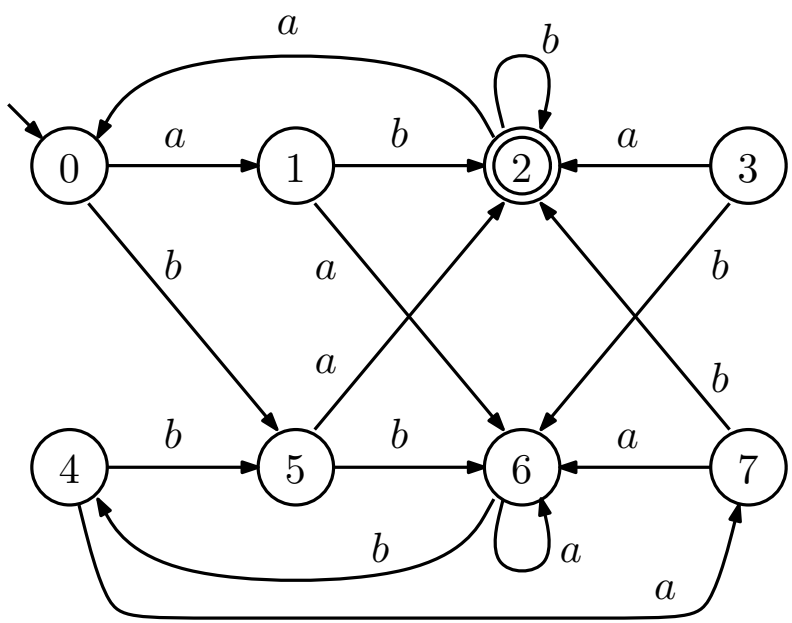
Claim: The relation $\equiv$ is an equivalence relation.

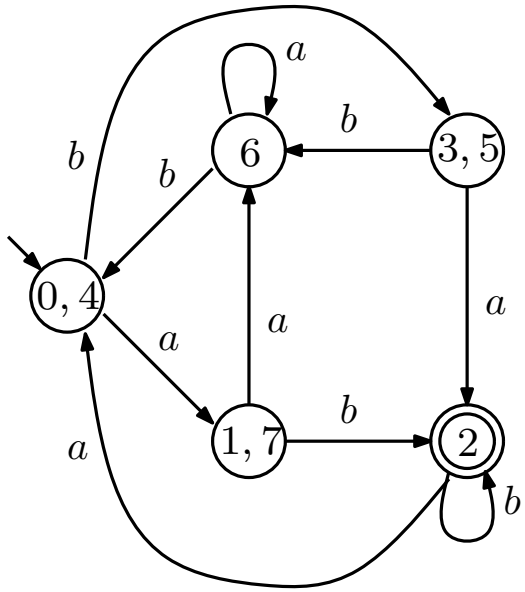
Proof: The other two cases are left for an exercise.

DFA Minimization

Given a DFA $A = (Q, \Sigma, \delta, s, F)$, we construct a DFA $B = (Q', \Sigma, \delta', s', F')$, where

1. $Q' = \{[p] \mid p \text{ is in } Q \text{ and } [p] \text{ is the equivalence class of } p \text{ in } \equiv\}$
2. $s' = [s]$
3. $F' = \{[f] \mid f \in F\}$
4. $\delta'([p], \sigma) = [\delta(p, \sigma)]$ for all $[p] \in Q'$ and $\sigma \in \Sigma$





Equivalent Class $= \{\{0, 4\}, \{1, 7\}, \{2\}, \{3, 5\}, \{6\}\}$

The Minimal DFA

Let $B = (Q_B, \Sigma, \delta_B, s_B, F_B)$ be the minimized DFA obtained from A by applying the TF-algorithm and the minimization construction. We know that $L(A) = L(B)$. Now the question is whether or not there is a DFA $C = (Q_C, \Sigma, \delta_C, s_C, F_C)$ such that $L(C) = L(B)$ and has fewer states than B?

Imagine the union of B and C. Now we run the TF-algorithm on the union of B and C. Since $L(B) = L(C)$, $s_B \equiv s_C$. Moreover, if $(p \in Q_B, q \in Q_C)$ are equivalent, then their successors on any symbol are also equivalent. **(WHY?)**

Claim: For each state $p \in B$, there is at least one state $q \in C$ such that $p \equiv q$.

Proof: Imagine the union of B and C. Now we run the TF-algorithm on the union of B and C. Since $L(B) = L(C)$, $s_B \equiv s_C$. Moreover, $\delta(s_B, \sigma) \equiv \delta(s_C, \sigma)$ for any σ.
Since p is accessible from q_B, there is a path from q_B to p with some string $\sigma_1\sigma_2\cdots\sigma_k$. Then, by simulating $\sigma_1\sigma_2\cdots\sigma_k$ on C, we can find a state and this state is q such that $p \equiv q$.

The Minimal DFA

Let $B = (Q_B, \Sigma, \delta_B, s_B, F_B)$ be the minimized DFA obtained from A by applying the TF-algorithm and the minimization construction. We know that $L(A) = L(B)$. Now the question is whether or not there is a DFA $C = (Q_C, \Sigma, \delta_C, s_C, F_C)$ such that $L(C) = L(B)$ and has fewer states than B?

NO!. Assume that there is C that has less states than B. This implies that there are two states p and r of B such that $p \equiv q \equiv r$ for some state q of C. But then, $p \equiv r$ since the equivalence relation is transitive—a contradiction since B is constructed by the TF-algorithm.
Therefore, "*the minimized DFA can't be beaten.*"

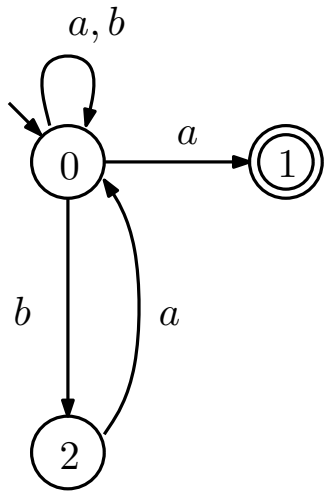
Claim: For a regular language, there is a unique minimal DFA (minimal number of states) for it.

Implication: There is a unique representation for a regular language. Note that there can be many different regular expressions, NFAs and even DFAs for the same language.

NFA Minimization

We cannot apply the TF-algorithm to NFAs.

For instance, to minimize



We simply remove state 2. But $0 \not\equiv 2$.

Additional remarks: The NFA minimization problem is known to be PSPACE-complete (beyond NP-complete).

Proving Regularity

A set of regular languages preserves certain regularity.

Here is a language that does not seem to be regular. Let $\Sigma = \{0, 1, \ldots, 9\}$ and L be the set of decimal representations of the natural numbers† that are divisible by 2 or 3.
e.g., $0, 2, 3, 843290, 578421, \ldots$.

- The set N of decimal representations of the natural numbers is

$$\{0\} \cup ((\Sigma \setminus \{0\}) \cdot \Sigma^*).$$

  Clearly, N is regular

- The set E of decimal representations of the even natural numbers is

$$N \cap (\Sigma^* \cdot \{0, 2, 4, 6, 8\}).$$

  Clearly, E is also regular

†: Here natural numbers are $0, 1, 2, \ldots$.

Proving Regularity

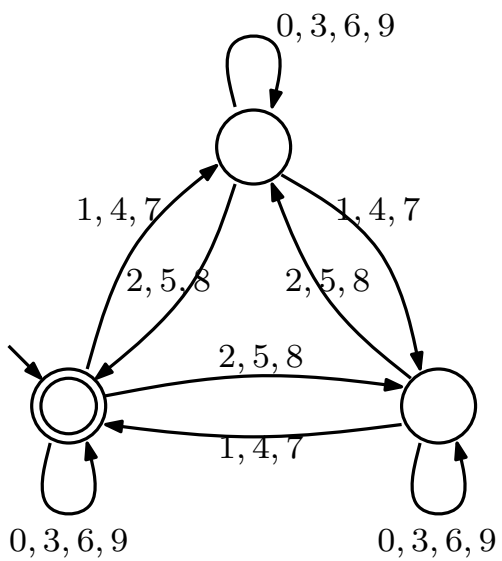
Here is a language that does not seem to be regular. Let $\Sigma = \{0, 1, \ldots, 9\}$ and L be the set of decimal representations of the natural numbers that are divisible by 2 or 3.

- The set T of decimal representations of the natural numbers that are multiples of 3 is

$$N \cap L(A),$$

  where $L(A)$ is the set of all strings accepted by the following DFA A. Since N and $L(A)$ are regular and the family of regular languages is closed under intersection, T is regular

- Hence, $L = E \cup T$ is regular



Nonregular Languages

The most famous nonregular language L is

$$L = \{a^i b^i \mid i \geq 0\}.$$

Why is it nonregular?

1. Intuitively, L is not regular because any automaton that can recongnize L must remember how many a's it has been seen so far (an unlimited number of possibilities)

2. But, this intuition is not a proof!

3. Most important, our intuition can sometimes lead us astray. Consider the two following languages:

   (a) $L_1 = \{w \mid w$ has the same number of a's and b's$\}$

   (b) $L_2 = \{w \mid w$ has the same number of the substrings ab and $ba\}$

   Are they regular?

Aside: The Pigeonhole Principle

The pigeonhole principle is: *Assume that we are given $n \geq 1$ pigeonholes and $m > n$ letters; however we assign the letters to pigeonholes, we can guarantee that some pigeonhole contains at least two letters.*

This result is very simple yet useful as we shall see. We prove the pigeonhole principle using induction on the number of pigeonholes.

Basis: $n = 1$ and $m \geq 2$. Thus, the only pigeonhole contains at least two letters.
Inductive hypothesis: Assume the claim holds for some $n \geq 1$.
Induction: We are given $n + 1$ pigeonholes and $m > n + 1$ letters. There are two cases:

1. One pigeonhole has at most one letter in it: Remove the chosen pigeonhole and its letter if it has one. Now observe that we have n pigeonholes and either m or $m-1$ letters. But, $m > n+1$ implies that $m > m - 1 > n$, so, by IH, the claim holds

2. No pigeonhole has fewer than two letters in it: This case is immediate

Proving nonregularity Example

We first prove, directly and by contradiction, that $L = \{a^i b^i \mid i \geq 0\}$ is not regular.

1. Assume that L is regular. This implies that there is a DFA $A = (Q, \Sigma, \delta, s, F)$ such that $L(A) = L$. We now derive a contradiction.

2. Let $n = |Q|$. There are infinitely many strings $w \in L$ such that $|w| = 2m$ and $m \geq n$. Consider one such string $w = a^n b^n$.

3. Now, since $w \in L(A)$, there is an accepting computation for w in A. We will write the computation as the sequence of transitions that are used in the computation:

   $$(p_0 = s, \sigma_1, p_1), (p_1, \sigma_2, p_2), \ldots, (p_{2n-1}, \sigma_{2n}, p_{2n}),$$

   where $w = \sigma_1 \sigma_2 \cdots \sigma_{2n}$ and $p_{2n} \in F$.

4. The crucial observation is that there are $2n + 1$ appearances of states in the computation; namely, $p_0, p_1, \ldots, p_{2n}$. But, there are only n different states; therefore, there must be multiple appearances of some states (Pigeonhole Principle).
   (We can think of states $\equiv$ pigeonholes, and appearances of states $\equiv$ letters)

Proving nonregularity Example

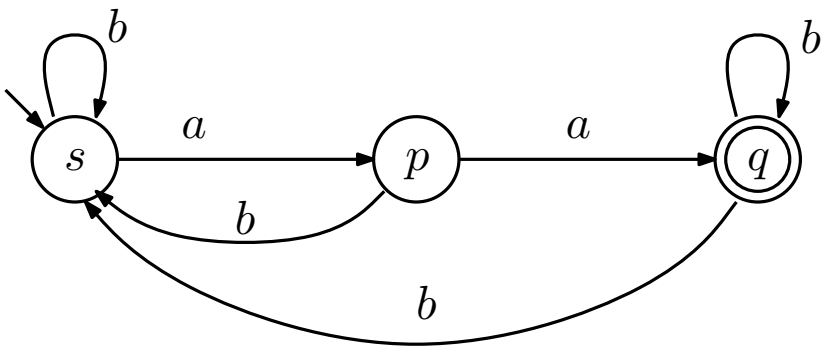
We first prove, directly and by contradiction, that $L = \{a^i b^i \mid i \geq 0\}$ is not regular.

5. Moreover, there must be at least two appearances of at least one state in the first n steps of the computation for the same reason. We conclude that there are p_i and p_j such that $0 \leq i < j \leq n$ and $p_i = p_j$.
6. Consider the portion of the computation from p_i to p_j. It consumes the substring $\sigma_{i+1} \cdots \sigma_j$
7. What do we know about this string? First, it is nonempty. Since $i < j$, it consists of at least one character. Second, it is a string of *only* a's; thus, $\sigma_{i+1} \cdots \sigma_j = a^{j-i}$
8. Finally, since this subcomputation begins and ends in the same actural state, we can omit it from the computation and *still have a valid accepting computation for a smaller string*
9. But, this omission implies that $a^{n+i-j}b^n$ is accepted—a contradiction (Note that since $i \neq j, n + i - j < n$)

Regular Pumping Lemma

This is really a theorem!

1. It is a property of all (infinite) regular languages
2. All strings in a regular language can be pumped if they are at least as long as a given positive integer, the pumping constant.
3. Infinite regular languages must have repetitive substructures that arise from a Kleene star in a regular expression or a cycle in a state diagram



Pumping Lemma

Let L be a regular language. Then, there is a pumping constant $n \geq 1$ such that if w is any string in L of length at least n, then w can be written as the catenation of three substrings $w = xyz$ (we often call this *factoring*) that satisfy the following three conditions:

1. $|y| > 0$
2. $|xy| \leq n$
3. For all $i \geq 0$, $xy^iz \in L$

Note that $|y| > 0$ implies that y is a not the empty string. Also note that when we proved the first nonregularity result, we factored $w = a^ib^i$ into three substrings.

Proof of Pumping Lemma

1. Since L is regular, there is a DFA A for L
2. Let n be the number of states in A
3. Let w be *any string* in L of length $m \geq n$, where $w = \sigma_1\sigma_2\cdots\sigma_m$, for $\sigma_i \in \Sigma$.
   Consider the accepting computation of w in A:
   $(p_0, \sigma_1, p_1), (p_1, \sigma_2, p_2), \ldots, (p_{n-1}, \sigma_n, p_n), \ldots$
4. Since A has n different states, by the Pigeonhole principle, there are i and j, $0 \leq i < j \leq n$ such that $p_i = p_j$
5. Hence, the substring $\sigma_{i+1}\sigma_{i+2}\cdots\sigma_j$ drives A from p_i back to $p_j = p_i$
6. Let $x = \sigma_1\cdots\sigma_i, y = \sigma_{i+1}\cdots\sigma_j$ and $z = \sigma_{j+1}\cdots\sigma_m$. Since $i < j, |y| > 0$ and, since $j \leq n, |xy| \leq n$
7. Now y can either be removed from w or be repeated many times and A still accepts the resulting strings. Hence, A accepts xy^iz for all $i \geq 0$

Proof of Pumping Lemma

Notes:

1. If all strings in L have length less than n, then the theorem is vacuously true. L is finite
2. Even if we remove the second condition $|xy| \leq n$, the claim still holds. This condition guarantees only that the first state repetition appears within the first n characters of the string

Using Pumping Lemma

Main idea: Proof by contradiction

1. To prove that a language L is not regular, we assume that L is regular
2. By the Pumping Lemma, we know that there is a pumping constant $n \geq 1$ such that *all strings* in L of length at least n can be pumped
3. Attempt to identify *one string* w in L that has length at least n but cannot be pumped
   (a) Choose *one string* $w \in L$ with $|w| \geq n$
   (b) Consider *all valid* ways of factoring w into xyz
   (c) For each valid factoring, demonstrate *one value* i such that $xy^iz \notin L$
4. The existence of such a w contradicts the assumed regularity of L
5. Hence, L is not regular

Using PL Exmaple I

Claim: $L = \{a^i b^i \mid i \geq 0\}$ is not regular.

Proof:

1. Suppose L is regular. Then, the Pumping Lemma applies
2. Let n be the pumping constant
3. Choose $w = a^n b^n$ (Note that $w \in L$ and $|w| \geq n$)
4. By PL, w can be factored into xyz such that $|xy| \leq n, |y| > 0$ and, for all $i \geq 0$, $xy^i z \in L$
5. Since $|xy| \leq n$ and $|y| > 0$, the string y consists only of a's and has at least one a. Let $y = a^k$, for some $k > 0$
6. Consider $i = 0$, then $xy^0 z = xz = a^{n-k} b^n \notin L$. We have obtained a contradiction of the assumed regularity of L
7. Hence, L is not regular

The Pumping Game

We can see the pumping lemma proof as a game between you and an Adversary (opponent). You want to show that L is not regular and the adversary wants to thwart you.

1. The adversary picks n
2. Given n, you pick a string w in L of length equal or greater than n
3. The adversary chooses the factoring of $w = xyz$, subject to $|xy| \leq n, |y| \geq 1$. You have to assume that the opponent makes the hardest choice for us to win the game
4. You try to pick i such that the pumped string $xy^iz \notin L$. If you can do so, you win the game. Otherwise, the adversary wins

Using PL Exmaple II

Claim: $L = \{a^i \mid i \text{ is a prime number}\}$ is not regular.

Proof:

1. Assume L is regular. Then, the Pumping Lemma applies
2. Let n be the pumping constant
3. Choose $w = a^P \in L$, where $P \geq n$ is a prime
4. By PL, w can be factored into xyz such that $|xy| \leq n, |y| > 0$ and, for all $i \geq 0$, $xy^iz \in L$
5. Since $|xy| \leq n$ and $|y| > 0$, $x = a^p, y = a^q, z = a^r$, where $p, q \geq 0$ and $p + q \leq n$
6. Consider $xy^{p+2q+r+2}z$. Then, the total number of a's is $p+q(p+2q+r+2)+r = (q+1)(p+2q+r)$. This gives a contradiction since the number is even and greater than 2. Hence, a^P is not in L—a contradiction
7. Hence, L is not regular

Proving Nonregularity

We can also prove that a language is not regular by using closure properties of regular languages to arrive at a contradiction.

Claim: $L = \{w \in \{a, b\}^* \mid w$ has the same numbers of a's and b's$\}$ is not regular.

Proof: Observe that $L \cap L(a^*b^*) = \{a^i b^i \mid i \geq 0\}$. If L were regular, then $L \cap L(a^*b^*)$ would also be regular because of closure under intersection. But, we have already proved that $\{a^i b^i \mid i \geq 0\}$ is not regular.

Example: Show that

$$L = \{a^n b^k c^{n+k} \mid n \geq 0, k \geq 0\}$$

is not regular.

Hint: You can prove this using the pumping lemma but you may want to use the closure property of regular languages under a "homomorphism".

Additional Remarks

1. Is it possible to use the Pumping Lemma to prove that a language is regular?
2. If a language is not regular, is it always possible to prove that it is not regular using the Pumping Lemma?

1. WARNING: There are nonregular languages that satisfy the pumping lemma. For example, given $\Sigma = \{a, b, c\}$
$$L = (aa^*c)^n(bb^*c)^n + \Sigma^*cc\Sigma^*$$
is not regular but it satisfies the pumping lemma conditions
2. Thus, for a language L to be regular, satisfying the pumping lemma conditions is necessary but not sufficient
3. If L is finite, then it is always regular

Summary of Unit

- Closure properties
- Decision properties
- Minimization of DFAs
- Pumping Lemma