

Let us illustrate the subset construction on the NFA of Example 4.

A NFA for the language

$$L_3 = \{a, b\}^* \{abb\}.$$

Transition table δ_4 :

	a	b
0	$\{0, 1\}$	$\{0\}$
1	\emptyset	$\{2\}$
2	\emptyset	$\{3\}$
3	\emptyset	\emptyset

Set of accepting states: $F_4 = \{3\}$.

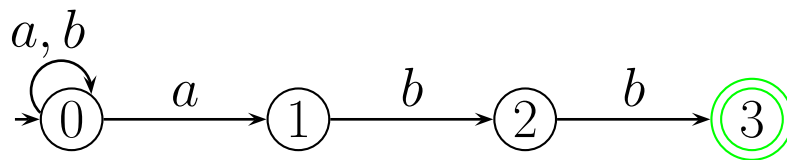


Figure 2.5: NFA for $\{a, b\}^* \{abb\}$

The pointer \Rightarrow corresponds to *marked* and the pointer \rightarrow to *total*.

Initial transition table Δ .

	names	states	<i>a</i>	<i>b</i>
\Rightarrow	<i>A</i>	{0}		
\rightarrow	<i>A</i>	{0}		

Just after entering the while loop

	names	states	<i>a</i>	<i>b</i>
$\Rightarrow \rightarrow$	<i>A</i>	{0}		

After the first round through the while loop.

	names	states	<i>a</i>	<i>b</i>
\Rightarrow	<i>A</i>	{0}	<i>B</i>	<i>A</i>
\rightarrow	<i>B</i>	{0, 1}		

After just reentering the while loop.

	names	states	<i>a</i>	<i>b</i>
$\Rightarrow \rightarrow$	<i>A</i>	{0}	<i>B</i>	<i>A</i>
	<i>B</i>	{0, 1}		

After the second round through the while loop.

	names	states	<i>a</i>	<i>b</i>
\Rightarrow	<i>A</i>	{0}	<i>B</i>	<i>A</i>
	<i>B</i>	{0, 1}	<i>B</i>	<i>C</i>
\rightarrow	<i>C</i>	{0, 2}		

After the third round through the while loop.

	names	states	a	b
	A	$\{0\}$	B	A
	B	$\{0, 1\}$	B	C
\Rightarrow	C	$\{0, 2\}$	B	D
\rightarrow	D	$\{0, 3\}$		

After the fourth round through the while loop.

	names	states	a	b
	A	$\{0\}$	B	A
	B	$\{0, 1\}$	B	C
	C	$\{0, 2\}$	B	D
$\Rightarrow \rightarrow$	D	$\{0, 3\}$	B	A

This is the DFA of Figure 2.3, except that in that example A, B, C, D are renamed $0, 1, 2, 3$

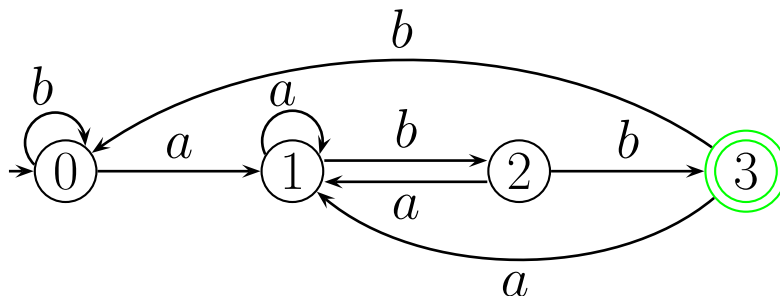


Figure 2.6: DFA for $\{a, b\}^*\{abb\}$

2.7 Finite State Automata With Output: Transducers

So far, we have only considered automata that recognize languages, i.e., automata that do not produce any output on any input (except “accept” or “reject”).

It is interesting and useful to consider input/output finite state machines. Such automata are called *transducers*. They compute functions or relations. First, we define a deterministic kind of transducer.

Definition 2.7.1 A *general sequential machine (gsm)* is a sextuple $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$, where

- (1) Q is a finite set of *states*,
- (2) Σ is a finite *input alphabet*,
- (3) Δ is a finite *output alphabet*,
- (4) $\delta: Q \times \Sigma \rightarrow Q$ is the *transition function*,
- (5) $\lambda: Q \times \Sigma \rightarrow \Delta^*$ is the *output function* and
- (6) q_0 is the *initial* (or *start*) *state*.

If $\lambda(p, a) \neq \epsilon$, for all $p \in Q$ and all $a \in \Sigma$, then M is *nonerasing*. If $\lambda(p, a) \in \Delta$ for all $p \in Q$ and all $a \in \Sigma$, we say that M is a *complete sequential machine (csm)*.

In order to define how a gsm works, we extend the transition and the output functions. We define $\delta^*: Q \times \Sigma^* \rightarrow Q$ and $\lambda^*: Q \times \Sigma^* \rightarrow \Delta^*$ recursively as follows: For all $p \in Q$, all $u \in \Sigma^*$ and all $a \in \Sigma$

$$\begin{aligned}\delta^*(p, \epsilon) &= p \\ \delta^*(p, ua) &= \delta(\delta^*(p, u), a) \\ \lambda^*(p, \epsilon) &= \epsilon \\ \lambda^*(p, ua) &= \lambda^*(p, u)\lambda(\delta^*(p, u), a).\end{aligned}$$

For any $w \in \Sigma^*$, we let

$$M(w) = \lambda^*(q_0, w)$$

and for any $L \subseteq \Sigma^*$ and $L' \subseteq \Delta^*$, let

$$M(L) = \{\lambda^*(q_0, w) \mid w \in L\}$$

and

$$M^{-1}(L') = \{w \in \Sigma^* \mid \lambda^*(q_0, w) \in L'\}.$$

Note that if M is a csm, then $|M(w)| = |w|$, for all $w \in \Sigma^*$. Also, a homomorphism is a special kind of gsm—it can be realized by a gsm with one state.

We can use gsm's and csm's to compute certain kinds of functions.

Definition 2.7.2 A function $f: \Sigma^* \rightarrow \Delta^*$ is a *gsm* (resp. *csm*) *mapping* iff there is a gsm (resp. csm), M , so that $M(w) = f(w)$, for all $w \in \Sigma^*$.

Remark: Ginsburg and Rose (1966) characterized gsm mappings as follows:

A function $f: \Sigma^* \rightarrow \Delta^*$ is a gsm mapping iff

- (a) f preserves prefixes, i.e., $f(x)$ is a prefix of $f(xy)$;
- (b) There is an integer, m , such that for all $w \in \Sigma^*$ and all $a \in \Sigma$, we have $|f(wa)| - |f(w)| \leq m$;
- (c) $f(\epsilon) = \epsilon$;
- (d) For every regular language, $R \subseteq \Delta^*$, the language $f^{-1}(R) = \{w \in \Sigma^* \mid f(w) \in R\}$ is regular.

A function $f: \Sigma^* \rightarrow \Delta^*$ is a csm mapping iff f satisfies (a) and (d) and for all $w \in \Sigma^*$, $|f(w)| = |w|$.

The following proposition is left as a homework problem.

Proposition 2.7.3 *The family of regular languages (over an alphabet Σ) is closed under both gsm and inverse gsm mappings.*

We can generalize the gsm model so that

- (1) the device is nondeterministic,
- (2) the device has a set of accepting states,
- (3) transitions are allowed to occur without new input being processed,
- (4) transitions are defined for input strings instead of individual letters.

Here is the definition of such a model, the *a-transducer*. A much more powerful model of transducer will be investigated later: the *Turing machine*.

Definition 2.7.4 An *a-transducer* (or *nondeterministic sequential transducer with accepting states*) is a sextuple $M = (K, \Sigma, \Delta, \lambda, q_0, F)$, where

- (1) K is a finite set of *states*,
- (2) Σ is a finite *input alphabet*,
- (3) Δ is a finite *output alphabet*,
- (4) $q_0 \in K$ is the *start* (or *initial*) *state*,
- (5) $F \subseteq K$ is the set of *accepting* (of *final*) *states* and
- (6) $\lambda \subseteq K \times \Sigma^* \times \Delta^* \times K$ is a finite set of quadruples called the *transition function* of M .

If $\lambda \subseteq K \times \Sigma^* \times \Delta^+ \times K$, then M is ϵ -free

Clearly, a gsm is a special kind of *a-transducer*.

An *a-transducer* defines a binary relation between Σ^* and Δ^* , or equivalently, a function $M: \Sigma^* \rightarrow 2^{\Delta^*}$.

We can explain what this function is by describing how an a -transducer makes a sequence of moves from configurations to configurations.

The current *configuration* of an a -transducer is described by a triple

$$(p, u, v) \in K \times \Sigma^* \times \Delta^*,$$

where p is the current state, u is the remaining input, and v is some output produced so far.

We define the binary relation \vdash_M on $K \times \Sigma^* \times \Delta^*$ as follows: For all $p, q \in K$, $u, \alpha \in \Sigma^*$, $\beta, v \in \Delta^*$, if $(p, u, v, q) \in \lambda$, then

$$(p, u\alpha, \beta) \vdash_M (q, \alpha, \beta v).$$

Let \vdash_M^* be the transitive and reflexive closure of \vdash_M .

The function $M: \Sigma^* \rightarrow 2^{\Delta^*}$ is defined such that for every $w \in \Sigma^*$,

$$M(w) = \{y \in \Delta^* \mid (q_0, w, \epsilon) \vdash_M^* (f, \epsilon, y), f \in F\}.$$

For any language $L \subseteq \Sigma^*$ let

$$M(L) = \bigcup_{w \in L} M(w).$$

For any $y \in \Delta^*$, let

$$M^{-1}(y) = \{w \in \Sigma^* \mid y \in M(w)\}$$

and for any language $L' \subseteq \Delta^*$, let

$$M^{-1}(L') = \bigcup_{y \in L'} M^{-1}(y).$$

Remark: Notice that if $w \in M^{-1}(L')$, then there exists some $y \in L'$ such that $w \in M^{-1}(y)$, i.e., $y \in M(w)$. This **does not** imply that $M(w) \subseteq L'$, only that $M(w) \cap L' \neq \emptyset$.

One should realize that for any $L' \subseteq \Delta^*$ and any a -transducer, M , there is some a -transducer, M' , (from Δ^* to 2^{Σ^*}) so that $M'(L') = M^{-1}(L')$.

The following proposition is left as a homework problem:

Proposition 2.7.5 *The family of regular languages (over an alphabet Σ) is closed under both a -transductions and inverse a -transductions.*

2.8 Directed Graphs and Paths

It is often useful to view DFA's and NFA's as labeled directed graphs.

Definition 2.8.1 A *directed graph* is a quadruple $G = (V, E, s, t)$, where V is a set of *vertices*, or *nodes*, E is a set of *edges*, or *arcs*, and $s, t: E \rightarrow V$ are two functions, s being called the *source* function, and t the *target* function. Given an edge $e \in E$, we also call $s(e)$ the *origin* (or *source*) of e , and $t(e)$ the *endpoint* (or *target*) of e .

Remark: the functions s, t need not be injective or surjective. Thus, we allow “isolated vertices”.

Example: Let G be the directed graph defined such that

$$E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\},$$

$$V = \{v_1, v_2, v_3, v_4, v_5, v_6\}, \text{ and}$$

$$s(e_1) = v_1, s(e_2) = v_2, s(e_3) = v_3, s(e_4) = v_4,$$

$$s(e_5) = v_2, s(e_6) = v_5, s(e_7) = v_5, s(e_8) = v_5,$$

$$t(e_1) = v_2, t(e_2) = v_3, t(e_3) = v_4, t(e_4) = v_2,$$

$$t(e_5) = v_5, t(e_6) = v_5, t(e_7) = v_6, t(e_8) = v_6.$$

Such a graph can be represented by the following diagram:

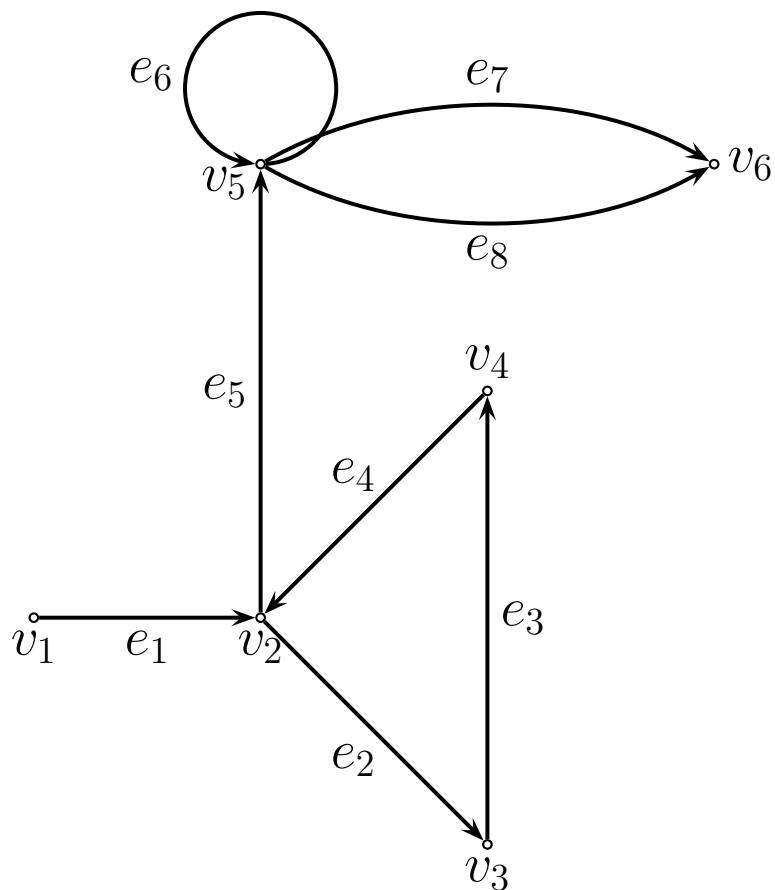


Figure 2.7: A directed graph

In drawing directed graphs, we will usually omit edge names (the e_i), and sometimes even the node names (the v_j).

We now define paths in a directed graph.

Definition 2.8.2 Given a directed graph $G = (V, E, s, t)$ for any two nodes $u, v \in V$, a *path from u to v* is a triple $\pi = (u, e_1 \dots e_n, v)$, where $e_1 \dots e_n$ is a string (sequence) of edges in E such that, $s(e_1) = u$, $t(e_n) = v$, and $t(e_i) = s(e_{i+1})$, for all i such that $1 \leq i \leq n - 1$. When $n = 0$, we must have $u = v$, and the path (u, ϵ, u) is called the *null path from u to u* . The number n is the *length* of the path. We also call u the *source* (or *origin*) of the path, and v the *target* (or *endpoint*) of the path. When there is a nonnull path π from u to v , we say that *u and v are connected*.

Remark: In a path $\pi = (u, e_1 \dots e_n, v)$, the expression $e_1 \dots e_n$ is a **sequence**, and thus, the e_i are **not** necessarily distinct.

For example, the following are paths:

$$\pi_1 = (v_1, e_1 e_5 e_7, v_6),$$

$$\pi_2 = (v_2, e_2 e_3 e_4 e_2 e_3 e_4 e_2 e_3 e_4, v_2),$$

and

$$\pi_3 = (v_1, e_1 e_2 e_3 e_4 e_2 e_3 e_4 e_5 e_6 e_6 e_8, v_6).$$

Clearly, π_2 and π_3 are of a different nature from π_1 . Indeed, they contain cycles. This is formalized as follows.

Definition 2.8.3 Given a directed graph $G = (V, E, s, t)$, for any node $u \in E$ a *cycle (or loop) through u* is a nonnull path of the form $\pi = (u, e_1 \dots e_n, u)$ (equivalently, $t(e_n) = s(e_1)$). More generally, a nonnull path $\pi = (u, e_1 \dots e_n, v)$ *contains a cycle* iff for some i, j , with $1 \leq i \leq j \leq n$, $t(e_j) = s(e_i)$. In this case, letting $w = t(e_j) = s(e_i)$, the path $(w, e_i \dots e_j, w)$ is a cycle through w . A path π is *acyclic* iff it does not contain any cycle. Note that each null path (u, ϵ, u) is acyclic.

Obviously, a cycle $\pi = (u, e_1 \dots e_n, u)$ through u is also a cycle through every node $t(e_i)$. Also, a path π may contain several different cycles. Paths can be concatenated as follows.

Definition 2.8.4 Given a directed graph

$G = (V, E, s, t)$, two paths $\pi_1 = (u, e_1 \dots e_m, v)$ and $\pi_2 = (u', e'_1 \dots e'_n, v')$ can be *concatenated* provided that $v = u'$, in which case their *concatenation* is the path

$$\pi_1\pi_2 = (u, e_1 \dots e_m e'_1 \dots e'_n, v').$$

It is immediately verified that the concatenation of paths is associative, and that the concatenation of the path $\pi = (u, e_1 \dots e_m, v)$ with the null path (u, ϵ, u) or with the null path (v, ϵ, v) is the path π itself.

The following fact, although almost trivial, is used all the time, and is worth stating in detail.

Lemma 2.8.5 Given a directed graph $G = (V, E, s, t)$, if the set of nodes V contains $m \geq 1$ nodes, then every path π of length at least m contains some cycle.

A consequence of lemma 2.8.5 is that in a finite graph with m nodes, given any two nodes $u, v \in V$, in order to find out whether there is a path from u to v , it is enough to consider paths of length $\leq m - 1$.

Indeed, if there is path between u and v , then there is some path π of minimal length (not necessarily unique, but this doesn't matter). If this minimal path has length at least m , then by the lemma, it contains a cycle. However, by deleting this cycle from the path π , we get an even shorter path from u to v , contradicting the minimality of π .

We now turn to labeled graphs.

2.9 Labeled Graphs and Automata

In fact, we only need edge-labeled graphs.

Definition 2.9.1 A *labeled directed graph* is a tuple $G = (V, E, L, s, t, \lambda)$, where V is a set of *vertices*, or *nodes*, E is a set of *edges*, or *arcs*, L is a set of *labels*, $s, t: E \rightarrow V$ are two functions, s being called the *source* function, and t the *target* function, and $\lambda: E \rightarrow L$ is the *labeling function*. Given an edge $e \in E$, we also call $s(e)$ the *origin* (or *source*) of e , $t(e)$ the *endpoint* (or *target*) of e , and $\lambda(e)$ the *label* of e .

Note that the function λ need not be injective or surjective. Thus, distinct edges may have the same label.

Example: Let G be the directed graph defined such that

$$E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\},$$

$$V = \{v_1, v_2, v_3, v_4, v_5, v_6\}, L = \{a, b\}, \text{ and}$$

$$s(e_1) = v_1, s(e_2) = v_2, s(e_3) = v_3, s(e_4) = v_4,$$

$$s(e_5) = v_2, s(e_6) = v_5, s(e_7) = v_5, s(e_8) = v_5,$$

$$t(e_1) = v_2, t(e_2) = v_3, t(e_3) = v_4, t(e_4) = v_2,$$

$$t(e_5) = v_5, t(e_6) = v_5, t(e_7) = v_6, t(e_8) = v_6.$$

$$\lambda(e_1) = a, \lambda(e_2) = b, \lambda(e_3) = a, \lambda(e_4) = a,$$

$$\lambda(e_5) = b, \lambda(e_6) = a, \lambda(e_7) = a, \lambda(e_8) = b.$$

Such a labeled graph can be represented by the following diagram:

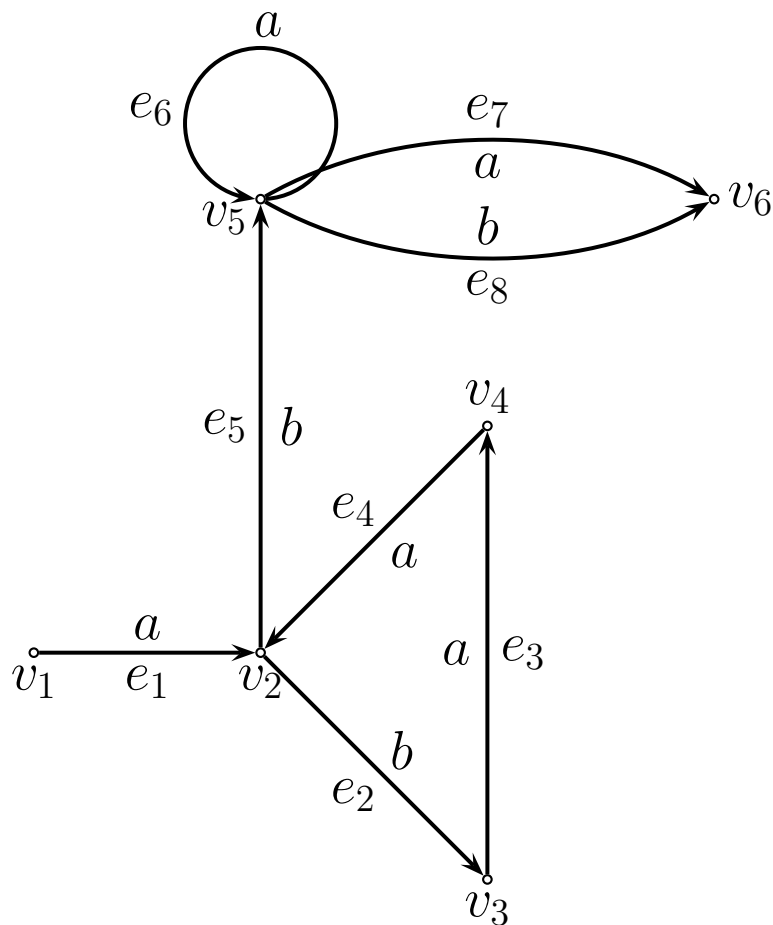


Figure 2.8: A labeled directed graph

In drawing labeled graphs, we will usually omit edge names (the e_i), and sometimes even the node names (the v_j). Paths, cycles, and concatenation of paths are defined just as before (that is, we ignore the labels). However, we can now define the *spelling* of a path.

Definition 2.9.2 Given a labeled directed graph $G = (V, E, L, s, t, \lambda)$ for any two nodes $u, v \in V$, for any path $\pi = (u, e_1 \dots e_n, v)$, the *spelling of the path* π is the string of labels

$$\lambda(e_1) \dots \lambda(e_n).$$

When $n = 0$, the spelling of the null path (u, ϵ, u) is the null string ϵ .

For example, the spelling of the path

$$\pi_3 = (v_1, e_1 e_2 e_3 e_4 e_2 e_3 e_4 e_5 e_6 e_6 e_8, v_6)$$

is

$$abaabaabaab.$$

Every DFA and every NFA can be viewed as a labeled graph, in such a way that the set of spellings of paths from the start state to some final state is the language accepted by the automaton in question.

Given a DFA $D = (Q, \Sigma, \delta, q_0, F)$, where $\delta: Q \times \Sigma \rightarrow Q$, we associate the labeled directed graph $G_D = (V, E, L, s, t, \lambda)$ defined as follows:

$$V = Q, \quad E = \{(p, a, q) \mid q = \delta(p, a), p, q \in Q, a \in \Sigma\},$$

$$L = \Sigma, \quad s((p, a, q)) = p, \quad t((p, a, q)) = q,$$

$$\text{and } \lambda((p, a, q)) = a.$$

Such labeled graphs have a special structure that can easily be characterized.

It is easily shown that a string $w \in \Sigma^*$ is in the language $L(D) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}$ iff w is the spelling of some path in G_D from q_0 to some final state.

Similarly, given an NFA $N = (Q, \Sigma, \delta, q_0, F)$, where $\delta: Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$, we associate the labeled directed graph $G_N = (V, E, L, s, t, \lambda)$ defined as follows:

$$V = Q$$

$$E = \{(p, a, q) \mid q \in \delta(p, a), p, q \in Q, a \in \Sigma \cup \{\epsilon\}\},$$

$$L = \Sigma \cup \{\epsilon\}, s((p, a, q)) = p, t((p, a, q)) = q, \\ \lambda((p, a, q)) = a.$$

Remark: When N has no ϵ -transitions, we can let $L = \Sigma$.

Such labeled graphs have also a special structure that can easily be characterized.

Again, a string $w \in \Sigma^*$ is in the language

$L(N) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \cap F \neq \emptyset\}$ iff w is the spelling of some path in G_N from q_0 to some final state.

2.10 The Closure Definition of the Regular Languages

Let $\Sigma = \{a_1, \dots, a_m\}$ be an alphabet. We define a family $(R(\Sigma)_n)$ of sets of languages as follows:

$$R(\Sigma)_0 = \{\{a_1\}, \dots, \{a_m\}, \emptyset, \{\epsilon\}\},$$

$$R(\Sigma)_{n+1} = R(\Sigma)_n \cup \{L_1 \cup L_2, L_1L_2, L^* \mid L_1, L_2, L \in R(\Sigma)_n\}.$$

Then, we define $R(\Sigma)$ as

$$R(\Sigma) = \bigcup_{n \geq 0} R(\Sigma)_n.$$

Regular languages, Version 2 = $R(\Sigma)$

Lemma The family $R(\Sigma)$ is the smallest family of languages which contains the (atomic) languages $\{a_1\}, \dots, \{a_m\}, \emptyset, \{\epsilon\}$, and is closed under union, concatenation, and Kleene $*$.

Proof: Use induction on n .

Note: a given language L may be built up in different ways. For example,

$$\{a, b\}^* = (\{a\}^* \{b\}^*)^*.$$

2.11 Regular Expressions

Given an alphabet $\Sigma = \{a_1, \dots, a_m\}$, consider the new alphabet

$$\Delta = \Sigma \cup \{+, \cdot, *, (,), \emptyset, \epsilon\}.$$

We define a family $(\mathcal{R}(\Sigma)_n)$ of languages over Δ as follows:

$$\begin{aligned} \mathcal{R}(\Sigma)_0 &= \{a_1, \dots, a_m, \emptyset, \epsilon\}, \\ \mathcal{R}(\Sigma)_{n+1} &= \mathcal{R}(\Sigma)_n \cup \{(R_1 + R_2), (R_1 \cdot R_2), R^* \mid \\ &\quad R_1, R_2, R \in \mathcal{R}(\Sigma)_n\}. \end{aligned}$$

Then, we define $\mathcal{R}(\Sigma)$ as

$$\mathcal{R}(\Sigma) = \bigcup_{n \geq 0} \mathcal{R}(\Sigma)_n.$$

$\mathcal{R}(\Sigma)$ is the set of *regular expressions* (over Σ).

Lemma The language $\mathcal{R}(\Sigma)$ is the smallest language which contains the symbols $a_1, \dots, a_m, \emptyset, \epsilon$, from Δ , and such that $(R_1 + R_2)$, $(R_1 \cdot R_2)$, and R^* , also belong to $\mathcal{R}(\Sigma)$, when $R_1, R_2, R \in \mathcal{R}(\Sigma)$.

For simplicity of notation, write

$$(R_1 R_2)$$

instead of

$$(R_1 \cdot R_2).$$

Examples: $R = (a + b)^*$, $S = (a^* b^*)^*$.

$$T = (a + b)^* a \underbrace{(a + b) \cdots (a + b)}_n.$$

2.12 Regular Expressions and Regular Languages

Every regular expression $R \in \mathcal{R}(\Sigma)$ can be viewed as the *name*, or *denotation*, of some language $L \in \mathcal{L}(\Sigma)$. Similarly, every language $L \in \mathcal{L}(\Sigma)$ is the *interpretation* (or *meaning*) of some regular expression $R \in \mathcal{R}(\Sigma)$.

Think of a regular expression R as a *program*, and of $\mathcal{L}(R)$ as the result of the *execution* or *evaluation*, of R by \mathcal{L} .

This can be made rigorous by defining a function

$$\mathcal{L}: \mathcal{R}(\Sigma) \rightarrow \mathcal{L}(\Sigma).$$

This function is defined recursively:

$$\begin{aligned}\mathcal{L}[a_i] &= \{a_i\}, \\ \mathcal{L}[\emptyset] &= \emptyset, \\ \mathcal{L}[\epsilon] &= \{\epsilon\}, \\ \mathcal{L}[(R_1 + R_2)] &= \mathcal{L}[R_1] \cup \mathcal{L}[R_2], \\ \mathcal{L}[(R_1R_2)] &= \mathcal{L}[R_1]\mathcal{L}[R_2], \\ \mathcal{L}[R^*] &= \mathcal{L}[R]^*.\end{aligned}$$

Lemma For every regular expression $R \in \mathcal{R}(\Sigma)$, the language $\mathcal{L}[R]$ is regular (version 2), i.e. $\mathcal{L}[R] \in R(\Sigma)$. Conversely, for every regular (version 2) language $L \in R(\Sigma)$, there is some regular expression $R \in \mathcal{R}(\Sigma)$ such that $L = \mathcal{L}[R]$.

Note: the function \mathcal{L} is **not** injective.

Example: If $R = (a + b)^*$, $S = (a^*b^*)^*$, then

$$\mathcal{L}[R] = \mathcal{L}[S] = \{a, b\}^*.$$

For simplicity, we often denote $\mathcal{L}[R]$ as L_R .

Remark. If

$$R = (a + b)^* a \underbrace{(a + b) \cdots (a + b)}_n,$$

it can be shown that any minimal DFA accepting L_R has 2^{n+1} states.

Yet, both $(a + b)^* a$ and $\underbrace{(a + b) \cdots (a + b)}_n$ denote languages that can be accepted by “small” DFA’s (of size 2 and $n + 2$).

Definition Two regular expressions $R, S \in \mathcal{R}(\Sigma)$ are *equivalent*, denoted as $R \cong S$, iff $\mathcal{L}[R] = \mathcal{L}[S]$.

It is immediate that \cong is an equivalence relation.

The relation \cong satisfies some (nice) identities. For example:

$$\begin{aligned} ((R_1 + R_2) + R_3) &\cong (R_1 + (R_2 + R_3)), \\ ((R_1 R_2) R_3) &\cong (R_1 (R_2 R_3)), \\ (R_1 + R_2) &\cong (R_2 + R_1), \\ (R^* R^*) &\cong R^*, \\ R^{**} &\cong R^*. \end{aligned}$$

There are algorithms to test equivalence of regular expressions, but their complexity is exponential. It is an open problem to prove that the problem cannot be decided in polynomial time.

2.13 Regular Expressions and NFA's

Lemma There is an algorithm, which, given any regular expression $R \in \mathcal{R}(\Sigma)$, constructs an NFA N_R accepting L_R , i.e., such that $L_R = L(N_R)$.

See recursive algorithm given in class (or in any textbook). As a corollary,

Reg. languages version 2 \subseteq Reg. languages, version 1.

The reader should check that if one constructs the NFA corresponding to the regular expression $(a + b)^*abb$ and then applies the subset construction, one get the following DFA:

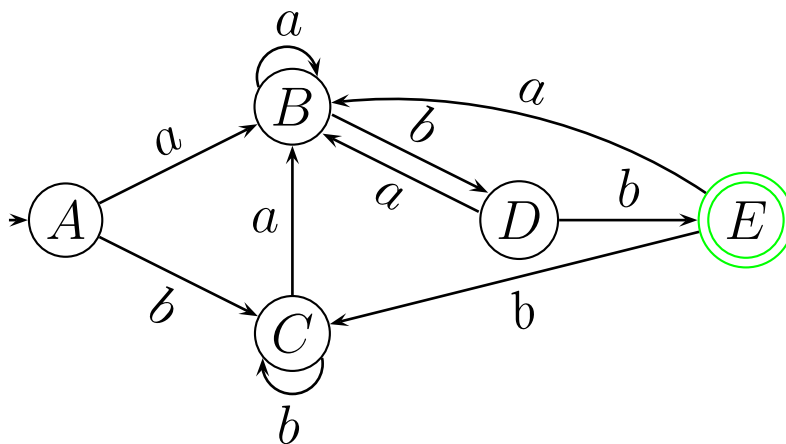


Figure 2.9: A non-minimal DFA for $\{a, b\}^*\{abb\}$

Lemma There is an algorithm, which, given any NFA N , constructs a regular expression $R \in \mathcal{R}(\Sigma)$, denoting $L(N)$, i.e., such that $L_R = L(N)$

As a corollary,

Reg. languages version 1 \subseteq Reg. languages, version 2.

This is the *node elimination algorithm*. The general idea is to allow more general labels on the edges of an NFA, namely, regular expressions. Then, such generalized NFA's are simplified by eliminating nodes one at a time, and readjusting labels.

Preprocessing, phase 1:

If necessary, we need to add a new start state with an ϵ -transition to the old start state, if there are incoming edges into the old start state.

If necessary, we need to add a new (unique) final state with ϵ -transitions from each of the old final states to the new final state, if there is more than one final state or some outgoing edge from any of the old final states.

At the end of this phase, the start state, say s , is a source (no incoming edges), and the final state, say t , is a sink (no outgoing edges).

Preprocessing, phase 2:

We need to “flatten” parallel edges. For any pair of states (p, q) ($p = q$ is possible), if there are k edges from p to q labeled u_1, \dots, u_k , then create a single edge labeled with the regular expression

$$u_1 + \dots + u_k.$$

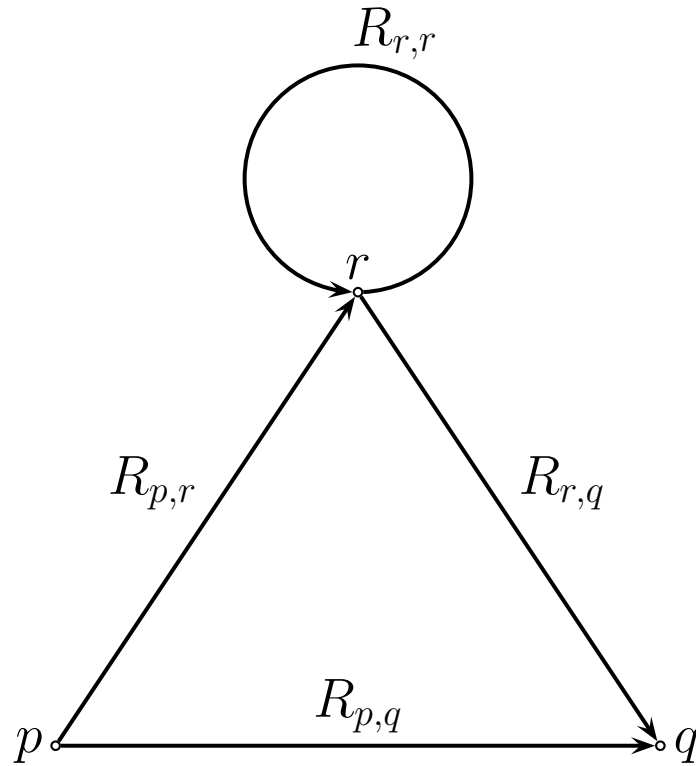
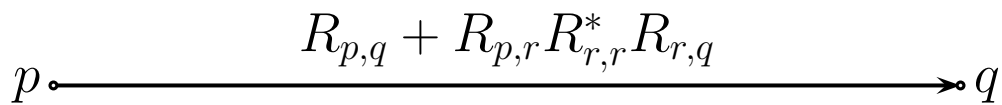
For any pair of states (p, q) ($p = q$ is possible) such that there is **no** edge from p to q , we put an edge labeled \emptyset .

At the end of this phase, the resulting “generalized NFA” is such that for any pair of states (p, q) (where $p = q$ is possible), there is a unique edge labeled with some regular expression denoted as $R_{p,q}$. When $R_{p,q} = \emptyset$, this really means that there is no edge from p to q in the original NFA N .

By interpreting each $R_{p,q}$ as a function call (really, a macro) to the NFA $N_{p,q}$ accepting $\mathcal{L}[R_{p,q}]$ (constructed using the previous algorithm), we can verify that the original language $L(N)$ is accepted by this new generalized NFA.

Node elimination only applies if the generalized NFA has at least one node distinct from s and t .

Pick any node r distinct from s and t . For every pair (p, q) where $p \neq r$ and $q \neq r$, replace the label of the edge from p to q as indicated below:

Figure 2.10: Before Eliminating node r Figure 2.11: After Eliminating node r

At the end of this step, delete the node r and all edges adjacent to r .

Note that $p = q$ is possible, in which case the triangle is “flat”. It is also possible that $p = s$ or $q = t$. Also, this step is performed for all **pairs** (p, q) , which means that both (p, q) and (q, p) are considered (when $p \neq q$). Note that this step only has an effect if there are edges from p to r and from r to q in the original NFA N . Otherwise, r can simply be deleted, as well as the edges adjacent to r . Other simplifications can be made. For example, when $R_{r,r} = \emptyset$, we can simplify $R_{p,r}R_{r,r}^*R_{r,q}$ to $R_{p,r}R_{r,q}$. When $R_{p,q} = \emptyset$, we have $R_{p,r}R_{r,r}^*R_{r,q}$.

The order in which the nodes are eliminated is irrelevant, although it affects the size of the final expression.

The algorithm stops when the only remaining nodes are s and t . Then, the label R of the edge from s to t is a regular expression denoting $L(N)$.

2.14 Summary of Closure Properties of the Regular Languages

The family of regular languages is closed under many operations. In particular, it is closed under the following operations listed below. Some of the closure properties are left as a homework problem.

- (1) Union, intersection, relative complement.
- (2) Concatenation, Kleene $*$, Kleene $+$.
- (3) Homomorphisms and inverse homomorphisms.
- (4) gsm and inverse gsm mappings, a -transductions and inverse a -transductions.

Another useful operation is substitution.

Given any two alphabets Σ, Δ , a *substitution* is a function, $\tau: \Sigma \rightarrow 2^{\Delta^*}$, assigning some language, $\tau(a) \subseteq \Delta^*$, to every symbol $a \in \Sigma$.

A substitution $\tau: \Sigma \rightarrow 2^{\Delta^*}$ is extended to a map $\tau: 2^{\Sigma^*} \rightarrow 2^{\Delta^*}$ by first extending τ to strings using the following definition

$$\begin{aligned}\tau(\epsilon) &= \{\epsilon\}, \\ \tau(ua) &= \tau(u)\tau(a),\end{aligned}$$

where $u \in \Sigma^*$ and $a \in \Sigma$, and then to languages by letting

$$\tau(L) = \bigcup_{w \in L} \tau(w),$$

for every language $L \subseteq \Sigma^*$.

Observe that a homomorphism is a special kind of substitution.

A substitution is a *regular* substitution iff $\tau(a)$ is a regular language for every $a \in \Sigma$. The proof of the next proposition is left as a homework problem.

Proposition 2.14.1 *If L is a regular language and τ is a regular substitution, then $\tau(L)$ is also regular. Thus, the family of regular languages is closed under regular substitutions.*