Languages and Computation (COMP2049/AE2LAC)

Deterministic Finite Automata

Dr. Tianxiang Cui

tianxiang.cui@nottingham.edu.cn

Finite Automata

- A finite automaton is a simple type of computer
  - Its output is limited to "**yes**" or "**no**"
  - It has very primitive memory capabilities
- Any computer that answers yes or no acts as a **language acceptor**
- For now, consider that:
  - The input comes in the form of a string of individual input symbols
  - The computer gives an answer for the current string
    - i.e. the string of symbols that have been read so far

Finite Automata

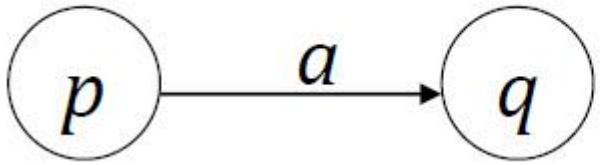
- In automata theory, the state of computer program is a technical term for all the stored information, at a given instant in time
  - The output of a computer program at any time is completely determined by its **current inputs** and its **state**
- A **finite automaton** or **finite state machine** is always in one of a finite number of states
- At each step it makes a move that depends only on the state it's currently in and the input symbol it gets
- The move is to enter a particular state
  - Possibly the same as the one it was already in

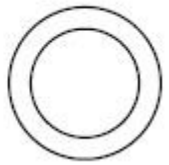
Finite Automata

- States are either **accepting** or **non-accepting**
  - Entering an accepting state means answering “yes”
  - Entering a non-accepting state means “no”
- Before a finite automaton has received any input, it is in its initial state, which is an accepting state precisely if the null string is accepted

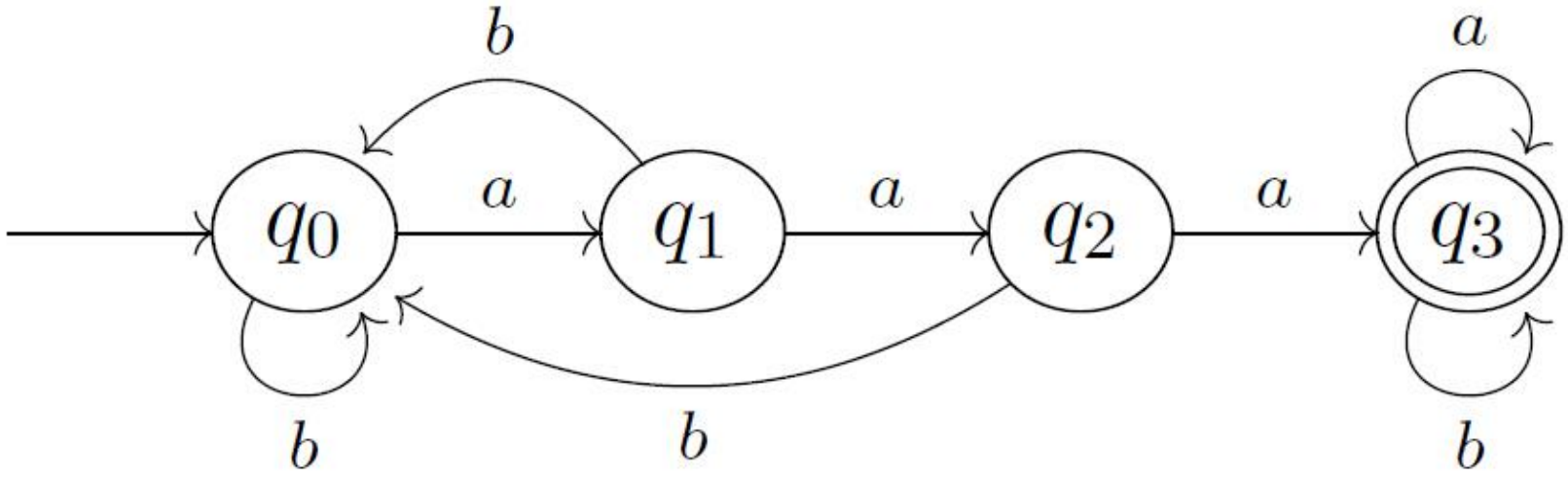
Finite Automata

- A finite automaton can be described by the **set of states**, the **input alphabet**, the **initial state**, the **set of accepting states**, and **a transition function**
- The transition function is also called a next state function meaning that the automaton moves into the next state if it receives the input symbol a while in the current state
- The finite automaton can be described by a diagram or a table of values
  - In a diagram, states are represented by circles, transitions by arrows labeled with input symbols, and accepting states by double circles





Finite Automata: Example



States: q_0, q_1, q_2, q_3.

Input symbols: a, b.

Transitions: as indicated above.

Start state: q_0.

Accepting state(s): q_3.

Deterministic Finite Automata

- If, for each pair of states and possible input symbols, there is a **unique** next state (as specified by the transitions), then the finite automaton is **deterministic** (**DFA**)
  - Otherwise, the finite automaton is **nondeterministic** (**NFA**)
  - i.e., For some state and input symbol, the next state may be nothing or one or two or more possible states
  - In fact, every DFA is also an NFA
- A DFA is a mathematical description of a simple machine capable of accepting/rejecting input words
  - i.e., A method to decide language membership

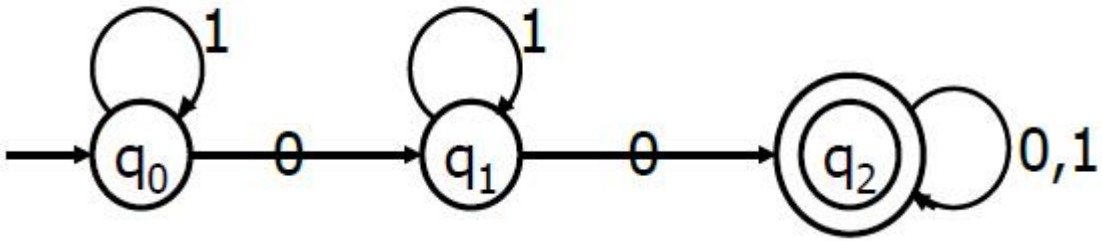
DFA: Formal Definition

- A DFA is a tuple $A = (Q,\Sigma,\delta,q_0,F)$ given by

1. A finite set of states Q
2. A finite set of input symbols (alphabets) Σ
3. A transition function δ: $Q \times \Sigma \rightarrow Q$
4. An initial state $q_0 \in Q$
5. A set of accepting (or final) states $F \subseteq Q$

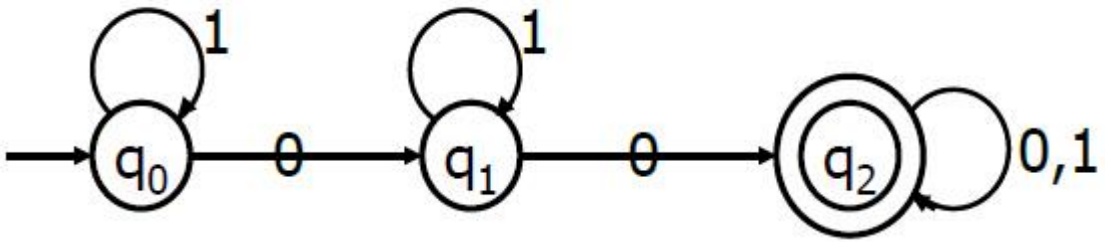
- From state q the machine will move to state $\delta(q, s)$ if it receives input symbol s

DFA: Example



- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{0,1\}$
- $q_0 = q_0$,
- $F = \{q_2\}$
- δ is given by 6 equalities
  - $\delta(q_0,0) = q_1$, $\delta(q_0,1) = q_0$, $\delta(q_1,1) = q_1$, $\delta(q_1,0) = q_2$, $\delta(q_2,0) = q_2$, $\delta(q_2,1) = q_2$

Transition Table



- All the information presenting a DFA can be given by a single thing, known as its **transition table**

	0	1
$\rightarrow q_0$	q_1	q_0
q_1	q_2	q_1
$*q_2$	q_2	q_2

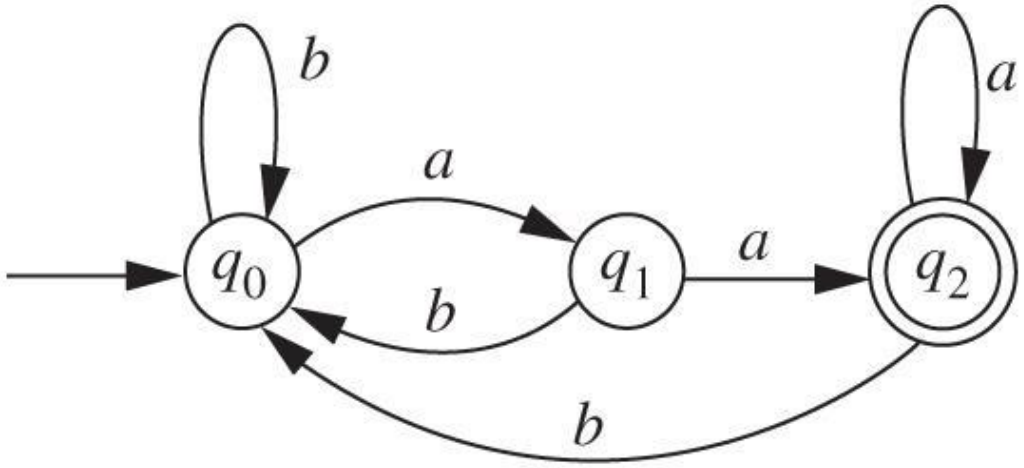
- The initial and final states are denoted by → and * respectively

How to Use a DFA

- Input: a word w in $\sum^*$
- Question: is w acceptable by the DFA?
- Steps:

1. Start at the "initial state" q_0
2. For every input symbol in the sequence w do
   - Compute the next state from the current state, given the current input symbol in w and the transition function
3. If after all symbols in w are consumed, the current state is one of the final states (F) then **accept** w
4. Otherwise, **reject** w

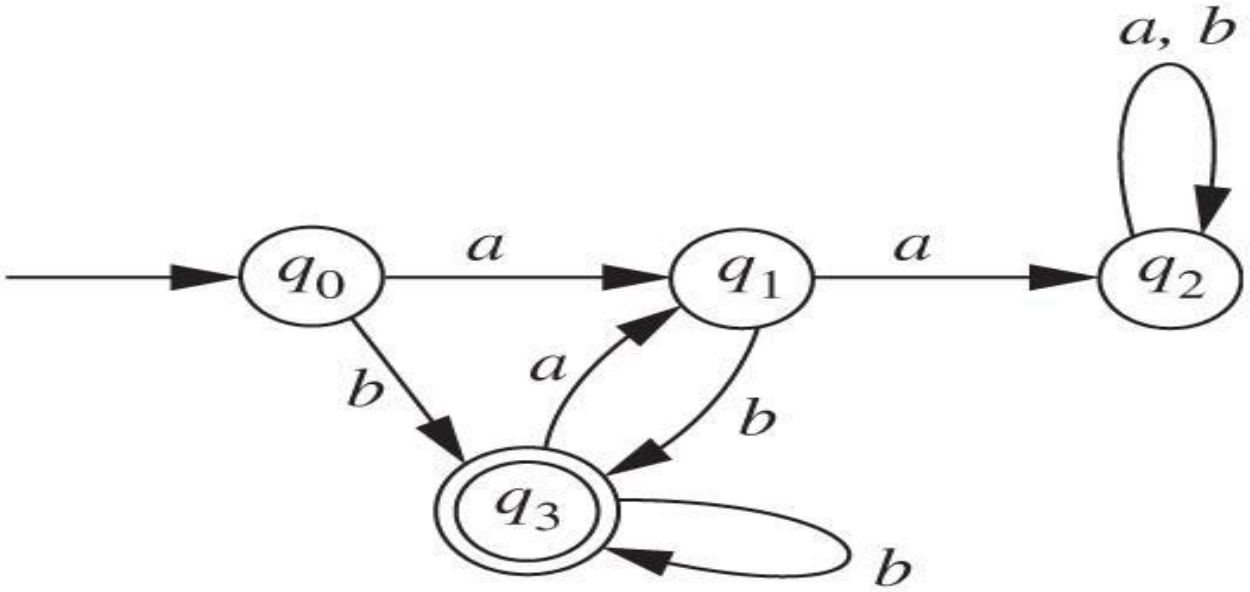
Example



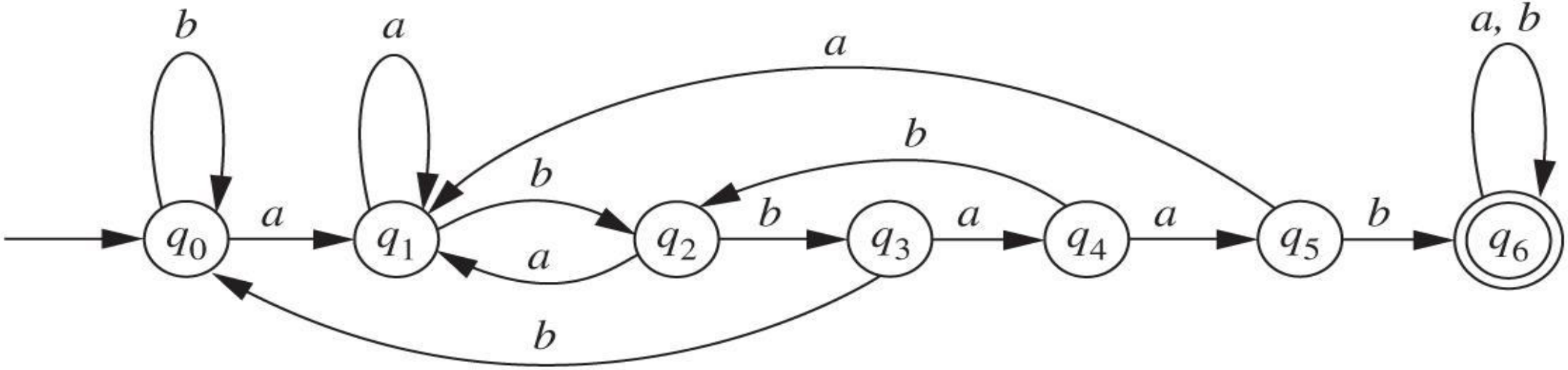
- What language does the above DFA accept?
- This DFA accepts the language of strings that end in aa
- The three states represent strings that end with no a's, one a, and two a's, respectively
- From each state, if the input is anything but an a, go back to the initial state, because now the current string doesn't end with a

Example

- Construct a DFA accepts the strings that end in *b* and do not contain *aa*
- The idea is to go to a permanently-non-accepting state if you ever read two *a*'s in a row
- Go to an accepting state if you see a *b* (and haven't read two *a*'s), leave it when you see anything else



Example



- What language does the above DFA accept?
- This DFA accepts the strings that contain *abbaab*
- What do we do when a prefix of *abbaab* has been read but the next symbol doesn't match?
  - Go back to the state representing the longest prefix of *abbaab* at the end of the new current string
  - If we've read *abba* and the next symbol is *b*, go to q_2, because *ab* is the longest prefix at the end of *abbab*

The Extended Transition Function

- The notation $\delta^*(q, x)$ describes the state the DFA is in after starting in state q and receiving input string x
- The extended transition function $\delta^*: Q \times \sum^* \rightarrow Q$ is defined recursively as follows:
  - For every $q \in Q$, $\delta^*(q, \varepsilon) = q$
  - For every $q \in Q$, every $y \in \sum^*$, and every symbol $s \in \sum$, $\delta^*(q, y.s) = \delta(\delta^*(q, y), s)$
- This just says that if you already know how to process the string y, then to process one additional symbol s, you use the ordinary transition function δ starting from the state $\delta^*(q, y)$

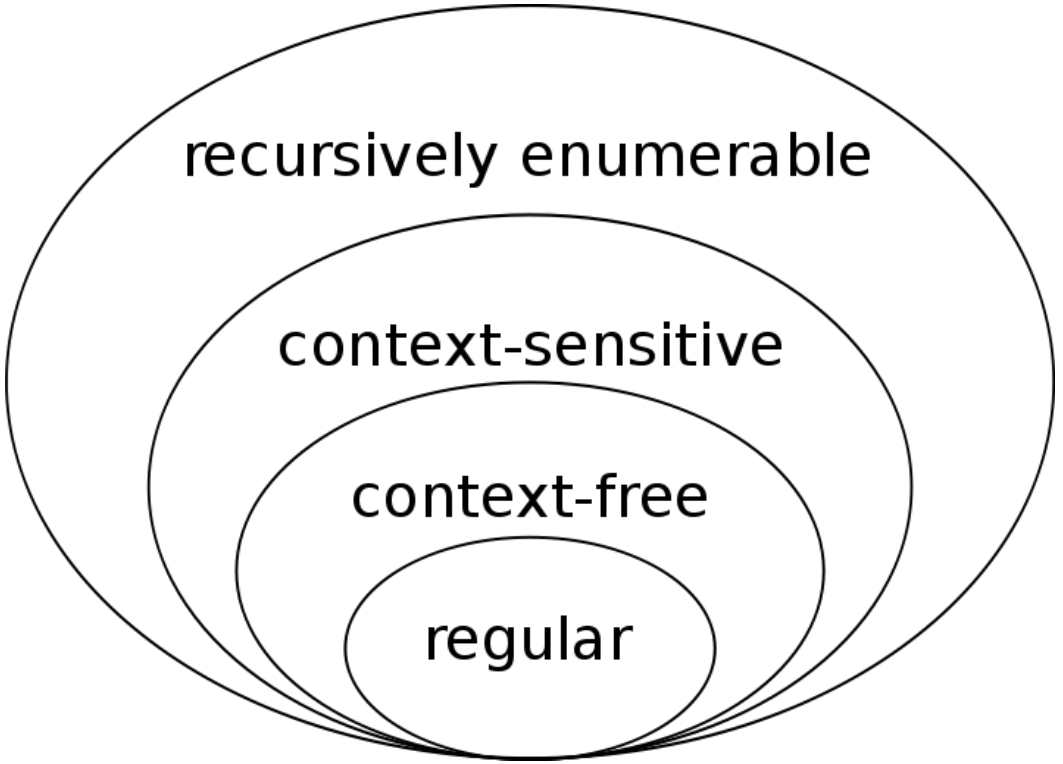
The Language of a DFA

- To each DFA A we associate a language $L(A) \subseteq \sum^*$. To see whether a word $w \in L(A)$ we put a marker in the initial state and when reading a symbol forward the marker along the edge marked with this symbol. When we are in an accepting state at the end of the word then $w \in L(A)$, otherwise $w \notin L(A)$
- Formally, let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA and let $w \in \sum^*$
  - w is accepted by A if $\delta^*(q_0, w) \in F$
  - w is rejected by A if $\delta^*(q_0, w) \notin F$
- Therefore the language of a DFA A is defined by

$$L(A) = \{w \in \sum^* \mid w \text{ is accepted by } A\}$$

Regular Languages

- A language is **regular** *iff* it is the set of strings accepted by some DFA
- Locate **regular languages** in the Chomsky Hierarchy
- Regular languages are very useful in input parsing and programming language design



Accepting the Union of Two Languages

- Suppose that L_1 and L_2 are languages over $\sum$
- Given a DFA that accepts L_1 and another DFA that accepts L_2, we can construct one that accepts $L_1 \cup L_2$
- The same approach works for intersection and difference as well
- The idea is to construct a DFA that executes both of the original deterministic finite automata at the same time
- This works because if $x \in \sum^*$, then knowing whether $x \in L_1$ and whether $x \in L_2$ is enough to determine whether $x \in L_1 \cup L_2$

Accepting the Union of Two Languages

- Suppose $A_1 = (Q_1,\Sigma,\delta_1,q_1,F_1)$ and $A_2 = (Q_2,\Sigma,\delta_2,q_2,F_2)$ are two deterministic finite automata that accepting L_1 and L_2. Let $A = (Q,\Sigma,\delta,q_0,F)$ be defined as follows

  $Q = Q_1 \times Q_2$ (Cartesian product)

  $q_0 = (q_1, q_2)$

  $\delta((p, q), s) = (\delta_1(p, s), \delta_2(q, s))$

- Then, if :

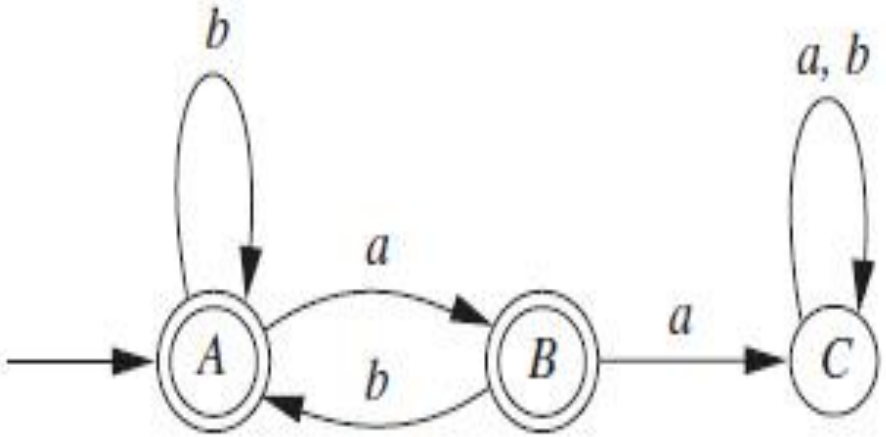
  $F = \{(p, q) \mid p \in F_1 \text{ or } q \in F_2\}$, A accepts $L_1 \cup L_2$

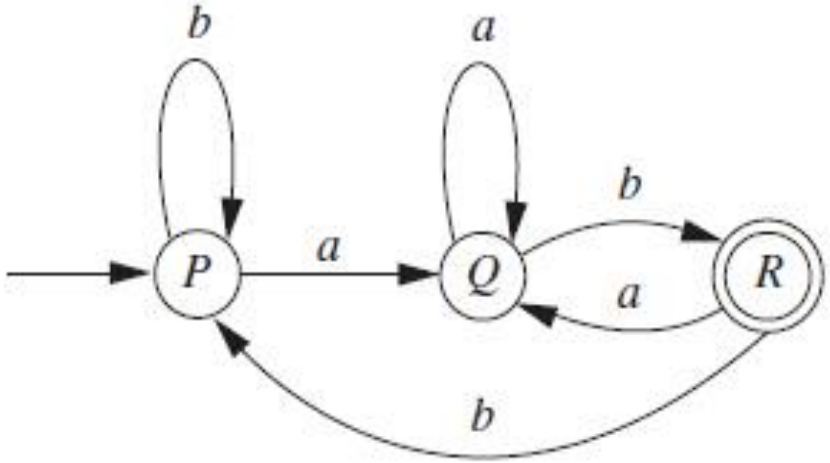
  $F = \{(p, q) \mid p \in F_1 \text{ and } q \in F_2\}$, A accepts $L_1 \cap L_2$

  $F = \{(p, q) \mid p \in F_1 \text{ and } q \notin F_2\}$, A accepts L_1-L_2

Example

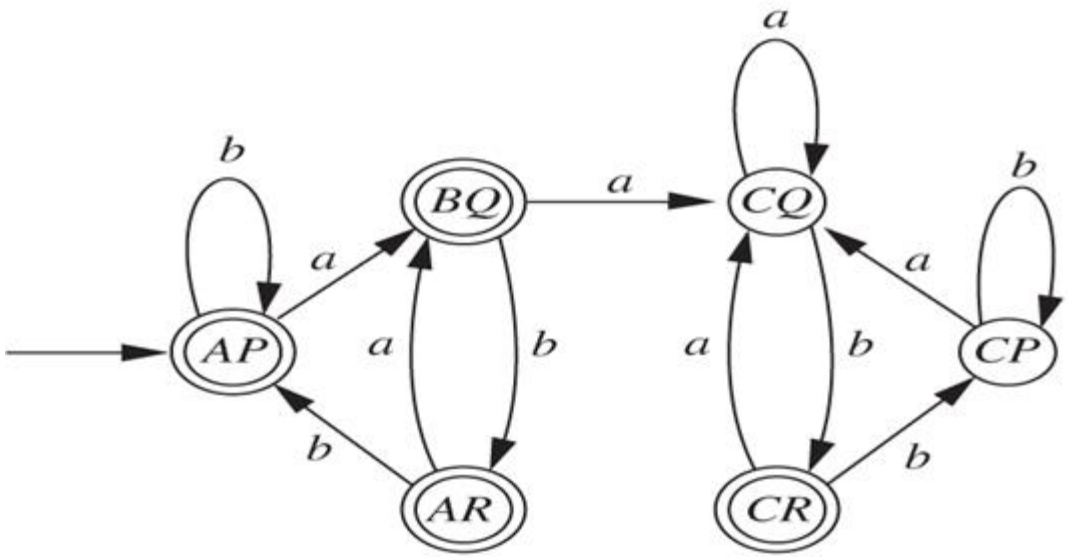
- $\sum = (a,b)$
- Draw the DFA accepting the language, L_1 and L_2, *s.t.*
  $L_1 = \{x \in \sum^* \mid aa \text{ is not a substring of } x\}$
  $L_2 = \{x \in \sum^* \mid x \text{ ends with } ab\}$





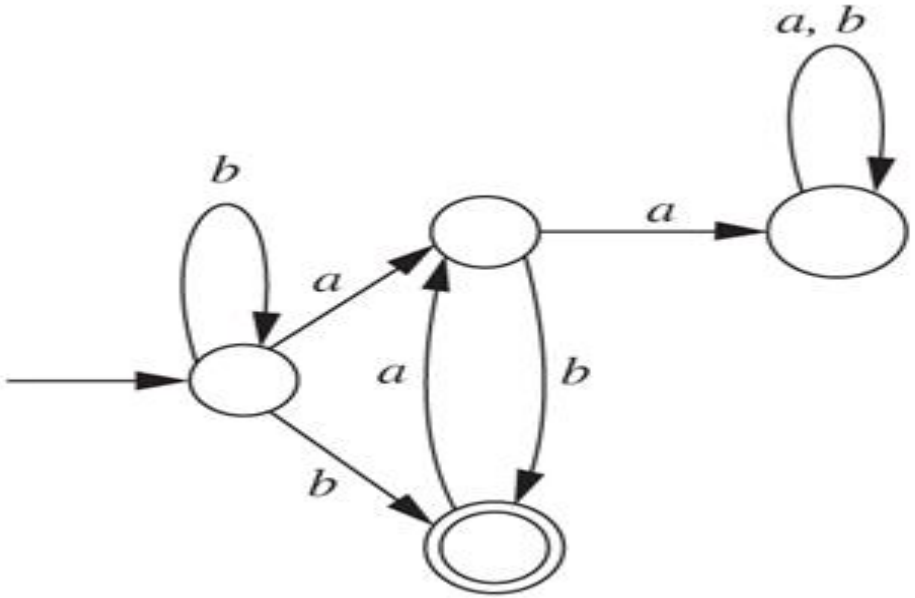
Example

- Draw the DFA accepting the union of language, L_1 and L_2
- Given two machines, create the Cartesian product of the state sets, and draw the necessary transitions
- Simplify the resulting machine, if possible, and designate the appropriate accepting states



Example

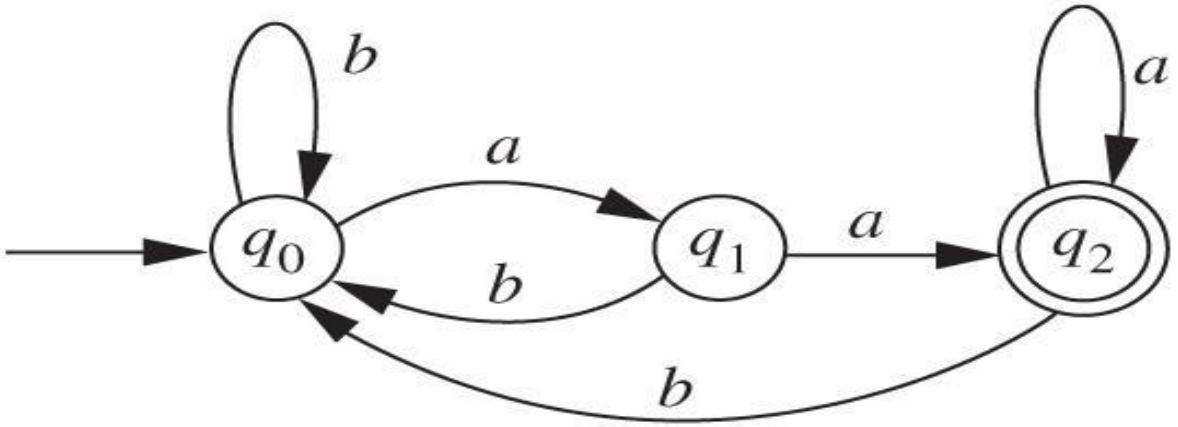
- For the intersection, we can simplify further



- The simplification involved turning states CP, CQ, and CR into a single state (none of them was accepting, and there was no way to leave them)

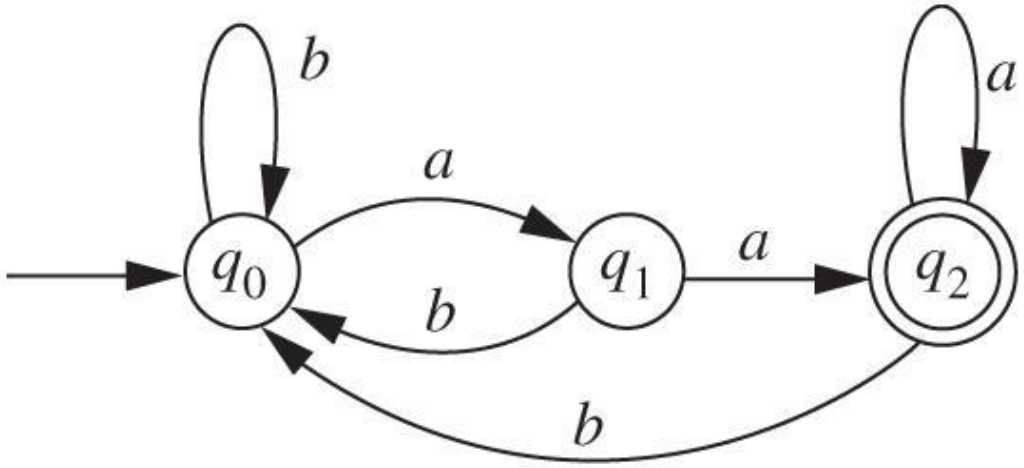
Idea: Distinguishable

- Think about a DFA, A
- If we have two strings, x and y, that can lead to the same state, q, we say x and y **cannot be distinguished** by the machine A
  - i.e. the machine cannot tell the strings apart
- Example



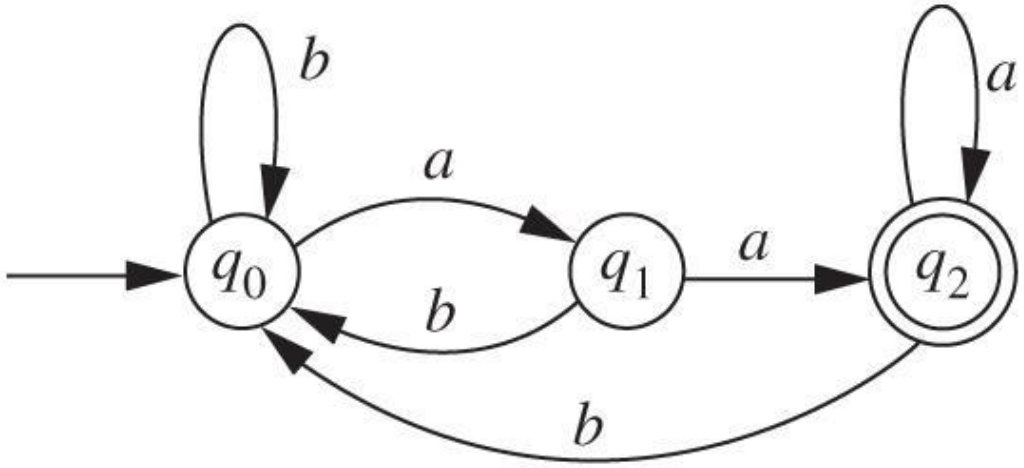
  - $\delta^*(q_0, bbbbba) = \delta^*(q_0, aabba) = q_1$
  - Both $bbbbba$ and $aabba$ lead to stage q_1

Language Distinguishable



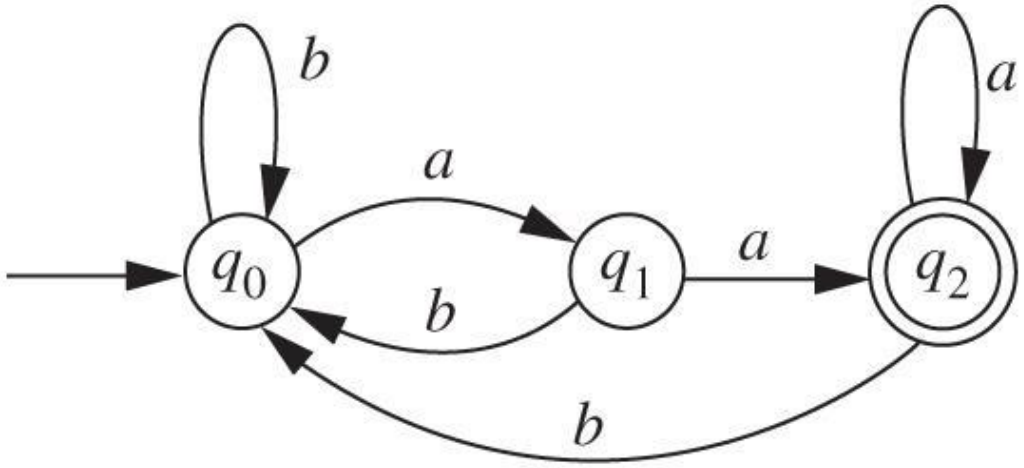
- This DFA accepts the language of strings that end in *aa*
- *L = {aa, aaa, baa, aaaa, abaa, baaa, bbaa, aaaaa, …}*

Language Distinguishable



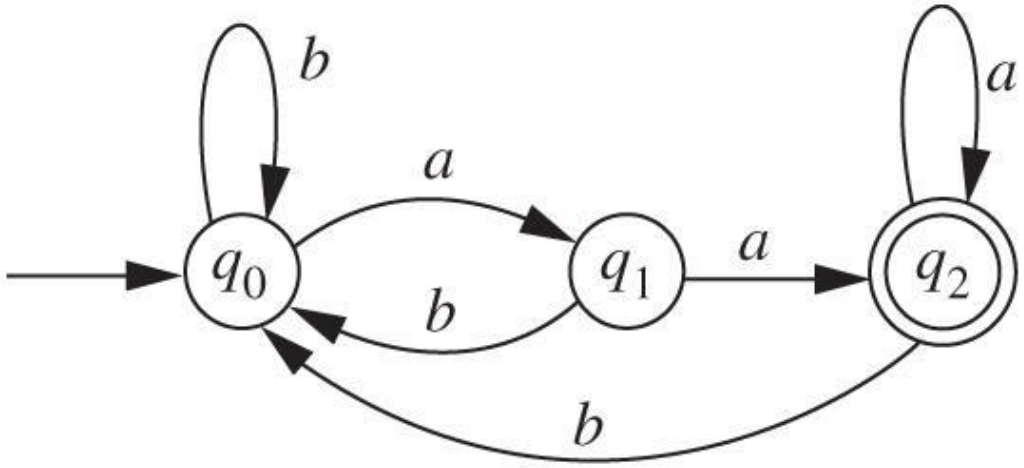
- $L = \{aa, aaa, baa, aaaa, abaa, baaa, bbaa, aaaaa, \ldots\}$
- We say that string a and string b are L-distinguishable by the string a since $aa \in L$ and $ba \notin L$
- A string a having this property is said to distinguish a and b with respect to L

Language Distinguishable



- $L = \{aa, aaa, baa, aaaa, abaa, baaa, bbaa, aaaaa, \ldots\}$
- Likewise, the string ε and string α are L-distinguishable by the string α since $\alpha\alpha \in L$ and $\varepsilon\alpha \notin L$
- A string α having this property is said to distinguish ε and α with respect to L

Language Distinguishable



- $L = \{aa, aaa, baa, aaaa, abaa, baaa, bbaa, aaaaa, \ldots\}$
- And, we say that string a and string aa are L-distinguishable by ε since $aa\varepsilon \in L$ and $a\varepsilon \notin L$
- The string ε having this property is said to distinguish a and aa with respect to L

Distinguishing One String from Another

- Any three-states DFA, such as the one that accepts the strings ending in *aa*, ignores, or "forgets", a lot of information
- *aba* and *aabbabbabaaaba* lead to the same state; there is no way for the DFA to remember which string has been seen
- *aba* and *ab*, however, lead to different states; the essential difference is that one ends with *a* and the other doesn't
- *aba* and *ab* are distinguishable with respect to the language accepted by the DFA
  - There is at least one string *z* (such as *a*) so that *abaz* is in the language (i.e., is accepted) and *abz* is not, or vice versa

Distinguishing One String from Another

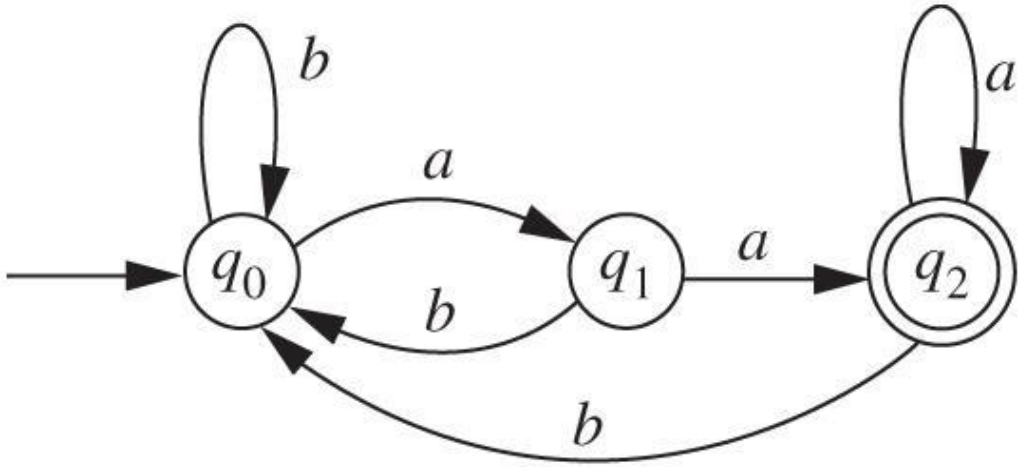
- **Definition**:
- If L is a language over the alphabet $\sum$, and x and y are strings in $\sum^*$, then x and y are distinguishable with respect to L, or L-distinguishable, if there is a string $z \in \sum^*$such that either $xz \in L$ and $yz \notin L$, or $yz \in L$ and $xz \notin L$
- A string z having this property is said to distinguish x and y with respect to L
- Equivalently, x and y are L-distinguishable if $L/x \neq L/y$,

$$\text{where } L/x = \{z \in \sum^* \mid xz \in L\}$$

Pairwise Distinguishable

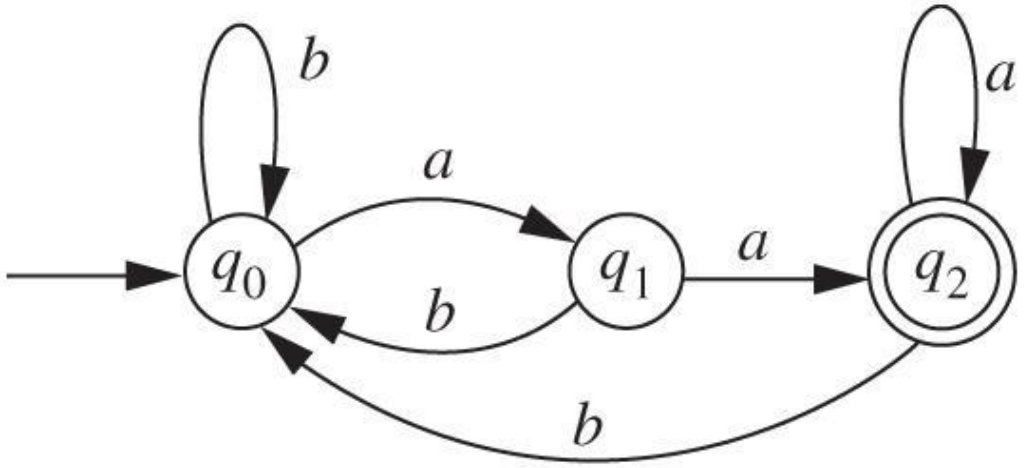
- The strings in a set $S \subseteq \sum^*$ are **pairwise *L*-distinguishable** if for every pair x, y of distinct strings in S, x and y are L-distinguishable

- Example:



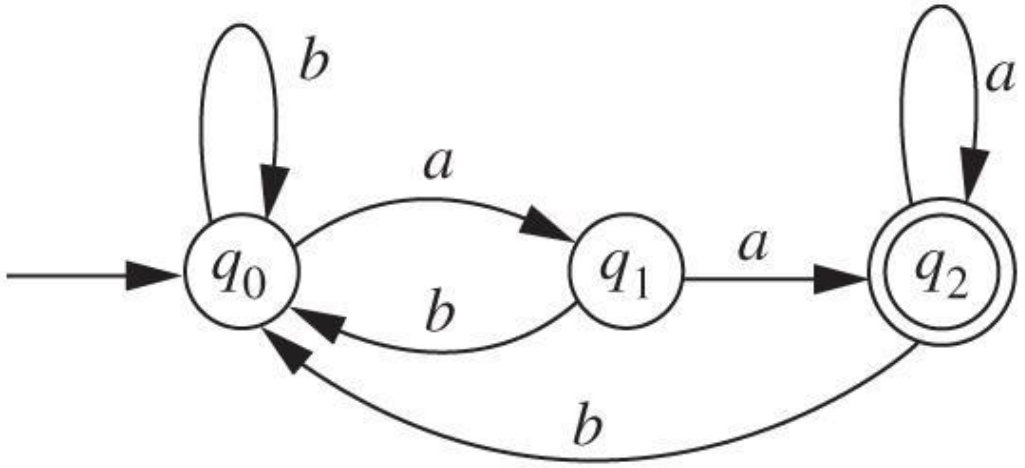
- $S = \{a, b, aa\}$ are pairwise L-distinguishable for A
- $S = \{\varepsilon, a, aa\}$ are also pairwise L-distinguishable for A

Pairwise Distinguishable



- $S = \{a, b, aa\}$ are pairwise L-distinguishable for A
- $\{a, b\}$:$aa \in L$ and $ba \notin$ L, (a distinguishes a and b wrt L)
- $\{a, aa\}$:$aa\varepsilon \in L$ and $a\varepsilon \notin$ L, (ε distinguishes a and aa wrt L)
- $\{b, aa\}$:$aa\varepsilon \in L$ and $b\varepsilon \notin$ L, (ε distinguishes b and aa wrt L)

What Do We Notice?

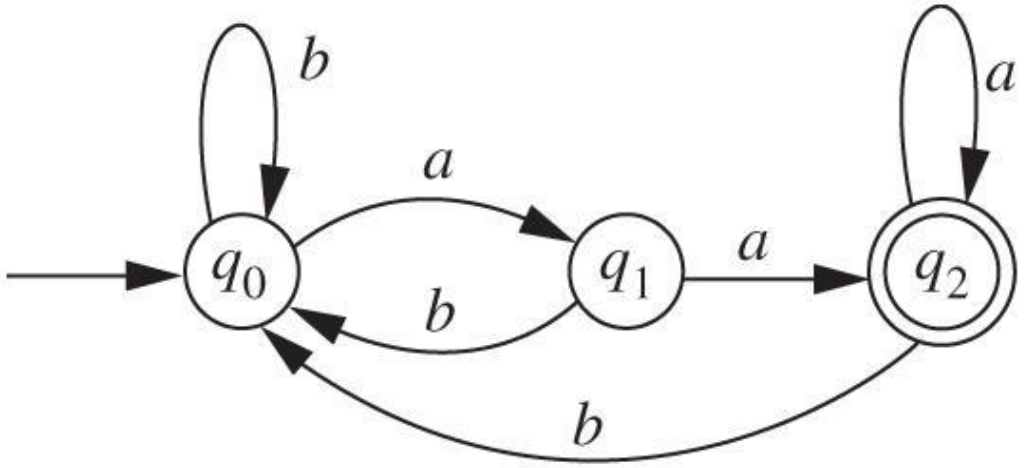


- $S = \{a, b, aa\}$ are pairwise L-distinguishable for A
  - How many elements of S? How many states of A?
- In order to L-distinguish between 3 pairwise L-distinguishable strings, then the machine A must have **at least** 3 states

Distinguishing Strings: Theorem

- **Theorem**
- Suppose $A = (Q,\Sigma,\delta,q_0,F)$ is a DFA that accepts the language $L \subseteq \sum^*$.
- If x and y are two strings in $\sum^*$ that are L-distinguishable, then $\delta^*(q_0, x) \neq \delta^*(q_0, y)$.
  - i.e., if x, y, are L-distinguishable, the extended transition function of strings x, y cannot result in the same state
- For every $n \geq 2$, if there is a set of n pairwise L-distinguishable strings in $\sum^*$, then Q must contain at least n states.
  - i.e., a finite machine A must have at least as many states as the set of n pairwise L-distinguishable strings in $\sum^*$

Back to Previous Example



- $S = \{a, b, aa\}$ are pairwise L-distinguishable for A
- The set S contains 3 pairwise L-distinguishable strings
- **For every $n \geq 2$, if there is a set of n pairwise L-distinguishable strings in $\sum^*$, then Q must contain at least n states**
- This shows why we need at least **three** states in any machine that accepts the language L of strings ending in aa

Minimum Number of States, Q

- We now know the minimum number of states a finite machine must have:

For every $n \geq 2$, if there is a set of n pairwise L-distinguishable strings in $\sum^*$, then Q must contain at least n states

- What happens if there is an **infinite** set S of pairwise L-distinguishable strings for a language L?

Then L **cannot be accepted** by a finite automaton!

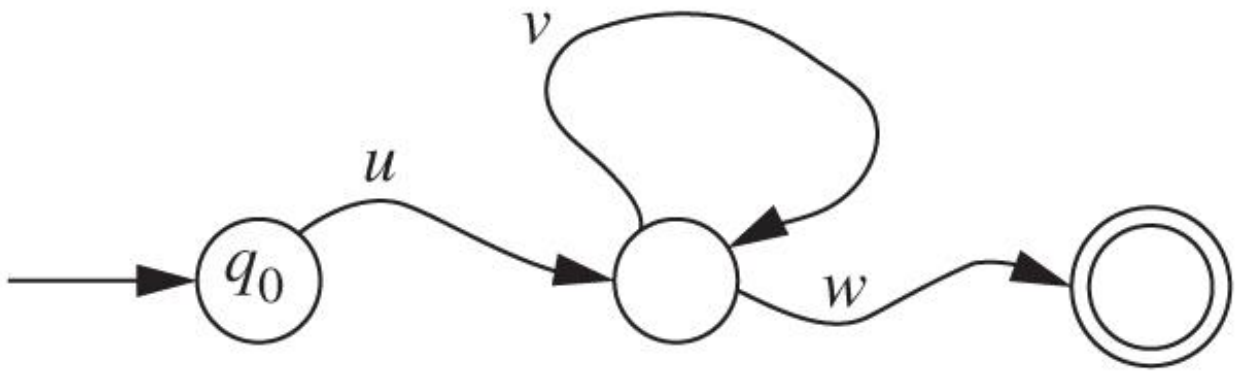
The Pumping Lemma

- Suppose that $A = (Q,\Sigma,\delta,q_0,F)$ is a DFA accepting L and that it has n states
- Let x be a string in L with $|x| = n\text{-}1$
- Then x has n distinct prefixes
  - ε is a prefix of every string
  - Every string is a prefix of itself
  - The number of distinct prefixes of x is, $|x| + 1 = n$
- Therefore it is **possible** that A is in a different state after processing every prefix of x
  - In this case every prefix leads to **exactly one** distinct state in A

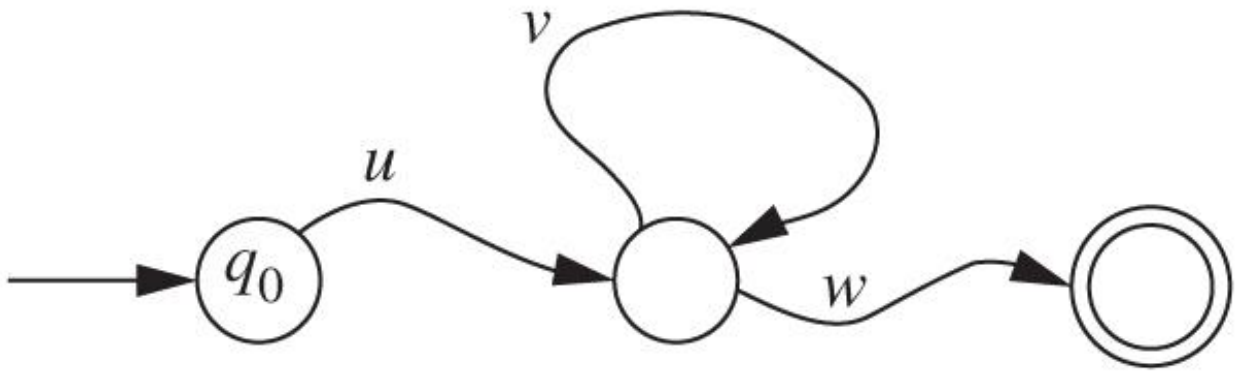
The Pumping Lemma

- Suppose that $A = (Q,\Sigma,\delta,q_0,F)$ is a DFA accepting L and that it has n states
- If it accepts a string x such that $|x| \geq n$, then by the time n symbols have been read, A must have entered some state more than once
- i.e., there must be two different prefixes u and uv such that

$$\delta^*(q_0, u) = \delta^*(q_0, uv)$$



The Pumping Lemma



- This implies that there are many more strings in L, because we can traverse the loop v any number of times (including leaving it out altogether)
- In other words, all of the strings uv^iw for $i \geq 0$ are in L
- This fact is known as the **Pumping Lemma** for **Regular Languages**
  - i.e. all sufficiently long words in a regular language may be pumped – that is, have a middle section of the word repeated an arbitrary number of times—to produce a new word that also lies within the same language

The Pumping Lemma: Theorem

- **Theorem**
- Suppose L is a language over $\sum$.
- If L is accepted by the DFA $A = (Q,\Sigma,\delta,q_0,F)$ then there exists an integer n so that for every x in L satisfying $|x| \geq n$, there are three strings u, v, and w such that $x = uvw$ and

1. $|uv| \leq n$
2. $|v| > 0$ (i.e. $v \neq \varepsilon$)
3. For every $i \geq 0$, the string uv^iw belongs to L

- In simple words, for any regular language L, any sufficiently long word $x \in L$ can be split into 3 parts. i.e. $x = uvw$, such that all the strings uv^iw for $i \geq 0$ are also in L

The Pumping Lemma

- The way we found n was to take the number of states in a DFA accepting L. In many applications we don't need to know this, only that **there is** such an n
- The most common application of the pumping lemma is to show that a language **cannot be accepted** by a DFA, because it doesn't have the properties that the pumping lemma says are required for every language that can be
- The proof is by contradiction. We suppose that the language can be accepted by a DFA, and we let n be the integer in the pumping lemma
- Then we choose a string x with $|x| \geq n$ to which we can apply the lemma so as to get a contradiction

The Pumping Lemma: Example

- Let L be the language $\{a^i b^i \mid i \geq 0\}$, prove that it cannot be accepted by a DFA

Proof:

- Suppose, for the sake of contradiction, that L is accepted by a DFA; $\exists n$ such that…
- Choose $x = a^n b^n$; then $x \in L$ and $|x| = 2n \geq n$
- Therefore, by P.L, there are strings u, v, and w such that $x = uvw$ and the 3 conditions hold
- Because $|uv| \leq n$ and x starts with n a's, all the symbols in u and v are a's; therefore, $v = a^k$ for some $k > 0$
- So, let $u = a^s$, $v = a^k$, $w = a^t b^n$

The Pumping Lemma: Example

- Let L be the language $\{a^i b^i \mid i \geq 0\}$, prove that it cannot be accepted by a DFA

Proof:

- $s + k \leq n$, $k > 0$, $t \geq 0$, $s + k + t = n$
- Therefore the last condition of P.L fails for $i = 0$

$$uv^0w = uw = a^s a^t b^n = a^{s+t} b^n \notin L \text{ since } s+t \neq n$$

- It is not the case that for all i uv^iw is in L
- This is our contradiction, and we conclude that L cannot be accepted by a DFA

The Pumping Lemma

- There are other languages that are not accepted by any DFA, among them:
  - *Balanced*, the set of balanced strings of parentheses
  - *Expr*, the language of simple algebraic expressions
  - The set of legal C programs
- In all three examples, because of the nature of these languages, a proof using the pumping lemma might look a lot like the proof for our previous example

Simple Machine Using Equivalence Classes

- Consider A, a DFA accepting the language of strings ending in aa
- We've shown that three states are needed
- In particular, we can confirm that A really does accept L by showing that if x and y are two strings that cause A to end in the same state, then A doesn't need to distinguish them, because they are not L-distinguishable
- The three states of A correspond to three sets of strings: those not ending in a, those ending in a but not aa, and those ending in aa
  - e.g., consider a string x in the 2^{nd} set, we don't exactly know what x is, only that x ends with a but not aa. For every string z, if $z = a$ or z itself ends with aa, then $xz \in L$. Therefore no two strings in this set are L-distinguishable

Simple Machine Using Equivalence Classes

- We use L-**in**distinguishable to mean **not** L-distinguishable
- If S is any one of the previous 3 sets, then

1. Any two strings in S are L-indistinguishable
2. No string of S is L-indistinguishable from a string not in S

- These two facts say that the three sets are the equivalence classes for a certain equivalence relation
- **Definition**
- For a language $L \subseteq \{a,b\}^*$, we define the relation I_L on $\sum^*$ as follows: For $x, y, \in \sum^*$, $x\ I_L\ y$ if and only if x and y are L-indistinguishable
  - I_L is an equivalence relation

Simple Machine Using Equivalence Classes

- We can start building a DFA accepting L by using the induced equivalence classes as its states
- The initial state should be [ε], because when we start we have received no input
- The accepting state should be the equivalence class of strings ending in aa, since that's the language we want to accept
- The transitions are defined in a natural way:
  - Take any element x of one class, and consider xa or xb
  - The new string is in some equivalence class
  - The a-transition or b-transition from [x] simply goes to that class

Simple Machine Using Equivalence Classes

- **Theorem** (Myhill-Nerode): $L \subseteq \sum^*$ can be accepted by a DFA **if and only if** the set Q_L of equivalence classes of the relation I_L on $\sum^*$ is finite
- Conversely, if the set Q_L is finite, then the DFA $A_L = (Q_L, \Sigma, \delta, q_0, F)$ accepts L, where:

1. $q_0 = [\varepsilon]$
2. $F = \{q \in Q_L \mid q \subseteq L\}$,
3. For every $x \in \sum^*$ and every $s \in \sum$, $\delta([x], s) = [xs]$

- Finally, A_L has **the fewest** states of any DFA accepting L

Simple Machine Using Equivalence Classes

- It is often easier to construct a DFA directly than to determine the set of equivalence classes
- The previous theorem serves to answer the question of how much a computer accepting a language L needs to remember about the current string x: **only its equivalence class**
- Identifying the equivalence classes, if we already have a DFA accepting L, is not too hard

1. For each state q, we define $L_q = \{x \in \sum^* \mid \delta^*(q_0, x) = q\}$
2. Every L_q is a subset of some equivalence class of I_L (**is the whole class if the DFA has as few states as possible**)
   - Otherwise, for some q, L_q contains strings in two different equivalence classes, there would be two L-distinguishable strings that caused A to end up in the same state, which contradicts the theorem on slide 33

Simple Machine Using Equivalence Classes

- The Myhill-Nerode Theorem provides a way of showing a language cannot be accepted by a DFA (and it might apply even if the pumping lemma doesn't, since it's an ***if-and-only-if result***)
- Consider the equivalence classes of I_L for language $L = \{a^n b^n \mid n \geq 0\}$
- For $i \neq j$, a^i and a^j are L-distinguishable, because $a^i b^i \in L$ and $a^j b^i \notin L$
- This implies that there are an infinite number of equivalence classes, since there are infinitely many choices of i and j such that $i \neq j$
- Thus there can be no DFA accepting L

Minimizing Number of States in DFA

- Suppose we have a DFA $A = (Q,\Sigma,\delta,q_0,F)$ accepting the language $L \subseteq \sum^*$
- For a state q of A, define the equivalence class $L_q = \{x \in \sum^* \mid \delta^*(q_0, x) = q\}$

1. The first step in minimizing the number of states is to remove every state q for which $L_q = \varnothing$, along with transitions from these states; removing them has no effect on the language
2. Now define the equivalence relation $\equiv$ on Q:
   - $p \equiv q$ means that strings in L_p are L-indistinguishable from strings in L_q
3. This is the same as saying L_p and L_q are subsets of the same equivalence class of I_L

Minimizing Number of States in DFA

- Two strings x and y are L-distinguishable if, for some string z, exactly one of xz, yz is in L
- Therefore, $p \not\equiv q$ if, for some string z, exactly one of the states $\delta^*(p, z)$, $\delta^*(q, z)$ is in F
- Define S_M as the set of unordered pairs (p, q) of distinct states satisfying $p \not\equiv q$
- A systematic way of finding S_M is this:

1. If exactly one of p, q is in F, then $(p, q) \in S_M$
2. For every pair of states r and s, and every symbol t, if $(\delta(r,t), \delta(s,t)) \in S_M$, then $(r, s) \in S_M$

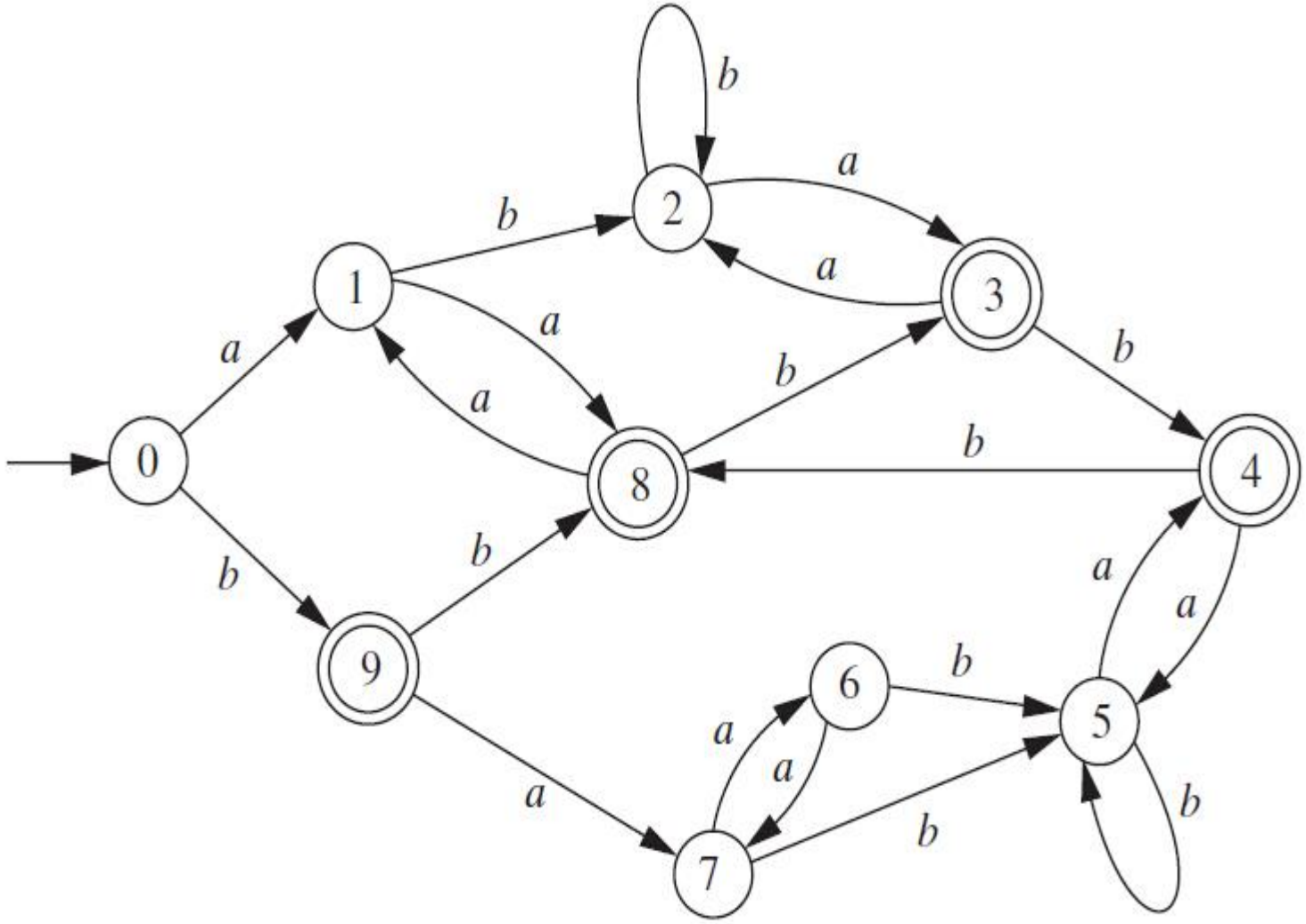
Minimizing Number of States in DFA

- An algorithm to identify the pairs (p, q) with $p \not\equiv q$:

1. List all unordered pairs (p, q) of distinct states
2. Make a sequence of passes through these pairs
3. On the first pass, mark each pair (p, q) so that exactly one of the two states is in F
4. On each subsequent pass, and each unmarked pair (r, s), if $\delta(r,t) = p$ and $\delta(s,t) = q$ for some $t \in \Sigma$, and (p, q) is marked, then mark (r, s)
5. After a pass in which no pairs are marked, stop
6. The marked pairs are the pairs (p, q) for which $p \not\equiv q$

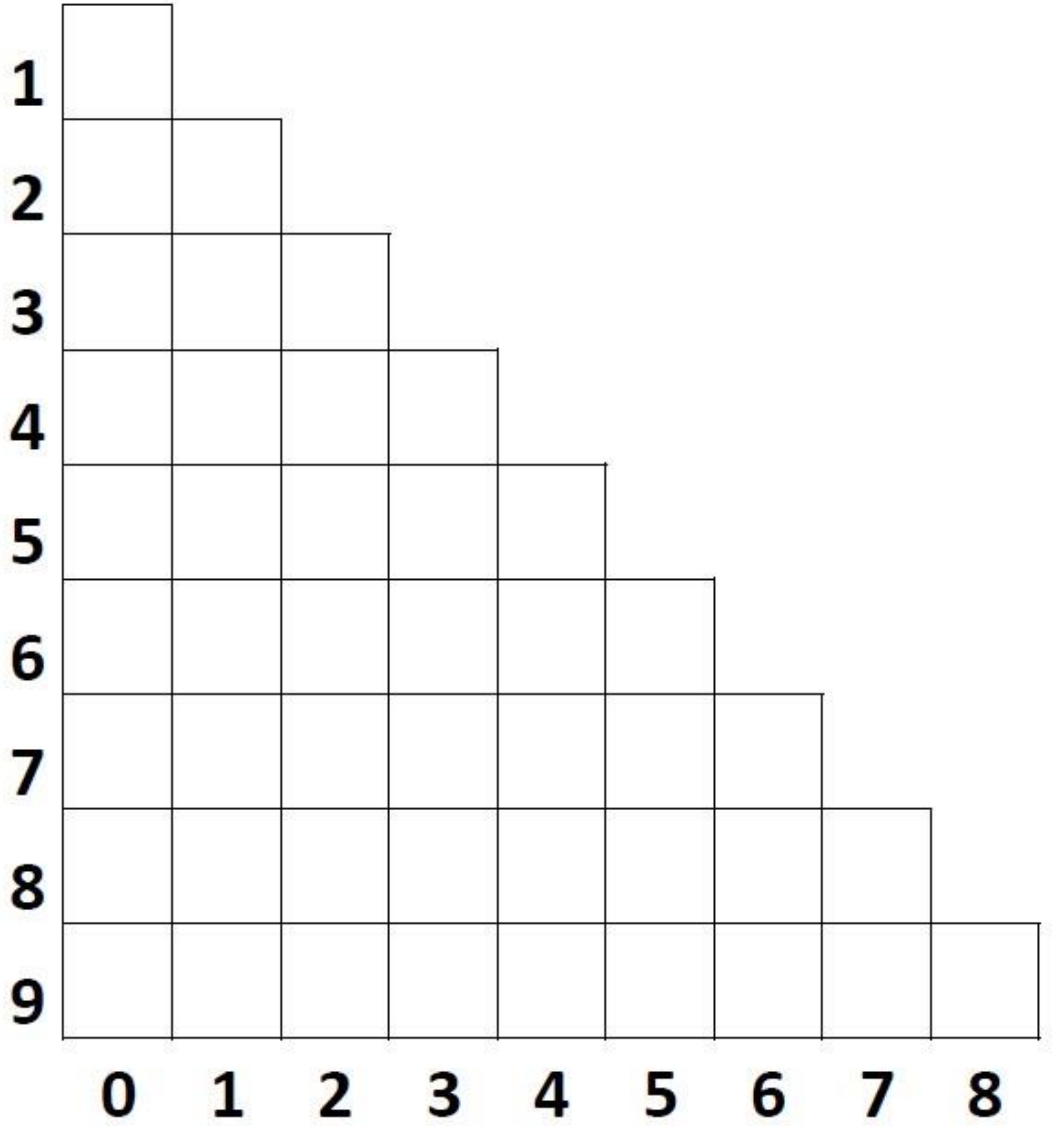
Example: Minimize the DFA

- We are trying to remove states that are **equivalent**



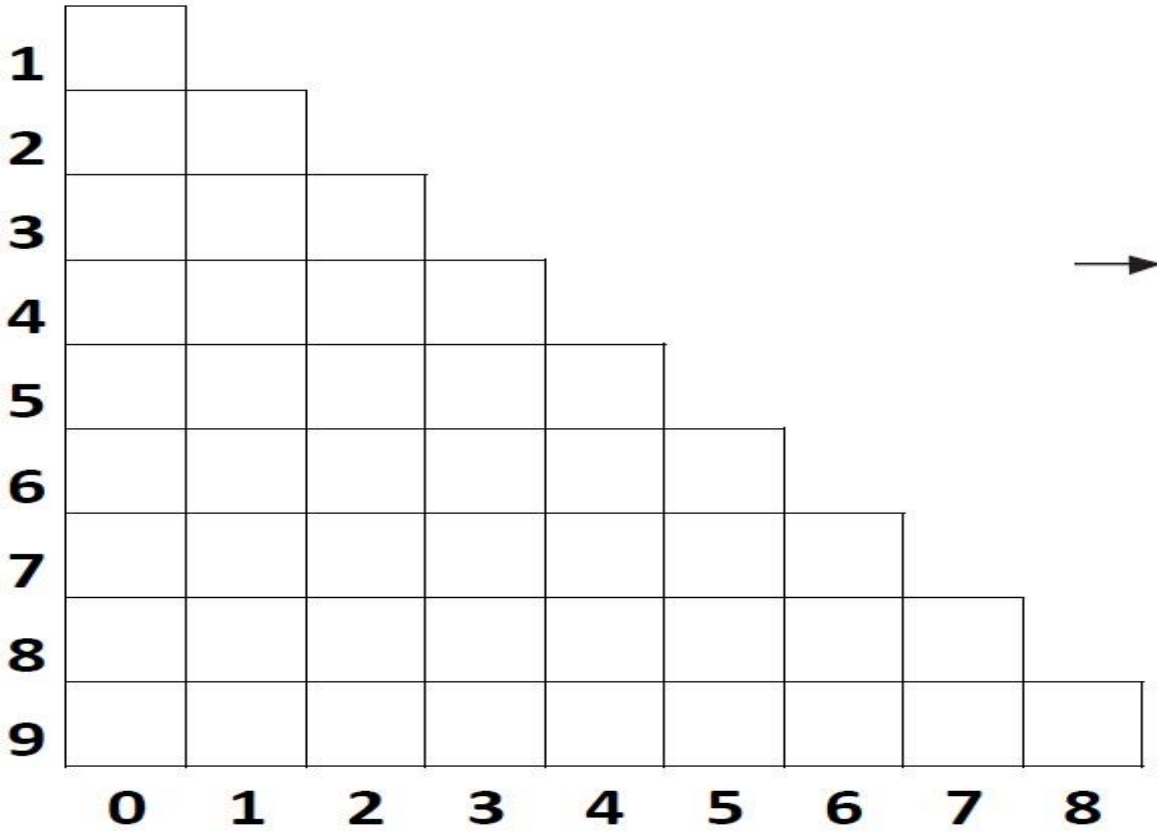
Example: Table Filling Algorithm

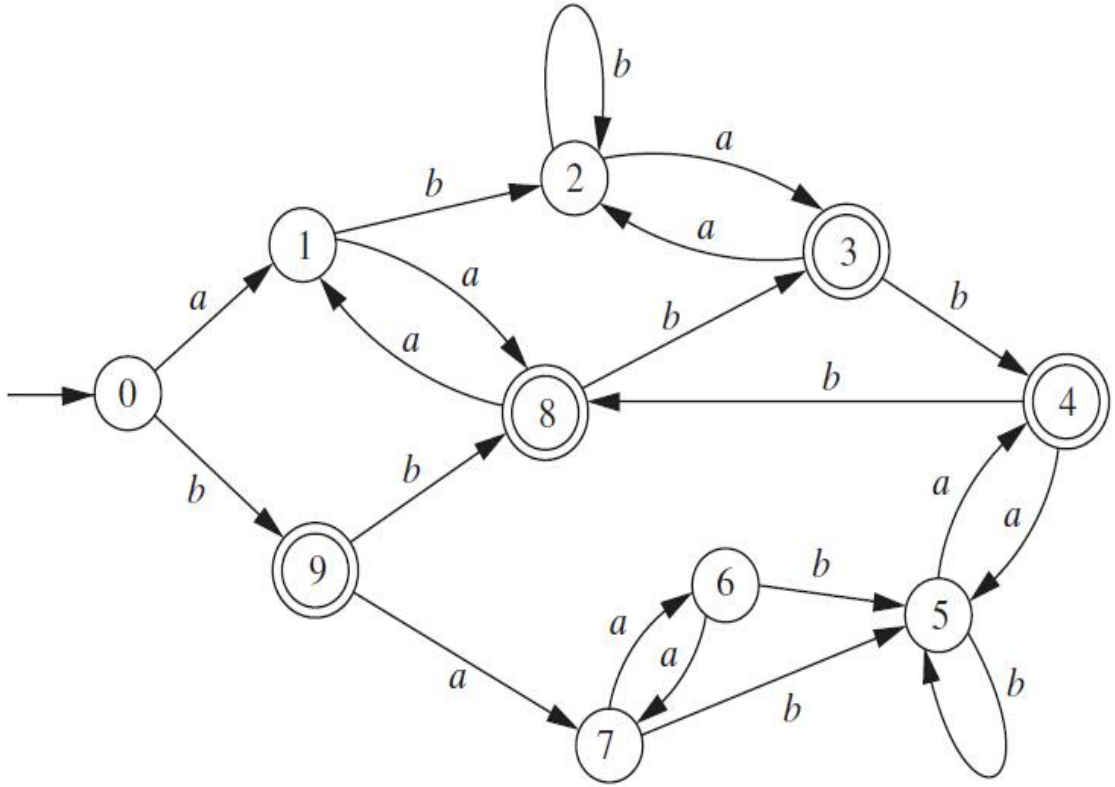
- First, draw a table containing all state pairs
- Each cell corresponds to an unordered pairs of distinct states, (p, q)



Step 1:Trivially Distinguishable

- Mark all pairs that trivially distinguish
  - Mark each pair (p, q) so that exactly one of the two states is in F

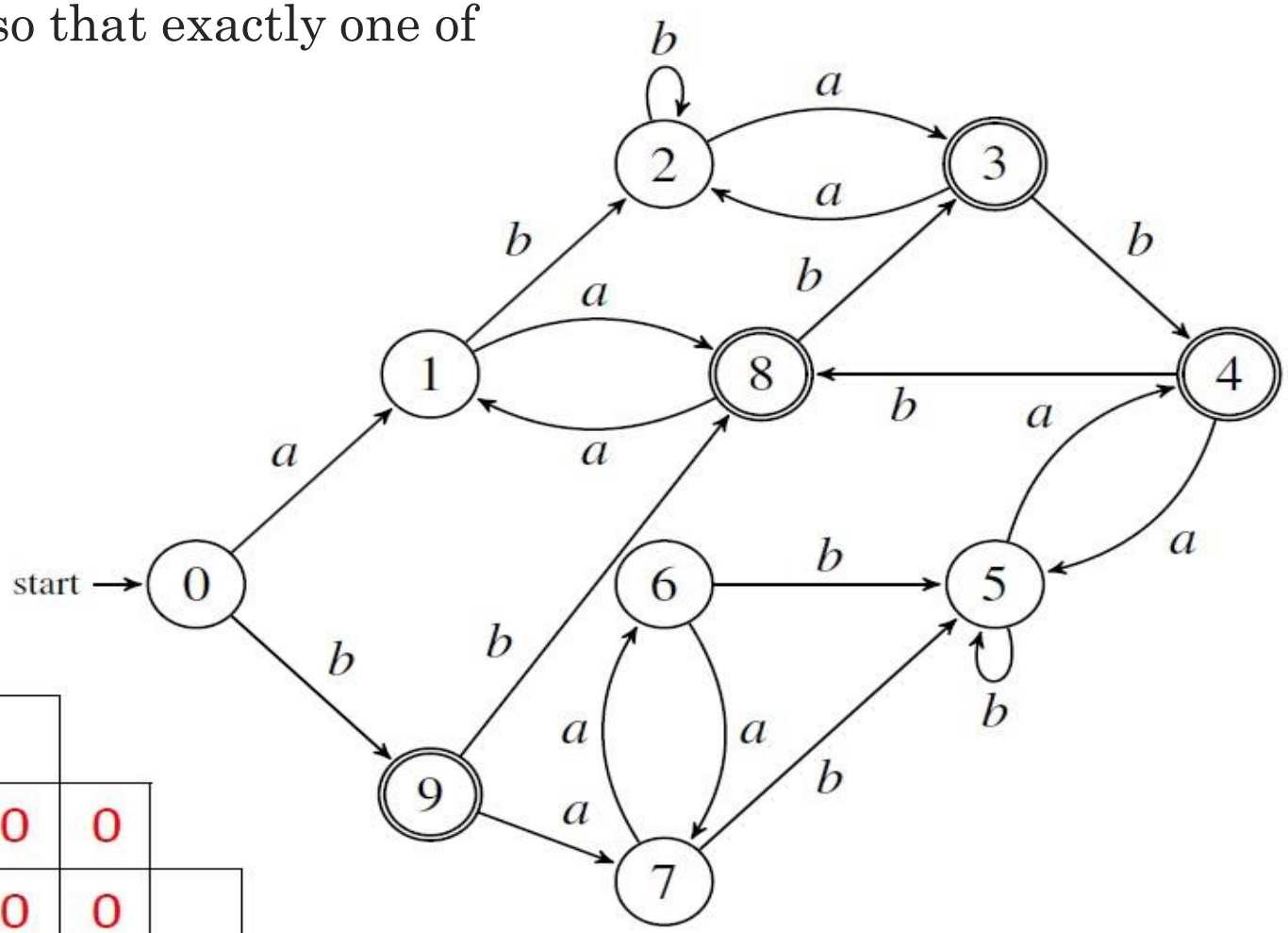




Step 1:Trivially Distinguishable

- Mark all pairs that trivially distinguish
  - Mark each pair (p, q) so that exactly one of the two states is in F
  - Mark "0"

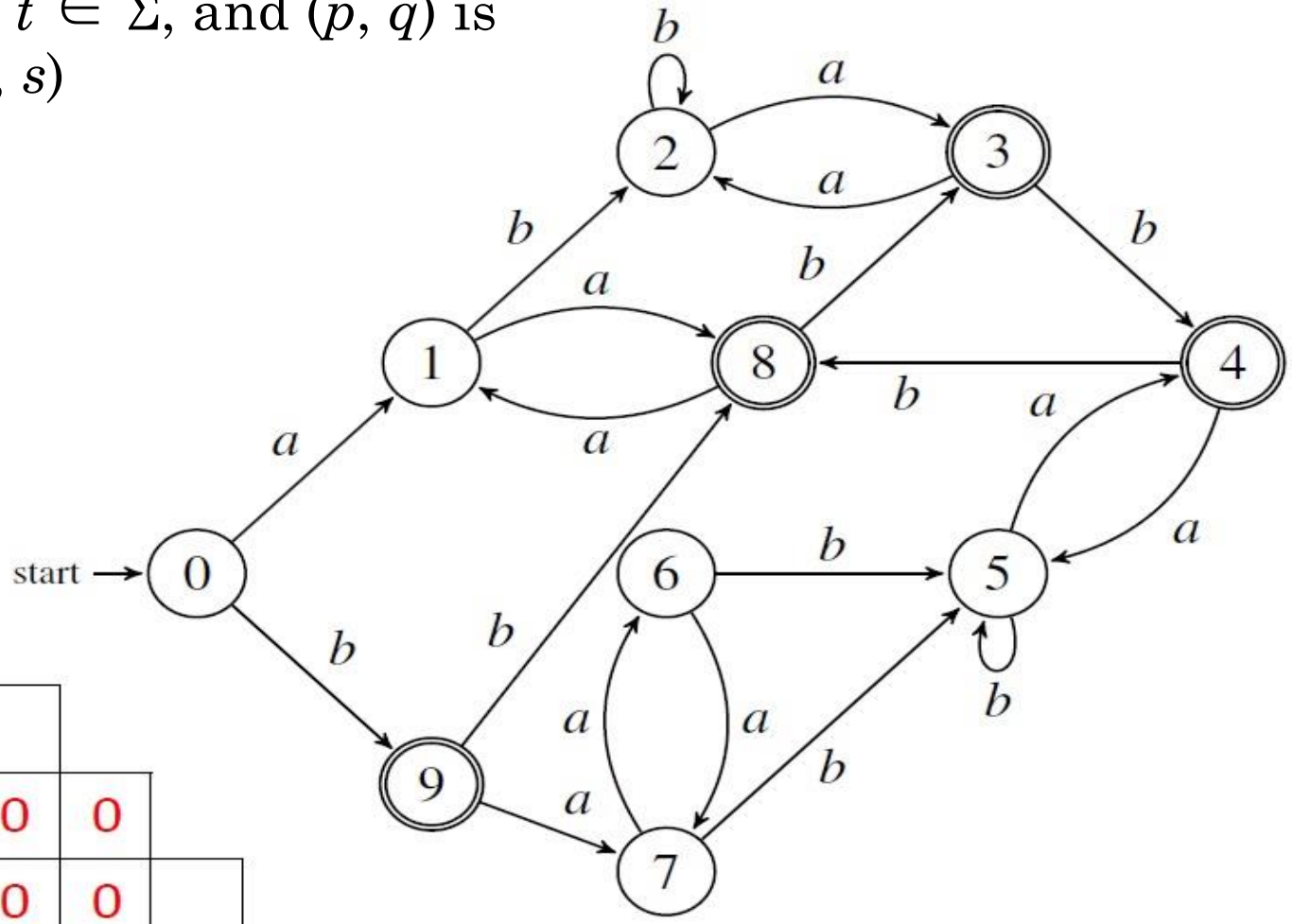
	0	1	2	3	4	5	6	7	8
1									
2									
3	0	0	0						
4	0	0	0						
5				0	0				
6				0	0				
7				0	0				
8	0	0	0			0	0	0	
9	0	0	0			0	0	0	



Step 2: Distinguishable By Induction

- For each unmarked pair (r, s), if $\delta(r,t) = p$ and $\delta(s,t) = q$ for some $t \in \Sigma$, and (p, q) is marked, then mark (r, s)

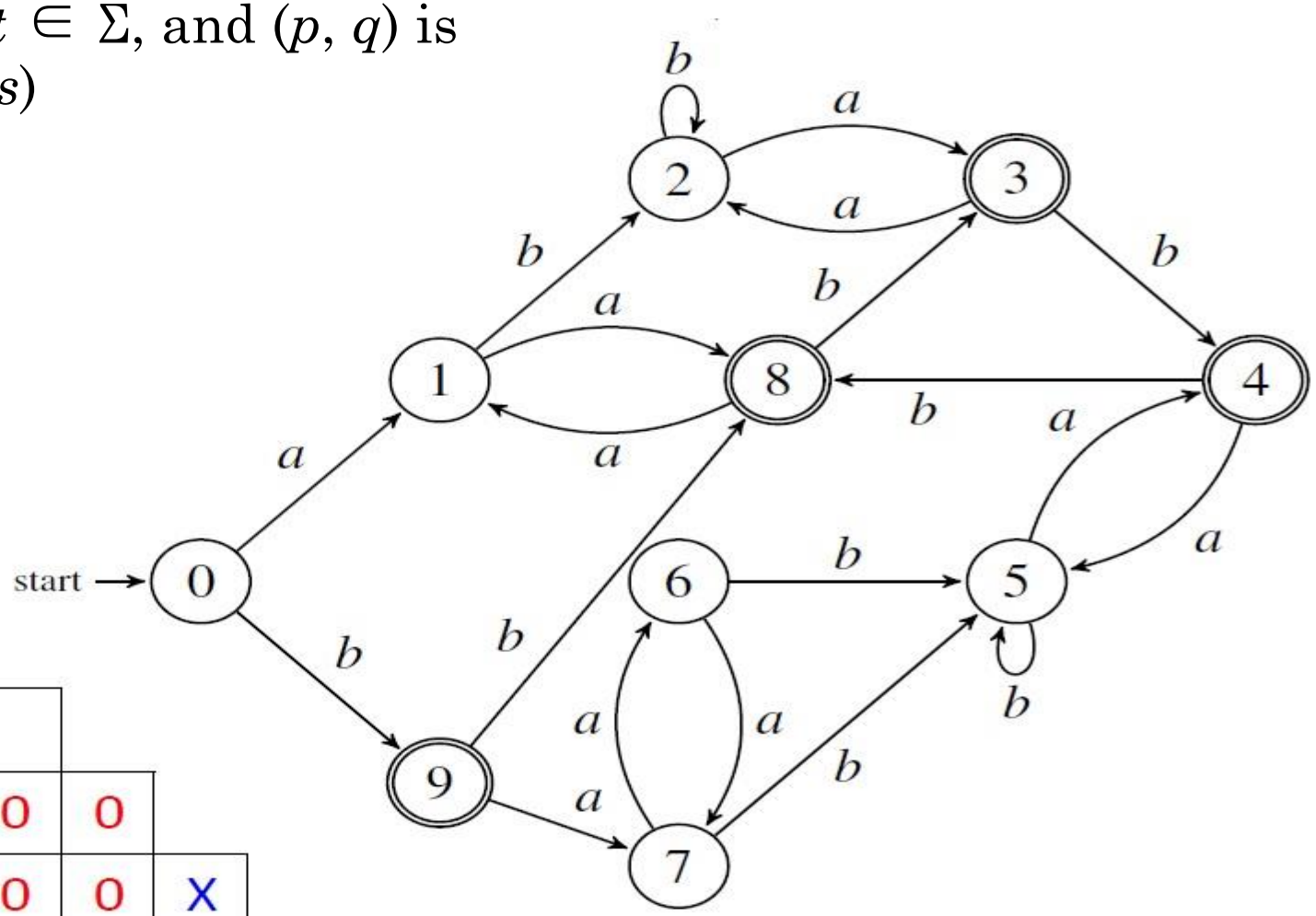
	0	1	2	3	4	5	6	7	8
1									
2									
3	0	0	0						
4	0	0	0						
5				0	0				
6				0	0				
7				0	0				
8	0	0	0			0	0	0	
9	0	0	0			0	0	0	



Step 2: Distinguishable By Induction

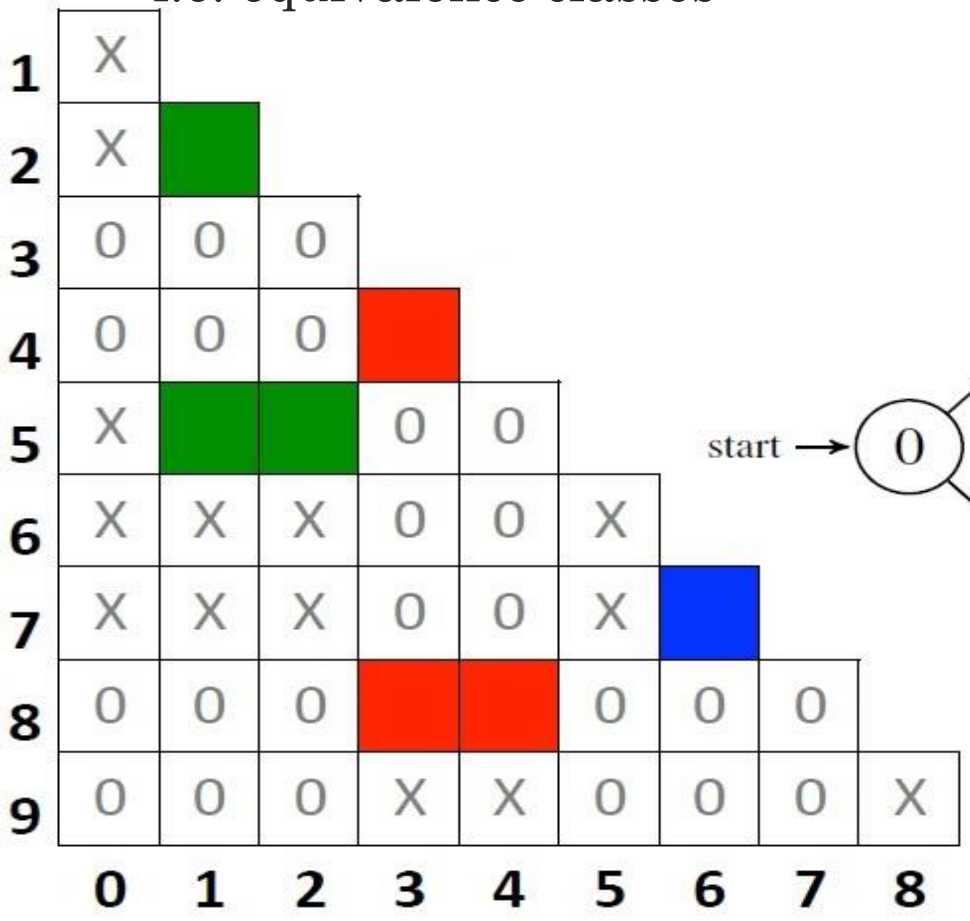
- For each unmarked pair (r, s), if $\delta(r,t) = p$ and $\delta(s,t) = q$ for some $t \in \Sigma$, and (p, q) is marked, then mark (r, s)
  - Mark "X"

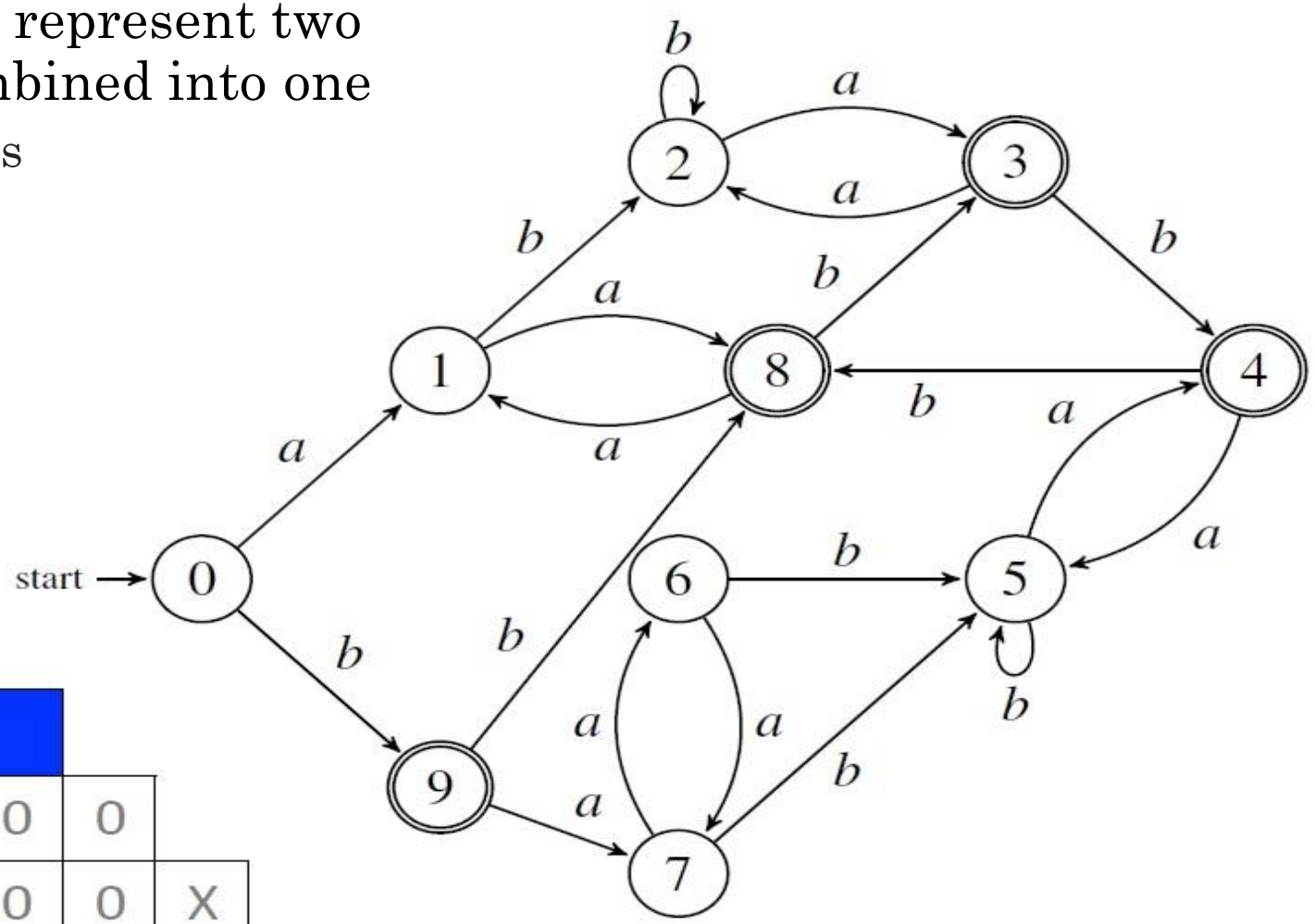
	0	1	2	3	4	5	6	7	8
1	X								
2	X								
3	0	0	0						
4	0	0	0						
5	X			0	0				
6	X	X	X	0	0	X			
7	X	X	X	0	0	X			
8	0	0	0			0	0	0	
9	0	0	0	X	X	0	0	0	X



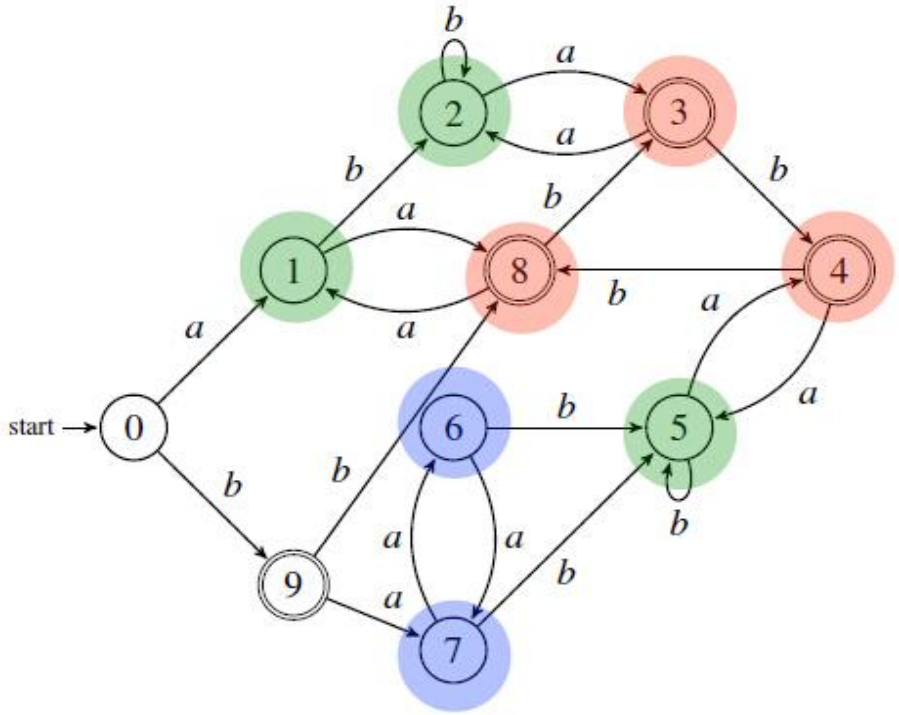
Step 3: Group The States

- When the algorithm terminates, the unmarked pairs (p, q) represent two states that can be combined into one
  - i.e. equivalence classes



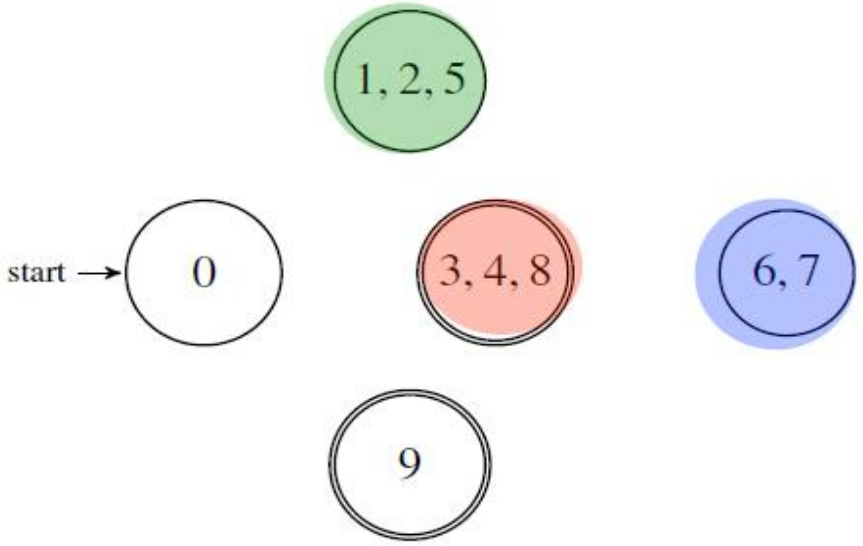


Equivalent States



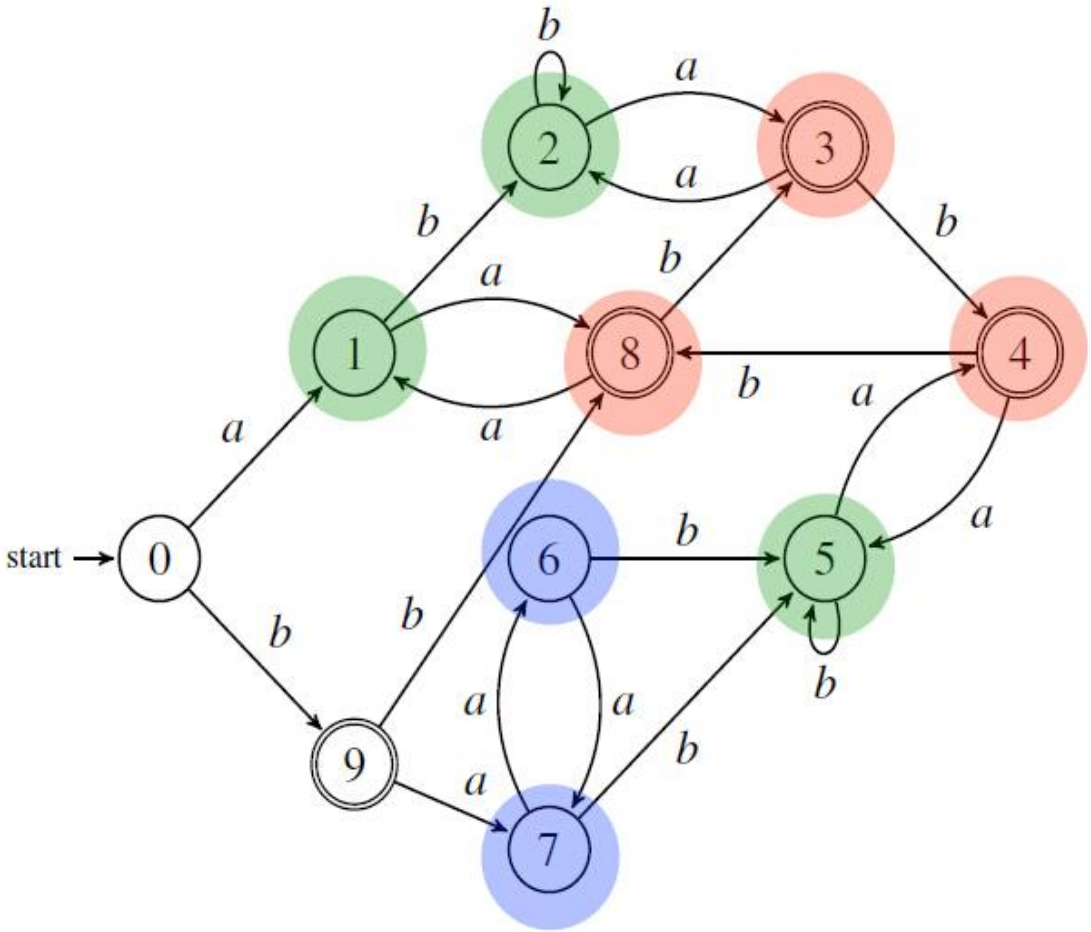
Equivalent states:
(1, 2), (1, 5), (2, 5); [non accepting]
(3, 4), (3, 8), (4, 8); [accepting]
(6, 7); non-accepting

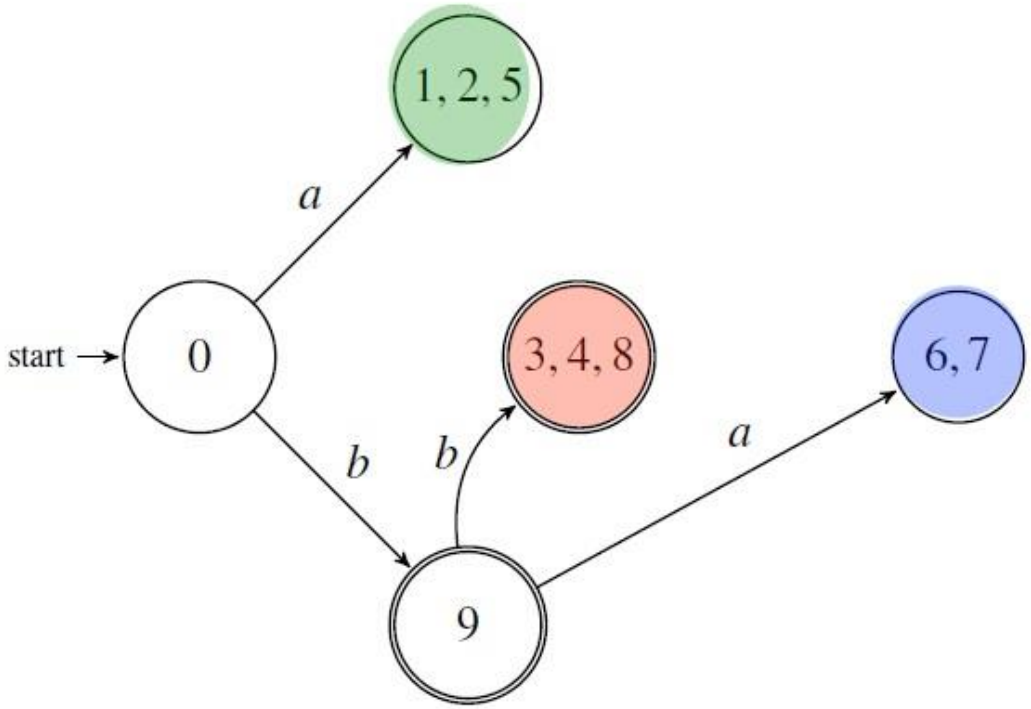
New states



Joining Up the States

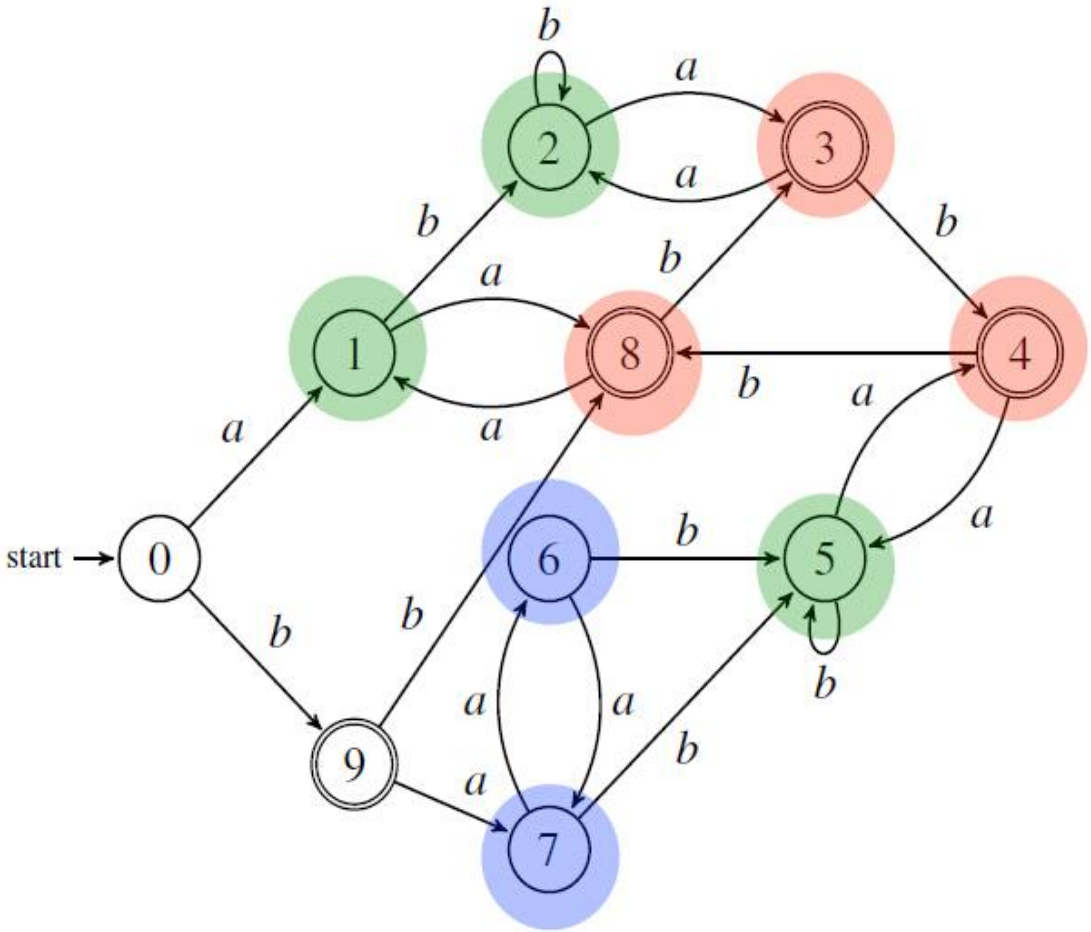
- First, draw transitions from unchanged states

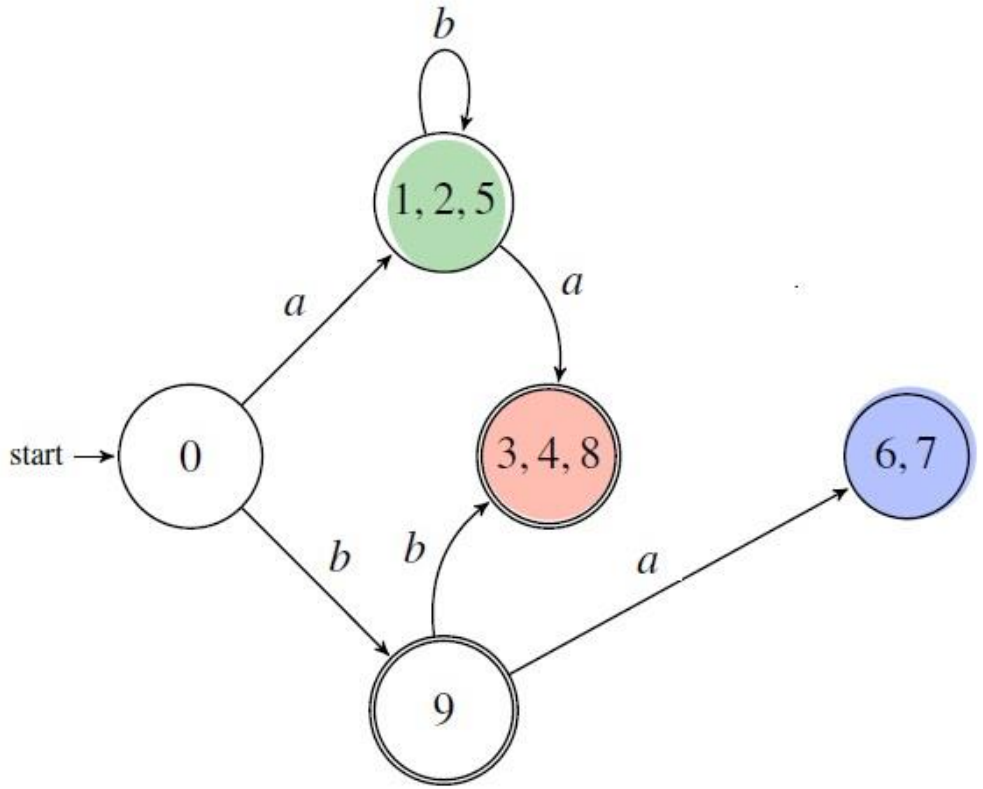




Joining Up the States

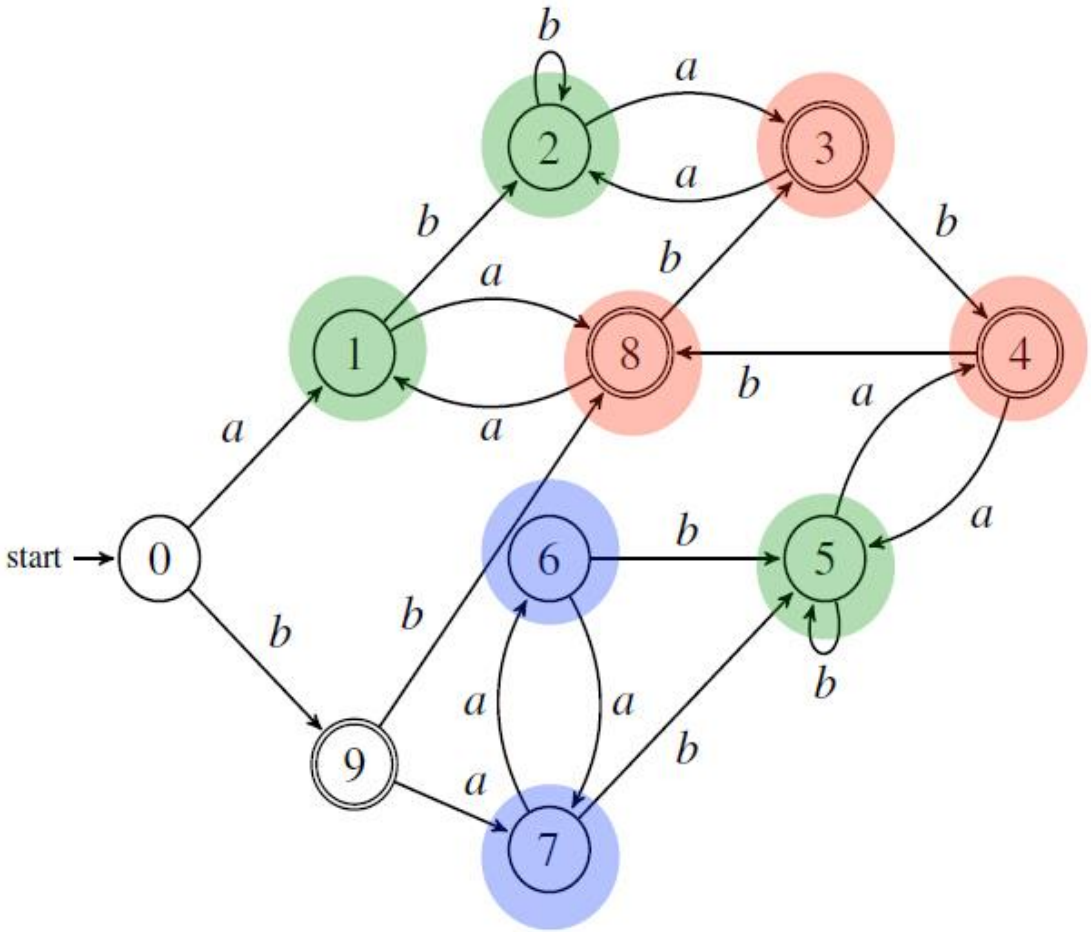
- Then, for each merged state, draw new transitions (1,2,5)

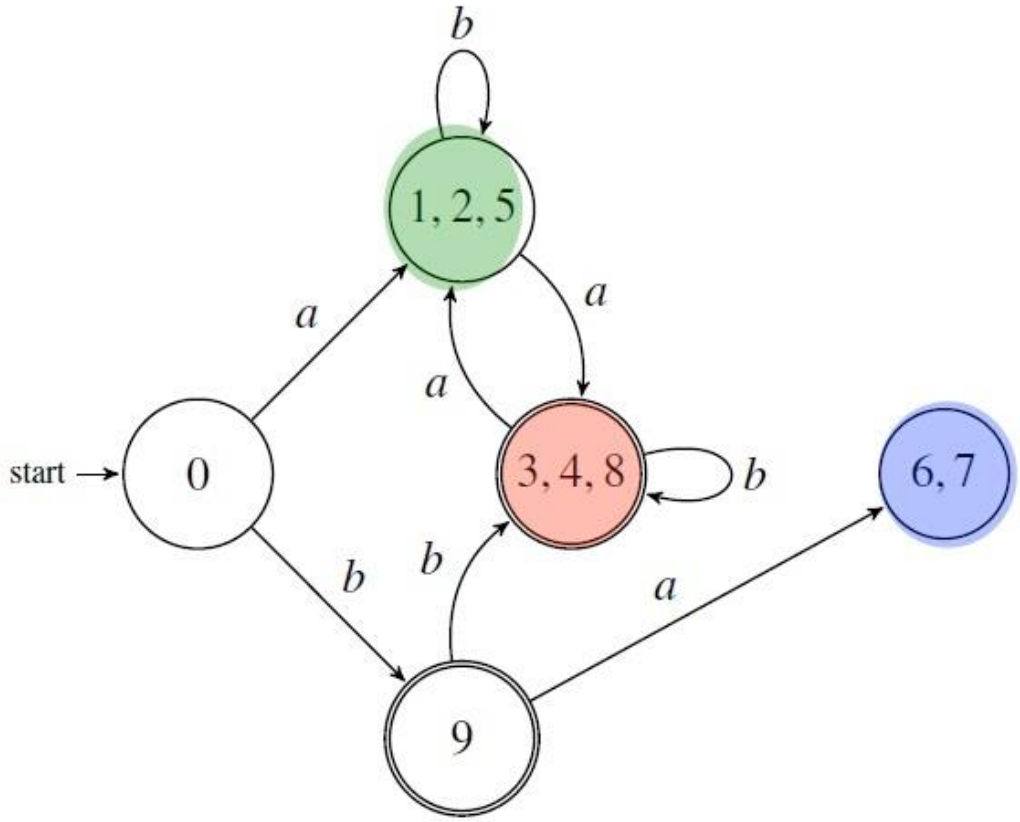




Joining Up the States

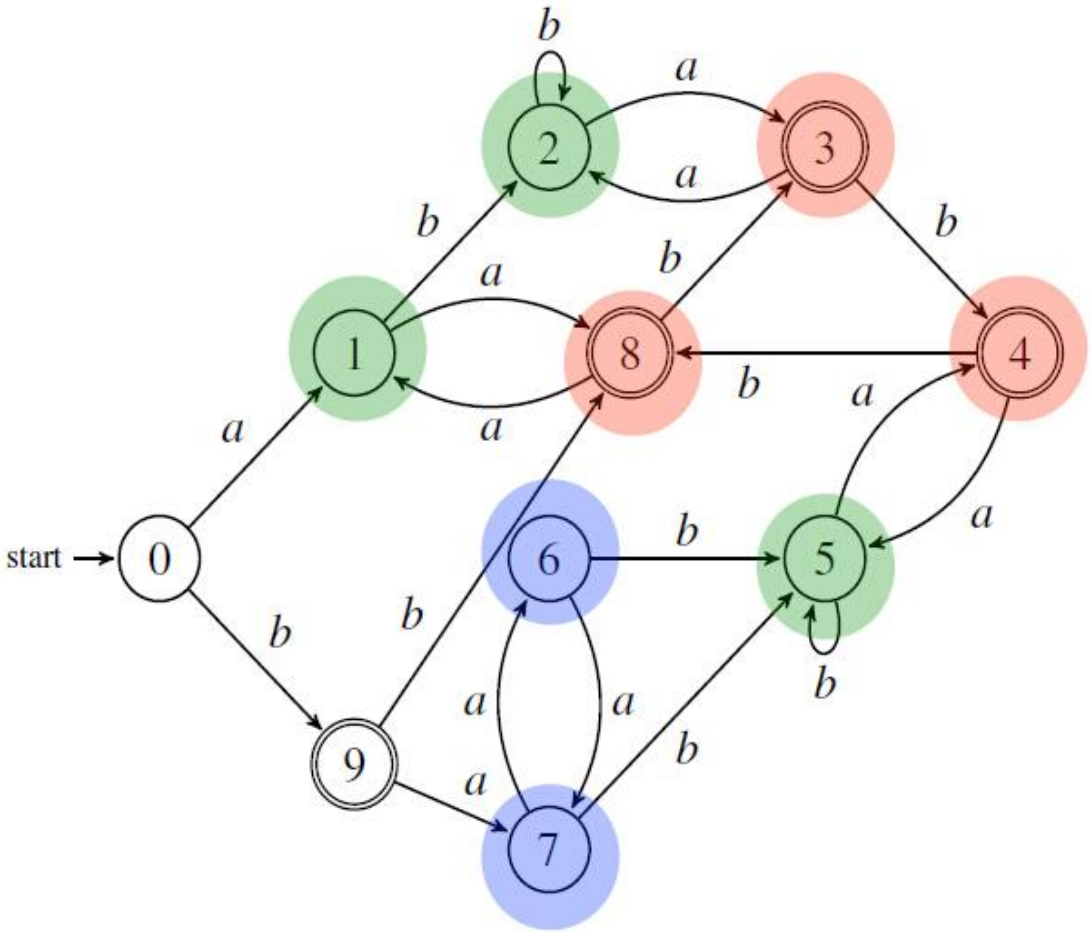
- Then, for each merged state, draw new transitions (3,4,8)

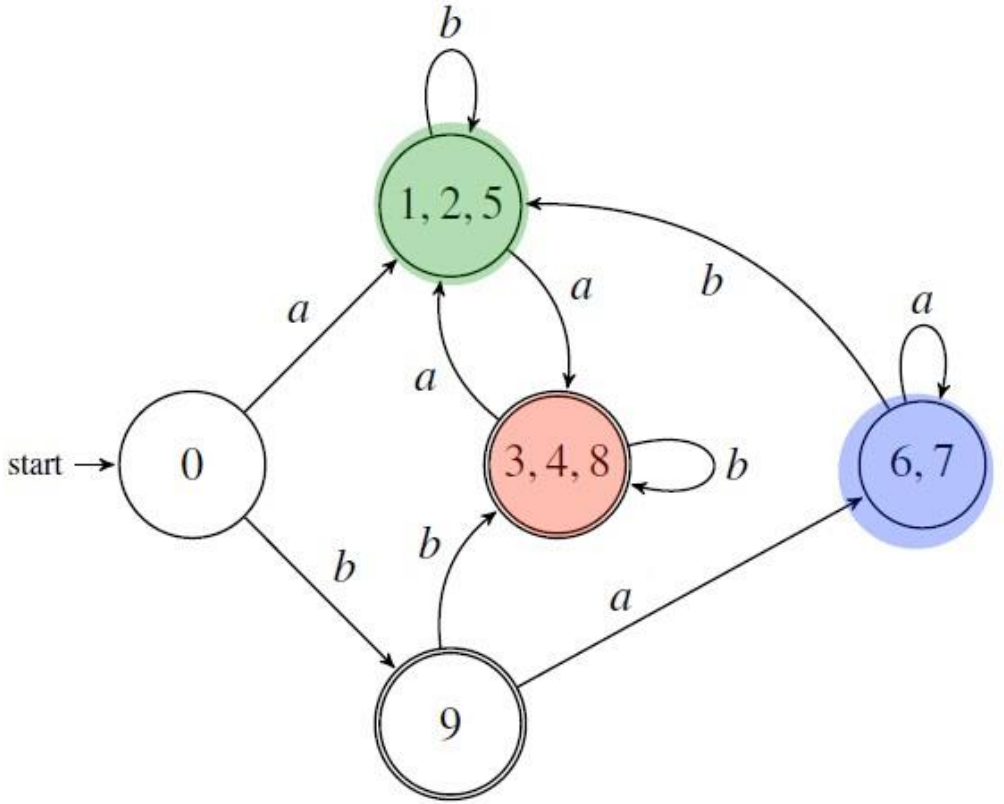




Joining Up the States

- Then, for each merged state, draw new transitions (6,7)





Concluding Remarks

- Deterministic finite automata
  - The language of a DFA
  - Accepting the union/intersection/difference of two Languages
- Language distinguishable
  - Pairwise distinguishable
- The Pumping Lemma
- Equivalence classes of DFA
- Minimizing the number of states in DFA