AUTOMATA THEORY

Exam materials

Author
Automata guy (AKA. Graph enjoyer)

Contents

1 Basic definitions

1.1 String

A string is a finite sequence, possibly empty, of symbols drawn from some alphabet Σ. Given any alphabet Σ, the shortest string that can be formed from Σ is the empty string, which we will write as ϵ. The set of all possible strings over an alphabet Σ is written Σ^*. This notation exploits the Kleene star operator, which we will define more generally below.

1.2 Language

A language is a (finite or infinite) set of strings over a finite alphabet Σ. When we are talking about more than one language, we will use the notation Σ_L to mean the alphabet from which the strings in the language L are formed.

1.3 Alphabet

An alphabet, denoted as Σ, is a non-empty finite set of symbols. These symbols are the basic units used to construct strings. For example, $\Sigma = \{a, b, c\}$ is an alphabet consisting of the symbols 'a', 'b', and 'c'.

1.4 Kleene Star

The Kleene star, denoted as Σ^*, is the set of all strings of finite length that can be constructed from the alphabet Σ. This includes the empty string ϵ, as well as all possible concatenations of symbols from Σ.

1.5 Concatenation

Concatenation is an operation that links two strings end to end. For strings x and y, the concatenation is denoted xy. If L_1 and L_2 are languages, their concatenation is $L_1L_2 = \{xy | x \in L_1 \text{ and } y \in L_2\}$.

1.6 Substrings, Prefixes, and Suffixes

A substring of a string is any sequence of consecutive symbols from that string. A prefix is a substring that includes the first symbol of the string, and a suffix is a substring that includes the last symbol of the string. For a string s, a substring t is such that $s = xty$ for some strings x and y.

1.7 Reversal

The reversal of a string s, denoted as s^R, is a string formed by writing the symbols of s in reverse order. For example, if $s = "abc"$, then $s^R = "cba"$.

1.8 Language Operations

The union of languages L_1 and L_2, denoted by $L_1 \cup L_2$, is the set of strings that are in either L_1 or L_2 or both. The intersection, denoted by $L_1 \cap L_2$, is the set of strings that are in both L_1 and L_2. The difference, denoted by $L_1 - L_2$, is the set of strings that are in L_1 but not in L_2. The complement of a language L, denoted by $\overline{L}$, is the set of all strings over Σ that are not in L.

1.9 Deterministic and Nondeterministic Machines

Deterministic machines are computational models where for every state and input, there is exactly one transition to a subsequent state. Nondeterministic machines can have multiple possible transitions for a state and input pair.

1.10 Transition Function

The transition function of an automaton is a mapping from the cross product of the state set and the input alphabet (including ϵ for nondeterministic automata) to the power set of the state set. It defines the state transitions of the automaton.

1.11 Acceptance of a String

A string s over an alphabet Σ is said to be accepted by an automaton if, starting from the initial state and processing the symbols of s, the automaton enters a final or accepting state after reading the entire string.

1.12 Finite Automata

Finite automata are abstract computational models that consist of states, transitions between these states, and rules for starting and accepting inputs. They are used to recognize regular languages.

2 Finite state machines (FSM)

2.1 Acceptor

Formally, a deterministic FSM (or DFSM) M is a quintuple $(K, \Sigma, , s, A)$, where:

- K is a finite set of states,
- Σ is the input alphabet,
- $s \in K$ is the start state,
- $A \subseteq K$ is the set of accepting states, and
- σ is the transition function. It maps from:

$$K \times \Sigma \to K$$

A configuration of a DFSM M is an element of $K \times \Sigma^*$. Think of it as a snapshot of M. It captures the two things that can make a difference to M's future behavior:

- its current state
- the input that is still left to read.

M halts whenever it enters either an accepting or a rejecting configuration. It will do so immediately after reading the last character of its input.

The language accepted by M, denoted $L(M)$, is the set of all strings accepted by M.

2.2 Equivalence of states

1. Acceptor states q and q' are not equivalent if exactly one of they have an accepting;

2. Transformer states q and q' are not equivalent if any inputs for letter x print different letters $y \neq y'$;

3. The states q and q' of any automaton are not equivalent if any for x leads to nonequivalent states $f(q, x) \neq f(q', x)$;

4. Otherwise, states q and q' are equivalent: $q \neq q'$.

2.3 Building DFA from NFA

The algorithm will be shown with an example. We will need the *eps* function.

$eps(q : state) =$

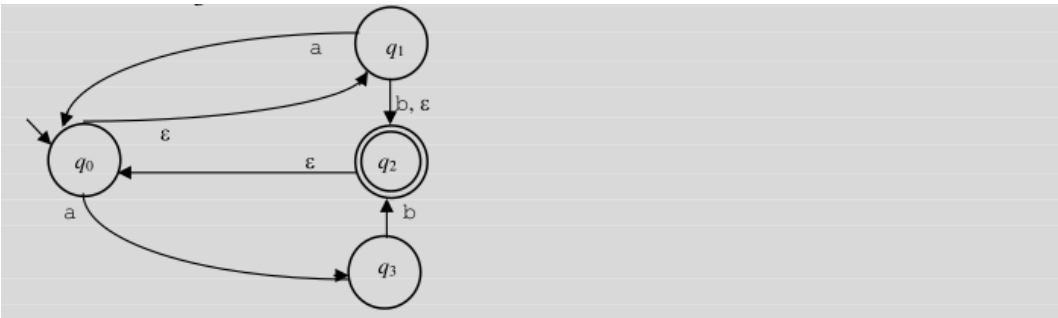
1. $result = q$.

2. While there exists some $p \in result$ and some $r \in result$ and some transition $(p, \epsilon, r) \in \delta$ do: Insert r into result.

3. Return $result$.

2.3.1 Example: Computing *eps*

Consider the following NDFSM M:



To compute $eps(q_0)$, we initially set result to $\{q_0\}$. Then q_1 is added, producing $\{q_0, q_1\}$. Then q_2 is added, producing $\{q_0, q_1, q_2\}$. There is an ϵ-transition from q_2 to q_0, but q_0 is already in result. So the computation of $e(q_0)$ halts.

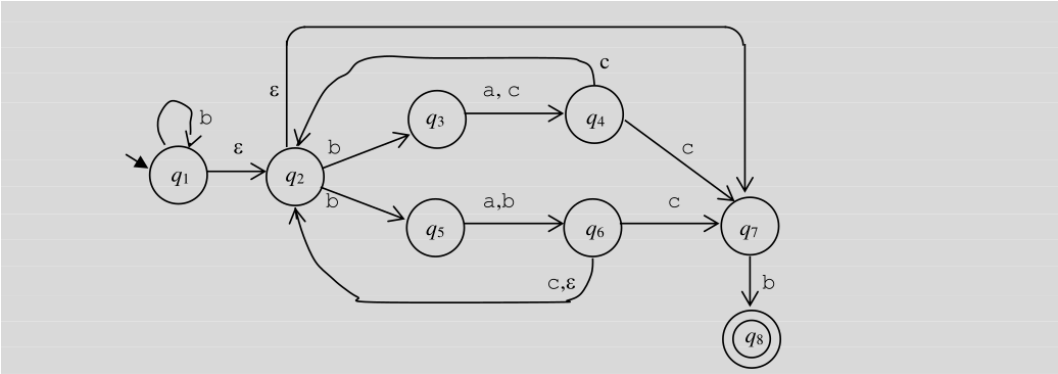
The result of running eps on each of the states of M is:

$$eps(q_0) = \{q_0, q_1, q_2\}.$$
$$eps(q_1) = \{q_0, q_1, q_2\}.$$
$$eps(q_2) = \{q_0, q_1, q_2\}.$$
$$eps(q_3) = \{q_3\}.$$

2.3.2 Example: Building DFA from NFA

Consider the following NDFSM M:

First, to get a feel for M, simulate it on the input string bbbacb, using coins to keep track of the states it enters.



We can apply `ndfsmtodfsm` to M as follows:

1. Compute $\varepsilon(q)$ for each state q in K_M: $\varepsilon(q_1) = \{q_1, q_2, q_7\}$, $\varepsilon(q_2) = \{q_2, q_7\}$, $\varepsilon(q_3) = \{q_3\}, \varepsilon(q_4) = \{q_4\}$, $\varepsilon(q_5) = \{q_5\}$, $\varepsilon(q_6) = \{q_2, q_6, q_7\}$, $\varepsilon(q_7) = \{q_7\}, \varepsilon(q_8) = \{q_8\}$.

2. $s' = \varepsilon(s) = \{q_1, q_2, q_7\}$.

3. Compute δ': active-states $= \{\{q_1, q_2, q_7\}\}$.

Consider $\{q_1, q_2, q_7\} : ((\{q_1, q_2, q_7\}, a), \emptyset)$. $((\{q_1, q_2, q_7\}, b), \{q_1, q_2, q_3, q_5, q_7, q_8\})$. $((\{q_1, q_2, q_7\}, c), \emptyset)$.

active-states $= \{\{q_1, q_2, q_7\}, \emptyset, \{q_1, q_2, q_3, q_5, q_7, q_8\}\}$.

Consider $\emptyset$: $((\emptyset, a), \emptyset)$. / $*$ $\emptyset$ is a dead state and we will generally omit it.
$((\emptyset, b), \emptyset)$. $((\emptyset, c), \emptyset)$.

active-states $= \{\{q_1, q_2, q_7\}, \emptyset,$
$\{q_1, q_2, q_3, q_5, q_7, q_8\}, \{q_2, q_4, q_6, q_7\}, \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \{q_4\}\}$.

Consider $\{q_2, q_4, q_6, q_7\} : ((\{q_2, q_4, q_6, q_7\}, a), \emptyset)$. $((\{q_2, q_4, q_6, q_7\}, b), \{q_3, q_5, q_8\})$.
$((\{q_2, q_4, q_6, q_7\}, c), \{q_2, q_7\})$.

active-states $= \{\{q_1, q_2, q_7\}, \emptyset,$
$\{q_1, q_2, q_3, q_5, q_7, q_8\}, \{q_2, q_4, q_6, q_7\}, \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \{q_4\}, \{q_3, q_5, q_8\}, \{q_2, q_7\}\}$.

Consider $\{q_1, q_2, q_3, q_5, q_6, q_7, q_8\} : ((\{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, a), \{q_2, q_4, q_6, q_7\})$.
$((\{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, b), \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\})$. $((\{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, c), \{q_2, q_4, q_7\})$.

active-states $= \{\{q_1, q_2, q_7\}, \emptyset,$
$\{q_1, q_2, q_3, q_5, q_7, q_8\}, \{q_2, q_4, q_6, q_7\}, \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \{q_4\}, \{q_3, q_5, q_8\}, \{q_2, q_7\}, \{q_2, q_4, q_7\}\}$.

Consider $\{q_4\} : ((\{q_4\}, a), \emptyset)$. $((\{q_4\}, b), \emptyset)$. $((\{q_4\}, c), \{q_2, q_7\})$.

active-states did not change.

Consider $\{q_3, q_5, q_8\} : ((\{q_3, q_5, q_8\}, a), \{q_2, q_4, q_6, q_7\})$. $((\{q_3, q_5, q_8\}, b), \{q_2, q_6, q_7\})$.
$((\{q_3, q_5, q_8\}, c), \{q_4\})$.

active-states $= \{\{q_1, q_2, q_7\}, \emptyset,$
$\{q_1, q_2, q_3, q_5, q_7, q_8\}, \{q_2, q_4, q_6, q_7\},$
$\{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \{q_4\}, \{q_3, q_5, q_8\}, \{q_2, q_7\}, \{q_2, q_4, q_7\}, \{q_2, q_6, q_7\}\}$.

Consider $\{q_2, q_7\} : ((\{q_2, q_7\}, a), \emptyset)$. $((\{q_2, q_7\}, b), \{q_3, q_5, q_8\})$. $((\{q_2, q_7\}, c), \emptyset)$.

active-states did not change.

Consider $\{q_2, q_4, q_7\} : ((\{q_2, q_4, q_7\}, a), \emptyset)$. $((\{q_2, q_4, q_7\}, b), \{q_3, q_5, q_8\})$. $((\{q_2, q_4, q_7\}, c), \{q_2, q_7\})$.

active-states did not change.

Consider $\{q_2, q_6, q_7\} : ((\{q_2, q_6, q_7\}, a), \emptyset)$. $((\{q_2, q_6, q_7\}, b), \{q_3, q_5, q_8\})$. $((\{q_2, q_6, q_7\}, c), \{q_2, q_7\})$.

active-states did not change. δ' has been computed for each element of active-states.

2.4 Transducer (Transformator)

This definition for a deterministic finite state transducer permits each machine to output any finite sequence of symbols as it makes each transition (in other words, as it reads each symbol of its input). FSMs that associate outputs with transitions are called Mealy machines, after their inventor George Mealy. A Mealy machine M is a six-tuple $(K, \Sigma, O, \delta, s, A)$, where:

- K is a finite set of states,
- Σ is an input alphabet,
- O is an output alphabet,
- $s \in K$ is the start state,
- $A \subseteq K$ is the set of accepting states (although for some applications this designation is not important),
- σ is the transition function. It is function $(K \times \Sigma) \to (K)$, and
- D is the display or output function. It is a function from $(K) \to (O^*)$.

A Mealy (transducer defined above) machine M computes a function $f(w)$ iff, when it reads the input string w, its output sequence is $f(w)$.

2.4.1 Note on determinism

Transducers are a class of automata used to model systems that transform input sequences into output sequences. Moore and Mealy machines are two types of transducers commonly studied in automata theory.

Both Moore and Mealy machines are inherently deterministic. In a deterministic transducer:

- Each state transition is uniquely determined by the current state and input symbol. The transition function δ of a Moore or Mealy machine maps each pair of current state and input symbol to exactly one next state.
- In the case of Moore machines, the output function λ maps each state to exactly one output symbol, whereas in Mealy machines, the output

function ω maps each pair of current state and input symbol to exactly one output symbol.

Determinism ensures that for any given input sequence, the transducer will always produce the same output sequence and end in the same final state, providing predictability and repeatability, which are essential properties for many practical applications. The concept of nondeterminism can be extended to transducers; however, this is not standard in classical automata.

2.5 Probabalistic automata

Probabalistic automata is defined by 6-tuple $(Q, X, p, q_A, q_0, \lambda)$:

- Q – set of states
- X – input alphabet
- p – transition function: $Q \times X \times Q[0;1]$ or $X \to [0;1]^{|Q|\times|Q|}$
- $Q_A \subseteq Q$ – accepting states
- q_0 – initial state (sometimes $q0 \in [0;1]^{|Q|}$)
- $\lambda \in [0;1]$ – acceptance threshold (sometimes there is no λ)

2.5.1 Finding Accepting Words in Probabilistic Automata

To determine whether a word is accepted by a probabilistic automaton, we compute the probability of reaching an accepting state after processing the word. The process involves two key steps:

1. For each word in the input alphabet, calculate the probability of transitioning from one state to another. This is done by summing up the probabilities of all possible paths that lead to an accepting state after processing the word. The transition function p gives the probability of transitioning from one state to another given an input symbol.

2. Compare the calculated probability with the acceptance threshold λ. If the cumulative probability of reaching an accepting state is greater than or equal to λ, the word is accepted; otherwise, it is rejected.

2.5.2 Using Matrix Multiplication for Each Input

Matrix multiplication can be used to efficiently compute the probabilities of transitioning between states for a given input string. Consider the transition matrix $P(x)$ for an input symbol $x \in X$, where each entry $P(x)_{ij}$ represents the probability of transitioning from state i to state j given the input x. The process is as follows:

1. Start with a row vector $\mathbf{v}$ representing the initial state distribution. If the initial state is certain, $\mathbf{v}$ is a one-hot vector with a 1 at the position corresponding to the initial state q_0.

2. For each symbol x in the input string, multiply the current state vector $\mathbf{v}$ by the transition matrix $P(x)$. This results in a new state vector representing the probability distribution over states after processing x.

$$\mathbf{v}' = \mathbf{v} \cdot P(x)$$

3. After processing the entire input string, the final state vector $\mathbf{v}'$ gives the probability distribution over all states. The probability of being in an accepting state is the sum of the probabilities in $\mathbf{v}'$ corresponding to the accepting states.

2.6 Pumping lemma for Regular Languages

Theorem: If L is a regular language, then:

$$\begin{aligned}\exists k \geq 1 (\forall \text{ strings } w \in L, where |w| \geq k (x, y, z (w = & xyz, \\ & |xy| \leq k, \\ & y \neq \epsilon, \text{and} \\ & \forall q \geq 0 (xy^q z \in L)))).\end{aligned}$$

The Pumping Theorem tells us something that is true of every regular language. Generally, if we already know that a language is regular, we won't particularly care about what the Pumping Theorem tells us about it. But

suppose that we are interested in some language L and we want to know whether or not it is regular. If we could show that the claims made in the Pumping Theorem are not true of L, then we would know that L is not regular. It is in arguments such as this that we will find the Pumping Theorem very useful. In particular, we will use it to construct proofs by contradiction. We will say, "If L were regular, then it would possess certain properties. But it does not possess those properties. Therefore, it is not regular."

To make maximum use of the Pumping Theorem's requirement that y fall in the first k characters, it is often a good idea to choose a string w that is substantially longer than the k characters required by the theorem. In particular, if w can be chosen so that there is a uniform first region of length at least k, it may be possible to consider just a single case for where y can fall. The Pumping Theorem inspires poets, as we'll see in Chapter 10. A^nB^n is a simple language that illustrates the kind of property that characterizes languages that aren't regular. It isn't of much practical importance, but it is typical of a family of languages, many of which are of more practical significance. In the next example, we consider Bal, the language of balanced parentheses. The structure of Bal is very similar to that of A^nB^n. Bal is important because most languages for describing arithmetic expressions, Boolean queries, and markup systems require balanced delimiters.

2.6.1 Example: Even Palindrome Language is Not Regular

The Even Palindrome Language is Not Regular Let L be $PalEven = \{ww : w \in \{a, b\}^*\}$. $PalEven$ is the language of even-length palindromes of a's and b's. We can use the Pumping Theorem to show that $PalEven$ is not regular. If it were, then there would exist some k such that any string w, where $|w| \geq k$, must satisfy the conditions of the theorem. We show one string w that does not. (Note here that the variable w used in the definition of L is different from the variable w mentioned in the Pumping Theorem.) We will choose w so that we only have to consider one case for where y could fall. Let $w = a^kb^kb^ka^k$. Since $|w| = 4k$ and w is in L, w must satisfy the conditions of the Pumping Theorem. So there must exist x, y, and z, such that $w = xyz$, $|xy| \leq k$, $y \neq \epsilon$, and $\forall q \geq 0(xy^qz \in L)$. Since $|xy| \leq k$, y must occur within the first k characters and so $y = a^p$ for some p. Since $y \neq \epsilon$, p must be greater than 0. Let $q = 2$. The resulting string is $a^{k+p}b^kb^ka^k$. If p is odd, then this string is not in $PalEven$ because all strings in $PalEven$ have even length. If p is even

then it is at least 2, so the first half of the string has more a's than the second half does, so it is not in $PalEven$. So $L = PalEven$ is not regular.

2.6.2 Example: A^nB^n Language is Not Regular

A^nB^n is not Regular. Let L be $A^nB^n = \{a^nb^n : n \neq 0\}$. We can use the Pumping Theorem to show that L is not regular. If it were, then there would exist some k such that any string w, where $|w| \geq k$, must satisfy the conditions of the theorem. We show one string w that does not. Let $w = a^kb^k$. Since $|w| = 2k$, and w is in L, it must satisfy the conditions of the Pumping Theorem. Therefore, there must exist x, y, and z such that $w = xyz$, $|xy| \leq k$, $y \neq \epsilon$, and $\forall q \geq 0(xy^qz \in L)$. Since $|xy| \leq k$, y must occur within the first k characters and therefore $y = a^p$ for some p. Since $y \neq \epsilon$, p must be greater than 0. Let $q = 2$. The resulting string is $a^{k+p}b^k$. The last condition of the Pumping Theorem states that this string must be in L, but it is not, since it has more a's than b's. Thus, there exists at least one long string in L that fails to satisfy the conditions of the Pumping Theorem. Therefore, $L = A^nB^n$ is not regular.

3 Regex (Regular expressions)

A regular expression over an alphabet Σ is a formal way of representing a regular language. It is constructed from members of Σ using union (denoted by +), concatenation, Kleene star, and parentheses for grouping.

A regular expression is a string that can be formed according to the following rules:

1. $\emptyset$ is a regular expression.
2. ϵ is a regular expression.
3. Every element in Σ is a regular expression.
4. Given two regular expressions α and β, $\alpha\beta$ is a regular expression.
5. Given two regular expressions α and β, $\alpha \vee \beta$ is a regular expression.
6. Given a regular expression α, α^* is a regular expression.
7. Given a regular expression α, α^+ is a regular expression.

8. Given a regular expression α, (α) is a regular expression.

Define the following semantic interpretation function L for the language of regular expressions:

1. $L(\emptyset) = \emptyset$, the language that contains no strings.

2. $L(\epsilon) = \{\epsilon\}$, the language that contains just the empty string.

3. For any $c \in \Sigma$, $L(c) = \{c\}$, the language that contains the single, one-character string c.

4. For any regular expressions α and β, $L(\alpha\beta) = L(\alpha)L(\beta)$. In other words, to form the meaning of the concatenation of two regular expressions, first determine the meaning of each of the constituents. Both meanings will be languages. Then concatenate the two languages together. Recall that the concatenation of two languages L_1 and L_2 is $\{w = xy \mid x \in L_1 \text{ and } y \in L_2\}$. Note that, if either $L(\alpha)$ or $L(\beta)$ is equal to $\emptyset$, then the concatenation will also be equal to $\emptyset$.

5. For any regular expressions α and β, $L(\alpha \cup \beta) = L(\alpha) \cup L(\beta)$. Again we form the meaning of the larger expression by first determining the meaning of each of the constituents. Each of them is a language. The meaning of $\alpha \cup \beta$ then, as suggested by our choice of the character $\cup$ as an operator, is the union of the two constituent languages.

6. For any regular expression α, $L(\alpha^*) = (L(\alpha))^*$, where $*$ is the Kleene star operator defined in Section 2.2.5. So $L(\alpha^*)$ is the language that is formed by concatenating together zero or more strings drawn from $L(\alpha)$.

7. For any regular expression α, $L(\alpha^+) = L(\alpha\alpha^*) = L(\alpha)(L(\alpha))^*$. If $L(\alpha)$ is equal to $\emptyset$, then $L(\alpha^+)$ is also equal to $\emptyset$. Otherwise $L(\alpha^+)$ is the language that is formed by concatenating together one or more strings drawn from $L(\alpha)$.

8. For any regular expression α, $L((\alpha)) = L(\alpha)$. In other words, parentheses have no effect on meaning except to group the constituents in an expression.

- * asterisk indicates zero or more occurrences of the preceding element.

For example, ab*c matches "ac", "abc", "abbc", "abbbc", and so on.

- + the plus sign indicates one or more occurrences of the preceding element. For example, ab+c matches "abc", "abbc", "abbbc", and so on, but not "ac".

3.1 Building Regex from FSM

- If α is the regular expression $\beta \cup \gamma$ and if both $L(\beta)$ and $L(\gamma)$ are regular, then we construct $M_3 = (K_3, \Sigma, \Delta_3, s_3, A_3)$ such that $L(M_3) = L(\alpha) = L(\beta) \cup L(\gamma)$. If necessary, rename the states of M_1 and M_2 so that $K_1 K_2 =$. Create a new start state, s_3, and connect it to the start states of M_1 and M_2 via ϵ-transitions. M_3 accepts if either M_1 or M_2 accepts. So $M_3 = (s_3 \cup K_1 \cup K_2, \Sigma, \Delta_3, s_3, A_1 \cup A_2)$, where $\Delta 3 = \Delta 1 \cup \Delta 2 \cup ((s_3, \epsilon), s_1), ((s_3, \epsilon), s_2)$.

- If α is the regular expression $\beta\gamma$ and if both $L(\beta)$ and $L(\gamma)$ are regular, then we construct $M_3 = (K_3, \Sigma, \Delta_3, s_3, A_3)$ such that $L(M_3) = L(\alpha) = L(\beta)L(\gamma)$. If necessary, rename the states of M_1 and M_2 so that $K_1 \cap K_2 = \emptyset$. We will build M_3 by connecting every accepting state of M_1 to the start state of M_2 via an ϵ-transition. M_3 will start in the start state of M_1 and will accept if M_2 does. So $M_3 = (K_1 \cup K_2, \Sigma, \Delta_3, s_1, A_2)$, where $\Delta_3 = \Delta_1 \cup \Delta_2 \cup \{((q, \epsilon), s_2) : q \in A_1\}$.

- If α is the regular expression β^* and if $L(\beta)$ is regular, then we construct $M_2 = (K_2, \Sigma, \Delta_2, s_2, A_2)$ such that $L(M_2) = L(\alpha) = L(\beta)^*$. We will create a new start state s_2 and make it accepting, thus assuring that M_2 accepts ϵ. (We need a new start state because it is possible that s_1, the start state of M_1, is not an accepting state. If it isn't and if it is reachable via any input string other than ϵ, then simply making it an accepting state would cause M_2 to accept strings that are not in $(L(M_1))^*$.) We link the new s_2 to s_1 via an ϵ-transition. Finally, we create ϵ-transitions from each of M_1's accepting states back to s_1. So $M_2 = (\{s_2\} \cup K_1, \Sigma, \Delta_2, s_2, \{s_2\} \cup A_1)$, where $\Delta_2 = \Delta_1 \cup \{((s_2, \epsilon), s_1)\} \cup \{((q, \epsilon), s_1) : q \in A_1\}$.

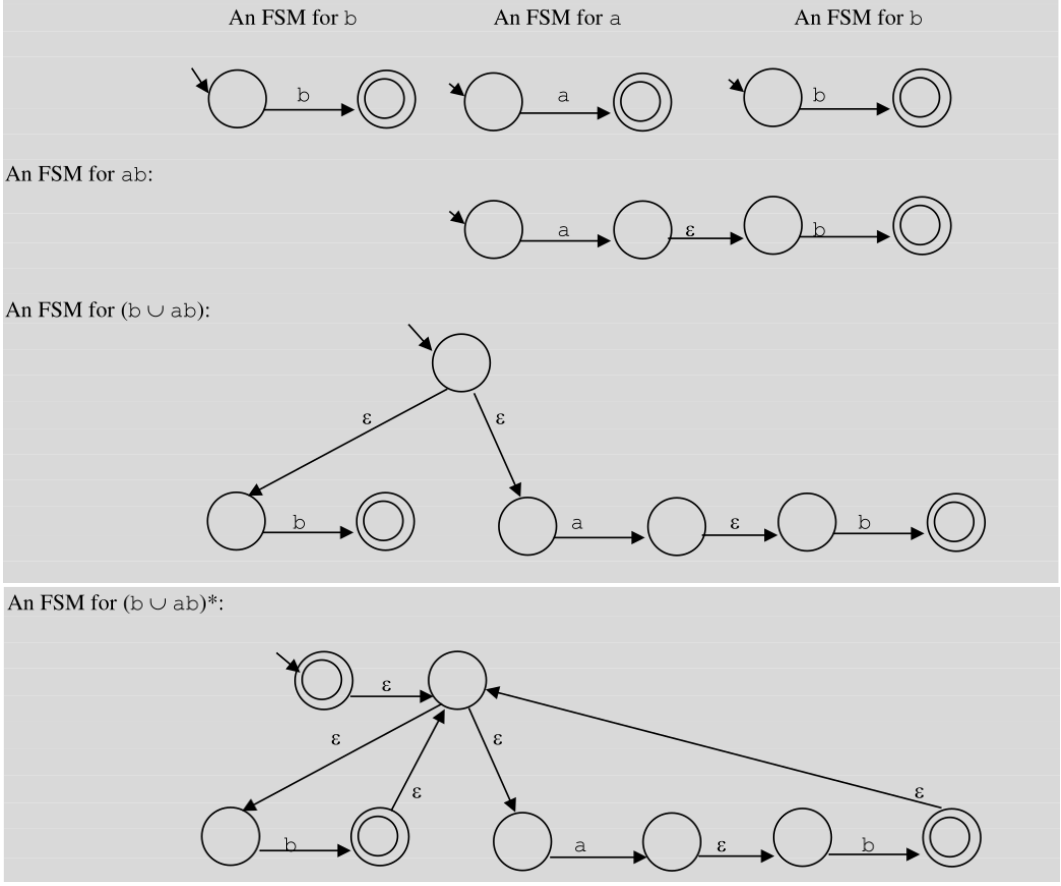
Notice that the machines that these constructions build are typically highly nondeterministic because of their use of ϵ-transitions. They also typically have a large number of unnecessary states. But, as a practical matter, that is not

a problem since, given an arbitrary NDFSM M, we have an algorithm that can construct an equivalent DFSM M'. We also have an algorithm that can minimize M'.

3.1.1 Example: Building FSM from Regex

Example 6.5 Building an FSM from a Regular Expression

Consider the regular expression (b ∪ ab)*. We use *regextofsm* to build an FSM that accepts the language defined by this regular expression:



3.2 Building Regex from FSM

Next we must show how to build FSMs to accept languages that are defined by regular expressions that exploit the operations of concatenation, union, and Kleene star. Let β and γ be regular expressions that define languages over the alphabet Σ. If $L(\beta)$ is regular, then it is accepted by some FSM $M_1 = (K_1, \Sigma, \Delta_1, s_1, A_1)$. If $L(\gamma)$ is regular, then it is accepted by some $FSMM_2 = (K_2, \Sigma, \Delta_2, s_2, A_2)$.

$fsmtoregexheuristic(M : FSM) =$

1. Remove from M any states that are unreachable from the start state.

2. If M has no accepting states then halt and return the simple regular expression $\emptyset$.

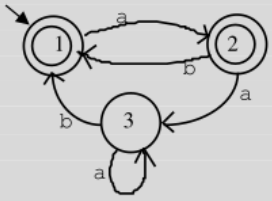
3. If the start state of M is part of a loop (i.e., it has any transitions coming into it), create a new start state s and connect s to M's start state via an ϵ-transition. This new start state s will have no transitions into it.

4. If there is more than one accepting state of M or if there is just one but there are any transitions out of it, create a new accepting state and connect each of M's accepting states to it via an ϵ-transition. Remove the old accepting states from the set of accepting states. Note that the new accepting state will have no transitions out from it.

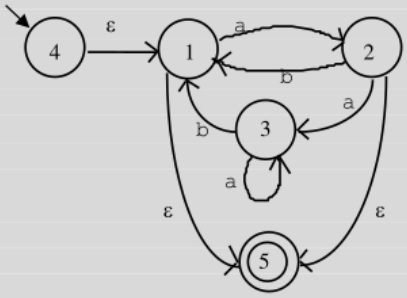
3.2.1 Example: Building Regex from FSM

Example 6.7 Building a Regular Expression from an FSM

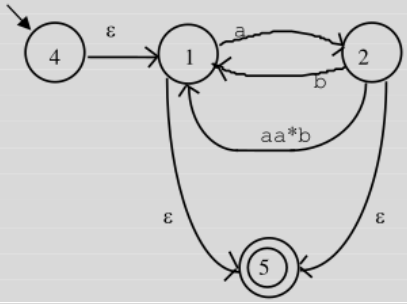
Let M be:



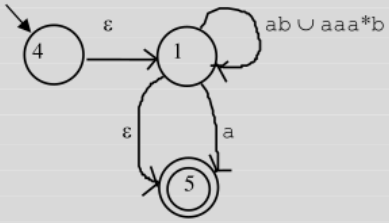
Create a new start state and a new accepting state and link them to M:



Remove state 3:



Remove state 2:



Remove state 1:

4 PDA (Pushdown automata)

4.1 Definition of a (Nondeterministic) PDA

A pushdown automaton, or PDA, is a finite state machine that has been augmented by a single stack. In a minute, we will present the formal definition of the PDA model that we will use. But, before we do that, one caveat to readers of other books is in order. There are several competing PDA definitions, from which we have chosen one to present here. All are provably equivalent, in the sense that, for all i and j, if there exists a version$i$ PDA that accepts some language L then there also exists a version$j$ PDA that accepts L. We'll return to this issue in Section 12.5, where we will mention a few of the other models and sketch an equivalence proof. For now, simply beware of the fact that other definitions are also in widespread use.

We will use the following definition: A pushdown automaton (or PDA) M is a six-tuple $(K, \Sigma, \Gamma, \Delta, s, A)$, where:

- K is a finite set of states,
- Σ is the input alphabet,
- Γ is the stack alphabet,
- $s \in K$ is the start state,
- $A \subseteq K$ is the set of accepting states, and
- Δ is the transition relation. It is a finite subset of

$$((K \times \Sigma \times \Gamma) \times (K \times \Gamma^*))$$

We will use the following notational convention for describing M's stack as a string: The top of the stack is to the left of the string.

If a $c_1c_2 \ldots c_n$ of characters is pushed onto the stack, they will be pushed rightmost first, so if the value of the stack before the push was s, the value after the push will be $c_1c_2 \ldots c_ns$.

M may only take the transition if the character c_2 matches the current top of

the stack. If it does, and the transition is taken, then M pops c_2 and then pushes γ. M cannot "peek" at the top of its stack without popping off the values that it examines.

C is an accepting computation iff $C = (s, w, \alpha)| -_M^* (q, \epsilon, \beta)$, for some $q \in A$. In our definition there are no restrictions on final stack contents. In other words, PDA accepts if it reaches the accepting configuration with no regard to the stack contents.