Automata theory

An algorithmic approach

Lecture Notes

Javier Esparza

October 16, 2011

Many thanks to Jörg Kreiker for many discussions on the topic of this notes, and for his contributions to several chapters. The chapter on boolean operations for Büchi automata is based on material by Orna Kupferman and Moshe Vardi. Breno Faria helped to draw many figures. He was funded by Studiengebühren. I also profited from many discussions with Jan Kretinsky. Thanks also to Birgit Engelmann, Moritz Fuchs, Stefan Krusche, Franz Saller, and Hayk Shoukourian, who attended a course based on these notes and provided helpful comments.

Contents

Chapter 1

Introduction and Preliminaries

Courses on data structures show how to represent sets of objects in a computer, so that operations like insertion, deletion, and lookup can be efficiently implemented. Typical representations are different variants of hash tables, search trees, or heaps.

In this course we also deal with the problem of representing and manipulating sets, but we are interested in the *basic operations* of set theory: union, intersection, complement with respect to some universe. We also wish to implement procedures for checking if a set is empty, or contains all elements of the universe. Finally, we also are interested in *relations*, and basic operations on them. Here is a more systematic list of operations, where U is the universe of objects, X, Y are subsets of U, x is an element of U, and $R, S \subseteq U \times U$ are binary relations on U:

Member(x, X)	:	returns **true** if $x \in X$, **false** otherwise.
Complement(X)	:	returns $U \setminus X$.
Intersection(X, Y)	:	returns $X \cap Y$.
Union(X, Y)	:	returns $X \cup Y$.
Empty(X)	:	returns **true** if $X = \emptyset$, **false** otherwise.
Universal(X)	:	returns **true** if $X = U$, **false** otherwise.
Included(X, Y)	:	returns **true** if $X \subseteq Y$, **false** otherwise.
Equal(X, Y)	:	returns **true** if $X = Y$, **false** otherwise.
Projection_1(R)	:	returns the set $\pi_1(R) = \{x \mid \exists y\ (x, y) \in R\}$.
Projection_2(R)	:	returns the set $\pi_2(R) = \{y \mid \exists y\ (x, y) \in R\}$.
Join(R, S)	:	returns $R \circ S = \{(x, z) \mid \exists y \in X\ (x, y) \in R \wedge (y, z) \in S\}$
Post(X, R)	:	returns $post_R(X) = \{y \in U \mid \exists x \in X : (x, y) \in R\}$.
Pre(X, R)	:	returns $pre_R(X) = \{y \in U \mid \exists x \in X : (y, x) \in R\}$.

Observe that many other operations, e.g. set difference, can be reduced to the ones above. Similarly, operations on n-ary relations for $n \geq 3$ can be reduced to operations on binary relations.

An important point is that we are not only interested on finite sets, we wish to have a data structure able to deal with infinite sets over some infinite universe. Observe, however, that no data

structure can represent *all* infinite sets. The reason is that there are uncountably many subsets of an infinite universe, but, since every instance of a data structure must have finite size, every data structure has only countably many instances. So, loosely speaking, there are many more sets to be represented than representations available, which implies that not all sets can be represented. Because of this limitation every good data structure for infinite sets must find a reasonable compromise between *expressibility* (how large is the set of representable sets) and *manipulability* (which operations can be carried out, and at which cost). The goal of these notes is to present the compromise offered by *word automata*, which, as shown by 50 years of research on the theory of formal languages, is the best one available for most purposes. Word automata, or just automata, are a data structure for representing and manipulating sets whose elements are encoded as *words*, i.e., as sequences of letters over an alphabet[1]

Notice that any kind of object can, at least in principle, be represented by a word. Natural numbers, for instance, are represented in computer science as sequences of digits, i.e., as words over the alphabet of digits. Vectors and lists can also be represented as words by concatenating the word representations of their elements. In fact, whenever we store an object in a file we are always representing it as a word over some alphabet, like ASCII. So word automata are a very general and powerful structure. However, while any object can be represented by a word, not every object can be represented by a *finite* word, that is, a word of finite length. A typical example are real numbers, but there are others, like non-terminating executions of a program. When objects cannot be represented by finite words, computers usually renounce the goal of exactly representing them, and representing some approximation of the object: a float instead of a real number, or a finite prefix instead of a non-terminating computation. A fascinating possibility, which we study in the second part of the notes, is to use automata to exactly represent certain sets of infinite objects.

Automata for representing sets of *finite* words and sets of *infinite* words are studied in Part I and Part II of these notes. The theory of automata on finite words is often considered a "gold standard" of theoretical computer science, a powerful and beautiful theory with lots of important applications in many fields. Automata on infinite words are harder, and their theory does not achieve the same degree of "perfection". This gives us a structure for Part II of the notes: we follow the steps of Part I, always comparing the solutions for infinite words with the "gold standard". We point out why some techniques cannot be extended from finite to infinite words, and present replacements, or why no replacement is possible.

[1]There are generalizations of word automata in which objects are encoded as trees. The theory of tree automata is also very well developed, but not the subject of these notes. So we shorten word automaton to just automaton.

Part I

Automata on Finite Words

Chapter 2

Automata Classes and Conversions

In Section 2.1 we introduce basic definitions about words and languages, and then introduce regular expressions, a textual notation for defining languages of finite words. Like any other formal notation, it cannot be used to define each possible language. However, the next chapter shows that they are an adequate notation when dealing with automata, since they define exactly the languages that can be represented by automata on words.

2.1 Languages and regular expressions

An *alphabet* is a finite, nonempty set. The elements of an alphabet are called *letters*. A finite, possibly empty sequence of letters is a *word*. A word $a_1 a_2 \dots a_n$ has *length* n. The empty word is the only word of length 0 and it is written ε. The concatenation of two words $w_1 = a_1 \dots a_n$ and $w_2 = b_1 \dots b_m$ is the word $w_1 w_2 = a_1 \dots a_n b_1 \dots b_m$, sometimes also denoted by $w_1 \cdot w_2$. Notice that $\varepsilon \cdot w = w = w \cdot \varepsilon = w$. For every word w, we define $w^0 = \varepsilon$ and $w^{k+1} = w^k w$.

Given an alphabet Σ, we denote by Σ^* the set of all words over Σ. A set $L \subseteq \Sigma^*$ of words is a *language* over Σ.

The *complement* of a language L is the language $\Sigma^* \setminus L$, which we often denote by $\overline{L}$ (notice that this notation implicitly assumes the alphabet Σ is fixed). The *concatenation* of two languages L_1 and L_2 is $L_1 \cdot L_2 = \{w_1 w_2 \in \Sigma^* \mid w_1 \in L_1, w_2 \in L_2\}$. The *iteration* of a language $L \subseteq \Sigma^*$ is the language $L^* = \bigcup_{i \geq 0} L^i$, where $L_0 = \{\varepsilon\}$ and $L_{i+1} = L^i \cdot L$ for every $i \geq 0$.

Definition 2.1 *Regular expressions r over an alphabet Σ are defined by the following grammar, where $a \in \Sigma$*

$$r ::= \emptyset \mid \varepsilon \mid a \mid r_1 r_2 \mid r_1 + r_2 \mid r^*$$

The set of all regular expressions over Σ is written $\mathcal{RE}(\Sigma)$. The language *$L(r) \subseteq \Sigma^*$ of a regular expression $r \in \mathcal{RE}(\Sigma)$ is defined inductively as*

- $L(\emptyset) = \emptyset$,

- $L(\varepsilon) = \{\varepsilon\}$,
- $L(a) = \{a\}$,
- $L(r_1 r_2) = L(r_1) \cdot L(r_2)$,
- $L(r_1 + r_2) = L(r_1) \cup L(r_2)$, *and*
- $L(r^*) = L(r)^*$.

A language L is regular *if there is a regular expression r such that $L = L(r)$.*

In the coming chapters we often abuse language, and write "the language r" instead of "the language $L(r)$."

2.2 Automata classes

We briefly recapitulate the definitions of deterministic and nondeterministic finite automata, as well as nondeterministic automata with ε-transitions and regular expressions.

Definition 2.2 *A* deterministic automaton (DA) *is a tuple $A = (Q, \Sigma, \delta, q_0, F)$, where*

- *Q is a set of* states*,*
- *Σ is an alphabet,*
- *$\delta\colon Q \times \Sigma \to Q$ is a* transition function*,*
- *$q_0 \in Q$ is the* initial state*, and*
- *$F \subseteq Q$ is the set of* final states*.*

A run *of A on input $a_0 a_1 \dots a_n$ is a sequence $p_0 \xrightarrow{a_0} p_1 \xrightarrow{a_1} p_2 \dots \xrightarrow{a_{n-1}} p_n$, such that $p_i \in Q$ for $0 \leq i \leq n$, $p_0 = q_0$, and $\delta(p_i, a_i) = p_{i+1}$ for $0 \leq i < n - 1$. A run is* accepting *if $p_n \in F$. A* accepts *a word $w \in \Sigma^*$, if there is an accepting run on input w. The* language recognized *by A is the set $L(A) = \{w \in \Sigma^* \mid w \text{ is accepted by } A\}$.*

A deterministic finite automaton *(DFA) is a DA with a finite set of states.*

Notice that a DA has exactly one run on a given word. Given a DA, we often say "the word w leads from q_0 to q", meaning that the unique run of the DA on the word w ends at the state q.

Definition 2.3 *A* non-deterministic automaton (NA) *is a tuple $A = (Q, \Sigma, \delta, q_0, F)$, where Q, Σ, q_0, and F are as for DAs and*

- *$\delta\colon Q \times \Sigma \to \mathcal{P}(Q)$ is a* transition relation*.*

The runs of NAs are defined as for DAs, but substituting $p_{i+1} \in \delta(p_i, a_i)$ *for* $\delta(p_i, a_i) = p_{i+1}$. *Acceptance and the language recognized by a NA are defined as for DAs. A* nondeterministic finite automaton *(NFA) is a NA with a finite set of states.*

We often identify the transition function δ of a DA with the set of triples (q, a, q') such that $q' = \delta(q, a)$, and the transition relation δ of a NFA with the set of triples (q, a, q') such that $q' \in \delta(q, a)$; so we often write $(q, a, q') \in \delta$, meaning $q' = \delta(q, a)$ for a DA, or $q' \in \delta(q, a)$ for a NA.

Definition 2.4 *A* non-deterministic automaton with ε-transitions (NA-ε) *is a tuple* $A = (Q, \Sigma, \delta, q_0, F)$, *where* Q, Σ, q_0, *and* F *are as for NAs and*

- $\delta\colon Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ *is a* transition relation.

The runs and accepting runs of NA-ε *are defined as for NAs. A* accepts *a word* $a_1 \dots a_n \in \Sigma^*$ *if A has an accepting run on* $\varepsilon^{k_0} a_1 \varepsilon^{k_1} \dots \varepsilon^{k_{n-1}} a_n \varepsilon^{k_n} \in (\Sigma \cup \{\varepsilon\})^*$ *for some* $k_0, k_1, \dots, k_n \geq 0$.

A nondeterministic finite automaton with ε-transitions *(NFA-*ε*) is a NA-*ε *with a finite set of states.*

Definition 2.5 *Let* $A = (Q, \Sigma, \delta, q_0, F)$ *be an automaton. A state* $q \in Q$ *is* reachable from $q' \in Q$ *if* $q = q'$ *or if there exists a run* $q' \xrightarrow{a_1} \dots \xrightarrow{a_n} q$ *on some input* $a_1 \dots a_n \in \Sigma^*$. *A is in* normal form *if every state is reachable from the initial state.*

> **Convention:** Unless otherwise stated, we assume that automata are in normal form. All our algorithms preserve normal forms, i.e., when the output is an automaton, the automaton is in normal form.

We extend NAs to allow regular expressions on transitions. Such automata are called *NA-reg* and they are obviously a generalization of both regular expressions and NA-εs. They will be useful to provide a uniform conversion between automata and regular expressions.

Definition 2.6 *A* non-deterministic automaton with regular expression transitions (NA-reg) *is a tuple* $A = (Q, \Sigma, \delta, q_0, F)$, *where* Q, Σ, q_0, *and* F *are as for NAs, and where*

- $\delta\colon Q \times \mathcal{RE}(\Sigma) \to \mathcal{P}(Q)$ *is a relation such that* $\delta(q, r) = \emptyset$ *for all but a finite number of pairs* $(q, r) \in Q \times \mathcal{RE}(\Sigma)$.

Accepting runs are defined as for NFAs. A accepts a word $w \in \Sigma^*$ *if A has an accepting run on* $r_1 \dots r_k$ *such that* $w = L(r_1) \cdot \ldots \cdot L(r_k)$.

A nondeterministic finite automaton with regular expression transitions (NFA-reg) *is a NA-reg with a finite set of states.*

2.3 Conversion Algorithms between Finite Automata

We recall that all our data structures can represent exactly the same languages. Since DFAs are a special case of NFA, which are a special case of NFA-ε, it suffices to show that every language recognized by an NFA-ε can also be recognized by an NFA, and every language recognized by an NFA can also be recognized by a DFA.

2.3.1 From NFA to DFA.

The *subset construction* transforms an NFA A into a DFA B recognizing the same language. We first give an informal idea of the construction. Recall that a NFA may have many different runs on a word w, possibly leading to different states, while a DFA has exactly one run on w. Denote by Q_w the set of states q such that some run of A on w leads from its initial state q_0 to q. Intuitively, B "keeps track" of the set Q_w: its states are *sets of states* of A, with $\{q_0\}$ as initial state, and its transition function is defined to ensure that the run of B on w leads from $\{q_0\}$ to Q_w (see below). It is then easy to ensure that A and B recognize the same language: it suffices to choose the final states of B as the sets of states of A containing *at least one* final state, because for every word w:

$$\begin{array}{ll} & B \text{ accepts } w \\ \text{iff} & Q_w \text{ is a final state of } B \\ \text{iff} & Q_w \text{ contains at least a final state of } A \\ \text{iff} & \text{some run of } A \text{ on } w \text{ leads to a final state of } A \\ \text{iff} & A \text{ accepts } w. \end{array}$$

Let us now define the transition function of B, say Δ. "Keeping track of the set Q_w" amounts to satisfying $\Delta(Q_w, a) = Q_{wa}$ for every word w. But we have $Q_{wa} = \bigcup_{q \in Q_w} \delta(q, a)$, and so we define

$$\Delta(Q', a) = \bigcup_{q \in Q'} \delta(q, a)$$

for every $Q' \subseteq Q$.

Summarizing, given $A = (Q, \Sigma, \delta, q_0, F)$ we define the DFA $B = (\mathcal{Q}, \Sigma, \Delta, Q_0, \mathcal{F})$ as follows:

- $\mathcal{Q} = \mathcal{P}(Q)$;
- $\Delta(Q', a) = \bigcup_{q \in Q'} \delta(q, a)$ for every $Q' \subseteq Q$ and every $a \in \Sigma$;
- $Q_0 = \{q_0\}$; and
- $\mathcal{F} = \{Q' \in \mathcal{Q} \mid Q' \cap F \neq \emptyset\}$.

Notice, however, that B may not be in normal form: it may have many states non-reachable from Q_0. For instance, assume A happens to be a DFA with states $\{q_0, \ldots, q_{n-1}\}$. Then B has 2^n states, but only the singletons $\{q_0\}, \ldots, \{q_{n-1}\}$ are reachable, all other states are superfluous!

NFAtoDFA(A)
Input: NFA $A = (Q, \Sigma, \delta, q_0, F)$
Output: DFA $B = (\mathcal{Q}, \Sigma, \Delta, Q_0, \mathcal{F})$ with $L(B) = L(A)$

```
1  Q, Δ, F ← ∅; Q_0 ← {q_0}
2  W = {Q_0}
3  while W ≠ ∅ do
4     pick Q' from W
5     add Q' to Q
6     if Q' ∩ F ≠ ∅ then add Q' to F
7     for all a ∈ Σ do
8        Q'' ← ⋃_{q∈Q'} δ(q, a)
9        if Q'' ∉ Q then add Q'' to W
10       add (Q', a, Q'') to Δ
```

Table 2.1: *NFAtoDFA*(A)

The conversion algorithm shown in Table 2.3.1 constructs only the reachable states. It is written in pseudocode, with abstract sets as data structure. Like nearly all the algorithms presented in the next chapters, it is a *worklist algorithm*. Worklist algorithms maintain a worklist of objects waiting to be processed. The name 'worklist" is actually misleading, because the worklist is in fact a *workset*, it does not impose any order onto its elements, and it contains at most one copy of an element (i.e., if an element already in the workset is added to it again, the contents of the workset do not change). We however keep the name 'worklist" for historical reasons.

In *NFAtoDFA*() the worklist is called $\mathcal{W}$, in others just W (we use a calligraphic font to emphasize that in this case the objects of the worklist are sets). Worklist algorithms repeatedly pick an object from the worklist (instruction **pick** Q **from** $\mathcal{W}$), and process it; notice that picking an object removes it from the worklist. Processing an object may generate new objects that are added to the list. The algorithm terminates when the worklist is empty. Since objects removed from the list may generate new objects, worklist algorithms may potentially fail to terminate, even if the set of all objects is finite, because the same object might be added to and removed from the worklist infinitely many times. Termination is guaranteed by making sure that no object that has been removed from the list once is ever added to it again. For this, objects picked from the worklist are stored (in *NFAtoDFA*() they are stored in $\mathcal{Q}$), and objects are added to the worklist only if they have not been stored yet.

Figure 2.1 shows an NFA at the top, and some snapshots of the run of *NFAtoDFA*() on it. The states of the DFA are labelled with the corresponding sets of states of the NFA. The algorithm picks states from the worklist in order $\{1\}$, $\{1,2\}$, $\{1,3\}$, $\{1,4\}$, $\{1,2,4\}$. Snapshots (a)-(d) are taken right after it picks the states $\{1,2\}$, $\{1,3\}$, $\{1,4\}$, and $\{1,2,4\}$, respectively. Snapshot (e) is taken at the

end. Notice that out of the $2^4 = 16$ subsets of states of the NFA only 5 are constructed, because the rest are not reachable from $\{1\}$.

Complexity. If A has n states, then the output of *NFAtoDFA*(A) can have up to 2^n states. To show that this bound is essentially reachable, consider the family $\{L_n\}_{n\geq 1}$ of languages over $\Sigma = \{a, b\}$ given by $L_n = (a+b)^* a (a+b)^{(n-1)}$. That is, L_n contains the words of length at least n whose n-th letter *starting from the end* is an a. The language L_n is accepted by the NFA with $n+1$ states shown in Figure 2.2(a): intuitively, the automaton chooses one of the a's in the input word, and checks that it is followed by exactly $n-1$ letters before the word ends. Applying the subset construction, however, yields a DFA with 2^n states. The DFA for L_3 is shown on the left of Figure 2.2(b). The states of the DFA have a natural interpretation: they "store" the last n letters read by the automaton. If the DFA is in the state storing $a_1 a_2 \dots a_n$ and it reads the letter a_{n+1}, then it moves to the state storing $a_2 \dots a_{n+1}$. States are final if the first letter they store is an a. The interpreted version of the DFA is shown on right of Figure 2.2(b).

We can also easily prove that *any* DFA recognizing L_n must have at least 2^n states. Assume there is a DFA $A_n = (Q, \Sigma, \delta, q_0, F)$ such that $|Q| < 2^n$ and $L(A_n) = L_n$. We can extend δ to a mapping $\hat{\delta}\colon Q \times \{a, b\}^* \to Q$, where $\hat{\delta}(q, \varepsilon) = q$ and $\hat{\delta}(q, w \cdot \sigma) = \delta(\hat{\delta}(q, w), \sigma)$ for all $w \in \Sigma^*$ and for all $\sigma \in \Sigma$. Since $|Q| < 2^n$, there must be two words $u \cdot a \cdot v_1$ and $u \cdot b \cdot v_2$ of length n for which $\hat{\delta}(q_0, u \cdot a \cdot v_1) = \hat{\delta}(q_0, u \cdot b \cdot v_2)$. But then we would have that $\hat{\delta}(q_0, u \cdot a \cdot v_1 \cdot u) = \hat{\delta}(q_0, u \cdot b \cdot v_2 \cdot u)$; that is, either both $u \cdot a \cdot v_1 \cdot u$ and $u \cdot b \cdot v_2 \cdot u$ are accepted by A_n or neither do. Since, however, $|a \cdot v_1 \cdot u| = |b \cdot v_2 \cdot u| = n$, this contradicts the assumption that A_n consists of exactly the words with an a at the n-th position from the end.

2.3.2 From NFA-ε to NFA.

Let A be a NFA-ε over an alphabet Σ. In this section we use a to denote an element of Σ, and α, β to denote elements of $\Sigma \cup \{\varepsilon\}$.

Loosely speaking, the conversion first adds to A new transitions that make all ε-transitions redundant, without changing the recognized language: every word accepted by A before adding the new transitions is accepted after adding them by a run without ε-transitions. The conversion then removes all ε-transitions, delivering an NFA that recognizes the same language as A.

The new transitions are *shortcuts*: If A has transitions (q, α, q') and (q', β, q'') such that $\alpha = \varepsilon$ or $\beta = \varepsilon$, then the shortcut $(q, \alpha\beta, q'')$ is added. (Notice that either $\alpha\beta = a$ for some $a \in \Sigma$, or $\alpha\beta = \varepsilon$.) Shortcuts may generate further shortcuts: for instance, if $\alpha\beta = a$ and A has a further transition (q'', ε, q'''), then a new shortcut (q, a, q''') is added. We call the process of adding all possible shortcuts *saturation*. Obviously, saturation does not change the language of A, and if before saturation A has a run accepting a *nonempty* word, for example

$$q_0 \xrightarrow{\varepsilon} q_1 \xrightarrow{\varepsilon} q_2 \xrightarrow{a} q_3 \xrightarrow{\varepsilon} q_4 \xrightarrow{b} q_5 \xrightarrow{\varepsilon} q_6$$

then after saturation it has a run accepting the same word, and visiting no ε-transitions, namely

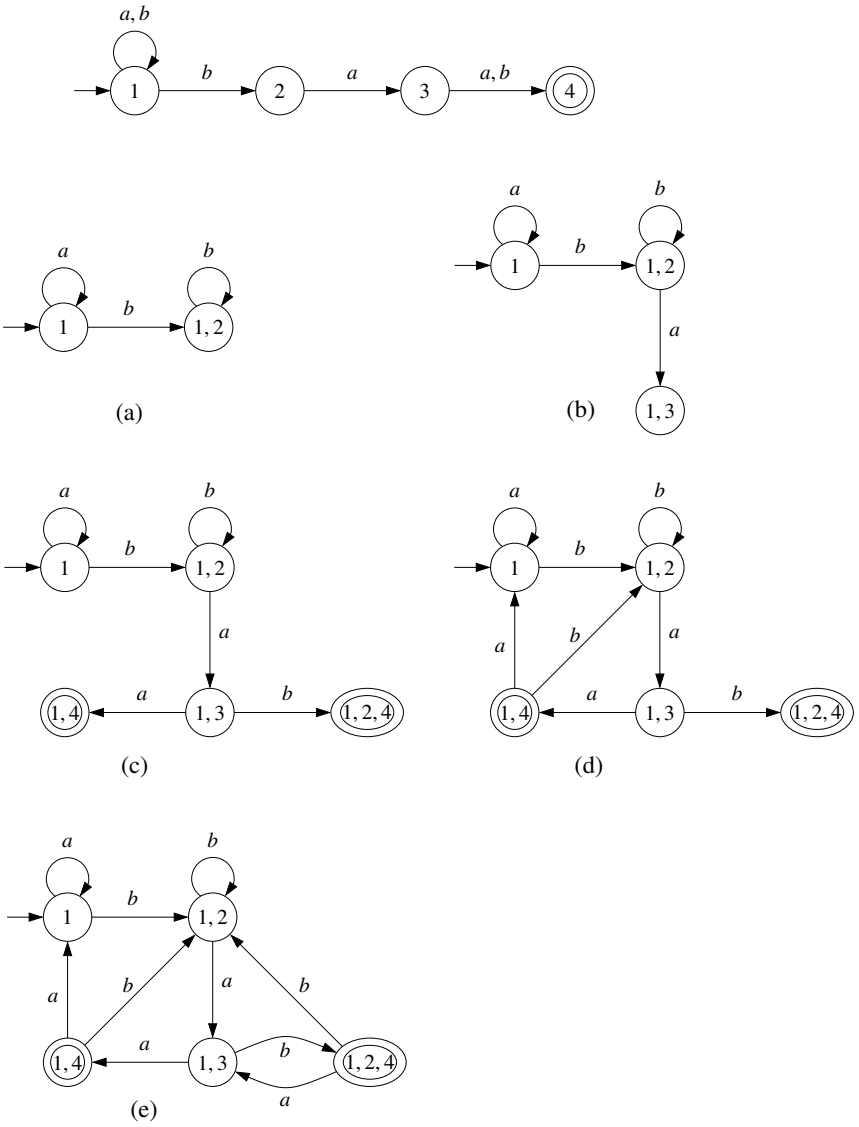
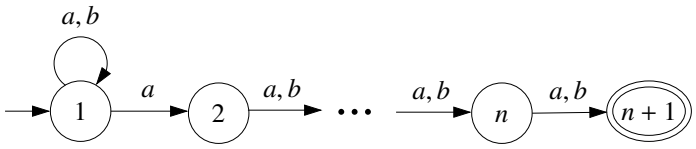
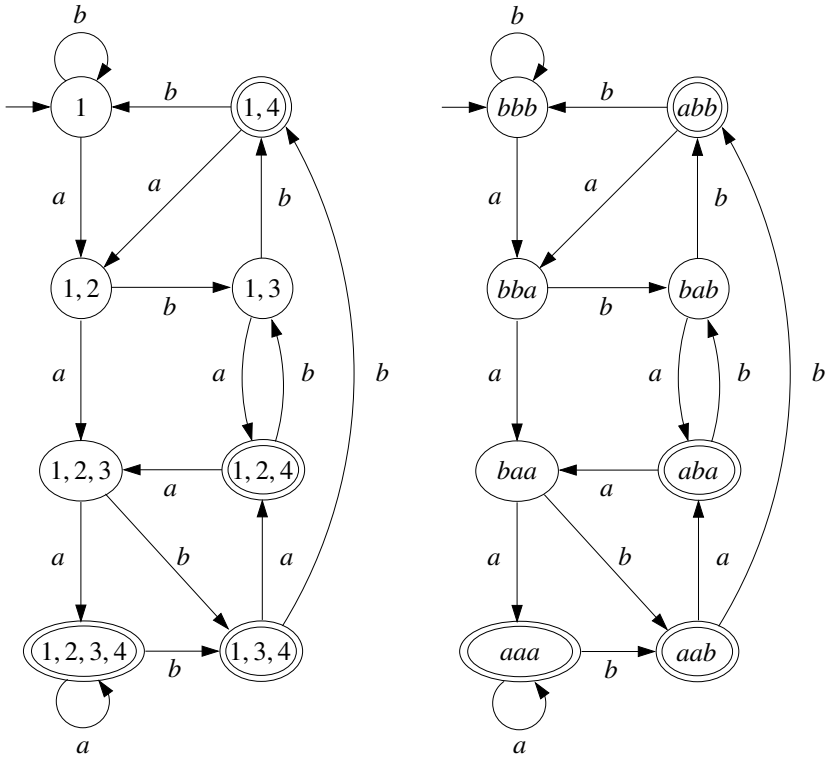


Figure 2.1: Conversion of a NFA into a DFA.



(a) NFA for L_n.

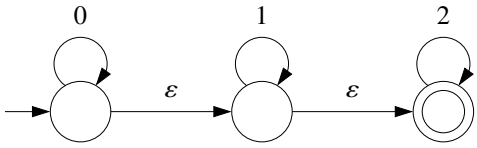


(b) DFA for L_3 and interpretation.

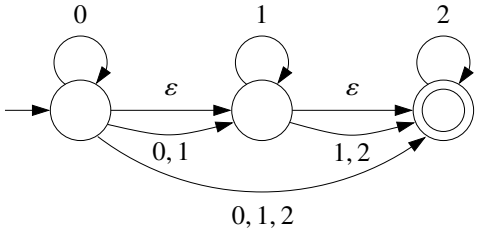
Figure 2.2: NFA for L_n, and DFA for L_3.

$$q_0 \xrightarrow{a} q_4 \xrightarrow{b} q_6$$

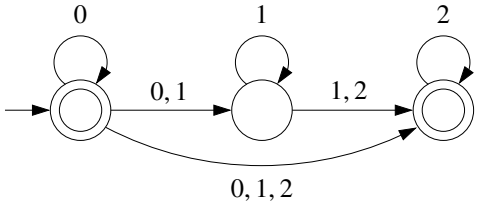
However, we cannot yet remove ε-transitions. The NFA-ε of Figure 2.3(a) accepts ε. After saturation we get the NFA-ε of Figure 2.3(b). However, removing all ε-transitions yields an NFA that no longer accepts ε. To solve this problem, if A accepts ε, then we mark its initial state as final,



(a) NFA-ε accepting $L(0^*1^*2^*)$



(b) After saturation



(c) After marking the initial state and final and removing all ε-transitions.

Figure 2.3: Conversion of an NFA-ε into an NFA by shortcutting ε-transitions.

which clearly does not change the language. To decide whether A accepts ε, we check if some state reachable from A by a sequence of ε-transitions is final. Figure 2.3(c) shows the final result. Notice that, in general, after removing ε-transitions the automaton may not be in normal form, because some states may no longer be reachable. So the naïve procedure runs in three phases: saturation, ε-check, and normalization. However, it is possible to carry all three steps in a single pass. We give a worklist implementation of this procedure in which the check is done while saturating, and only the reachable states are generated (in the pseudocode α and β stand for either a letter of Σ or ϵ, and a stands for a letter of Σ):

NFAεtoNFA(A)
Input: NFA-ε $A = (Q, \Sigma, \delta, q_0, F)$
Output: NFA $B = (Q', \Sigma, \delta', q_0', F')$ with $L(B) = L(A)$

```
1  q0' ← q0
2  Q' ← {q0}; δ' ← ∅; F' ← F ∩ {q0}
3  δ'' ← ∅; W ← {(q, α, q') ∈ δ | q = q0}
4  while W ≠ ∅ do
5      pick (q1, α, q2) from W
6      if α ≠ ε then
7          add q2 to Q'; add (q1, α, q2) to δ'; if q2 ∈ F then add q2 to F'
8          for all q3 ∈ δ(q2, ε) do
9              if (q1, α, q3) ∉ δ' then add (q1, α, q3) to W
10         for all a ∈ Σ, q3 ∈ δ(q2, a) do
11             if (q2, a, q3) ∉ δ' then add (q2, a, q3) to W
12     else  / * α = ε * /
13         add (q1, α, q2) to δ''; if q2 ∈ F then add q0 to F'
14         for all β ∈ Σ ∪ {ε}, q3 ∈ δ(q2, β) do
15             if (q1, β, q3) ∉ δ' ∪ δ'' then add (q1, β, q3) to W
```

The correctness proof is conceptually easy, but the different cases require some care, and so we devote it a proposition.

Proposition 2.7 *Let A be a NFA-ε, and let B = NFAεtoNFA(A). Then B is a NFA and L(A) = L(B).*

Proof: Notice first that every transition that leaves W is never added to W again: when a transition (q_1, α, q_2) leaves W it is added to either δ' or δ'', and a transition enters W only if it does not belong to either δ' or δ''. Since every execution of the while loop removes a transition from the worklist, the algorithm eventually exits the loop and terminates.

To show that B is a NFA we have to prove that it only has non-ε transitions, and that it is in normal form, i.e., that every state of Q' is reachable from q_0 in B. For the first part, observe that transitions are only added to δ' in line 7, and none of them is an ε-transition because of the guard in line 6. For the second part, we need the following invariant, which can be easily proved by inspection: for every transition (q_1, α, q_2) added to W, if $\alpha = \varepsilon$ then $q_1 = q_0$, and if $\alpha \neq \varepsilon$, then q_2 is reachable in B (after termination). Now, since new states are added to Q' only at line 7, applying the invariant we get that every state of Q' is reachable from q_0 in B. It remains to prove $L(A) = L(B)$. The inclusion $L(A) \supseteq L(B)$ follows from the fact that every transition added to δ' is a shortcut, which can be proved by inspection. For the inclusion $L(A) \subseteq L(B)$, we first prove that $\varepsilon \in L(A)$ implies $\varepsilon \in L(B)$. Let $q_0 \xrightarrow{\varepsilon} q_1 \ldots q_{n-1} \xrightarrow{\varepsilon} q_n$ be a run of A such that $q_n \in F$. If $n = 0$ (i.e., $q_n = q_0$), then we are done. If $n > 0$, then we prove by induction on n that a transition (q_0, ε, q_n) is eventually added to δ', and so that q_0 is eventually added to F' at line 13. If $n = 1$, then (q_0, ε, q_n)

is added at line 3. If $n > 1$, then by hypothesis $(q_0, \varepsilon, q_{n-1})$ is eventually added to W, picked from it at some later point, and (q_0, ε, q_n) is added at line 15. We now prove that for every $w \in \Sigma^+$, if $w \in L(A)$ then $w \in L(B)$. Let $w = a_1 a_2 \dots a_n$ with $n \geq 1$. Then A has a run

$$q_0 \xrightarrow{\varepsilon} \dots \xrightarrow{\varepsilon} q_{m_1} \xrightarrow{a_1} q_{m_1+1} \xrightarrow{\varepsilon} \dots \xrightarrow{\varepsilon} q_{m_n} \xrightarrow{a_n} q_{m_n+1} \xrightarrow{\varepsilon} \dots \xrightarrow{\varepsilon} q_m$$

such that $q_m \in F$. We have just proved that a transition $(q_0, \varepsilon, q_{m_1})$ is eventually added to W. So (q_0, a_1, q_{m_1+1}) is eventually added at line 15, $(q_0, a_1, q_{m+2}), \dots, (q_0, a_1, q_{m_2})$ are eventually added at line 9, and $(q_{m_2}, a_2, q_{m_2+1})$ is eventually added at line 11. Iterating this argument, we obtain that

$$q_0 \xrightarrow{a_1} q_{m_2} \xrightarrow{a_2} q_{m_3} \dots q_{m_n} \xrightarrow{a_n} q_m$$

is a run of B. Moreover, q_m is added to F' at line 7, and so $w \in L(B)$. □

Complexity. Observe that the algorithm processes pairs of transitions $(q_1, \alpha, q_2), (q_2, \beta, q_3)$, where (q_1, α, q_2) comes from W and (q_2, β, q_3) from δ (lines 8, 10, 14). Since every transition is removed from W at most once, the algorithm processes at most $|Q|^2 \cdot |\Sigma| \cdot |\delta|$ pairs. The runtime is dominated by the processing of the pairs, and so it is $\mathcal{O}(|Q|^2 \cdot |\Sigma| \cdot |\delta|)$.

2.4 Conversion algorithms between regular expressions and automata

To convert regular expressions to automata and vice versa we use NFA-regs as introduced in Definition 2.6. Both NFA-ε's and regular expressions can be seen as subclasses of NFA-regs: an NFA-ε is an NFA-reg whose transitions are labeled by letters or by ε, and a regular expression r "is" the NFA-reg A_r having two states, the one initial and the other final, and a single transition labeled r leading from the initial to the final state.

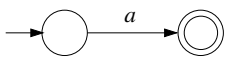
We present algorithms that, given an NFA-reg belonging to one of this subclasses, produces a sequence of NFA-regs, each one recognizing the same language as its predecessor in the sequence, and ending in an NFA-reg of the other subclass.

2.4.1 From regular expressions to NFA-ε's

Given a regular expression s over alphabet Σ, it is convenient do some preprocessing by exhaustively applying the following rewrite rules:

$$\begin{array}{rcl@{\qquad\qquad}rcl} \emptyset \cdot r & \rightsquigarrow & \emptyset & r \cdot \emptyset & \rightsquigarrow & \emptyset \\ r + \emptyset & \rightsquigarrow & r & \emptyset + r & \rightsquigarrow & r \\ \emptyset^* & \rightsquigarrow & \varepsilon & & & \end{array}$$

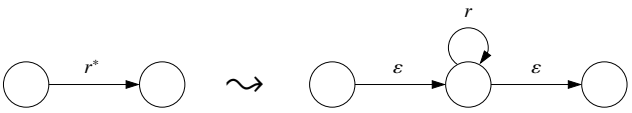
Since the left- and right-hand-sides of each rule denote the same language, the result of the preprocessing is a regular expression for the same language as the original one. Moreover, if r is the resulting regular expression, then either $r = \emptyset$, or r does not contain any occurrence of the $\emptyset$ symbol.



Automaton for the regular expression a, where $a \in \Sigma \cup \{\varepsilon\}$

Rule for concatenation

Rule for choice



Rule for Kleene iteration

Figure 2.4: Rules converting a regular expression given as NFA-reg into an NFA-ε.

In the former case, we can directly produce an NFA-ε. In the second, we transform the NFA-reg A_r into an equivalent NFA-ε by exhaustively applying the transformation rules of Figure 2.4.

It is easy to see that each rule preserves the recognized language (i.e., the NFA-regs before and after the application of the rule recognize the same language). Moreover, since each rule splits a regular expression into its constituents, we eventually reach an NFA-reg to which no rule can be applied. Furthermore, since the initial regular expression does not contain any occurrence of the $\emptyset$ symbol, this NFA-reg is necessarily an NFA-ε.

Example 2.8 Consider the regular expression $(a^*b^* + c)^*d$. The result of applying the transformation rules is shown in Figure 2.5 on page 23. □

Complexity. It follows immediately from the rules that the final NFA-ε has the two states of A_r plus one state for each occurrence of the concatenation or the Kleene iteration operators in r. The number of transitions is linear in the number of symbols of r. The conversion runs in linear time.

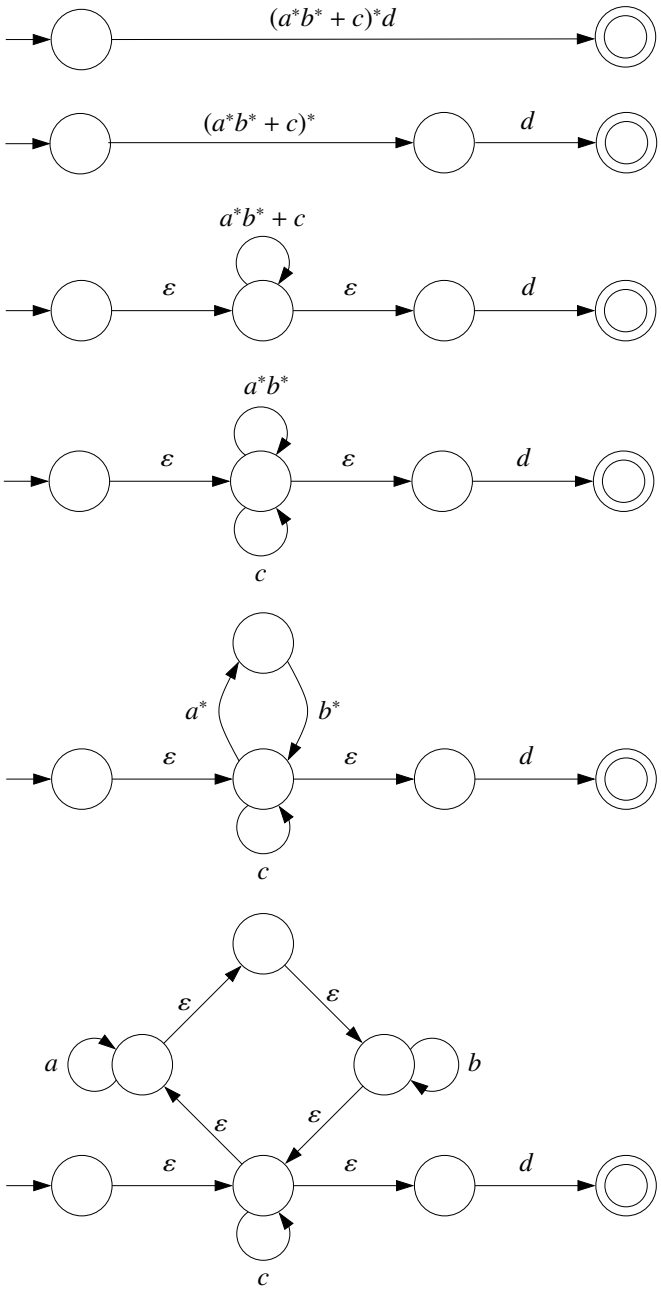
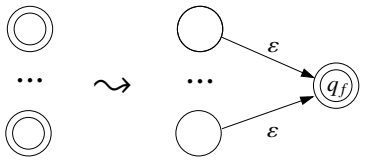


Figure 2.5: The result of converting $(a^*b^* + c)^*d$ into an NFA-ε.

2.4.2 From NFA-ε's to regular expressions

Given an NFA-ε A, we transform it into the NFA-reg A_r for some regular expression r. It is again convenient to apply some preprocessing to guarantee that the NFA-ε has one single final state, and that this final state has no outgoing transition:

- Add a new state q_f and ε-transitions leading from each final state to q_f, and replace the set of final states by $\{q_f\}$. Graphically:



Rule 1

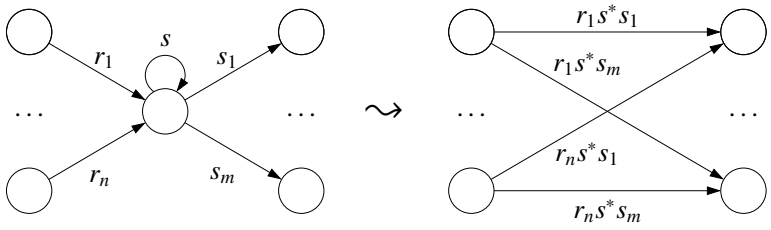
After preprocessing, the algorithm runs in phases. Each phase consist of two steps. The first step yields an automaton with at most one transition between any two given states:

- Repeat exhaustively: replace a pair of transitions (q, r_1, q'), (q, r_2, q') by a single transition $(q, r_1 + r_2, q')$.

Rule 2

The second step reduces the number of states by one, unless the only states left are the initial state and the final state q_f.

- Pick a non-final and non-initial state q, and *shortcut* it: If q has a self-loop (q, r, q)[1], replace each pair of transitions (q', s, q), (q, t, q'') (where $q' \neq q \neq q''$, but possibly $q' = q''$) by a shortcut (q', sr^*t, q''); otherwise, replace it by the shortcut (q', st, q''). After shortcutting all pairs, remove q. Graphically:



Rule 3

[1]Notice that it can have at most one, because otherwise we would have two parallel edges.

At the end of the last phase we have now an NFA-reg with exactly two states, the initial state q_0 (which is non-final) and the final state q_f. Moreover, q_f has no outgoing transitions, because it was initially so and the application of the rules cannot change it. After apply Rule 2 exhaustively to guarantee that there is at most one transition from q_0 to itself, and from q_0 to q_f, we are left with an NFA-reg having at most two transitions left: a transition (q_0, r_1, q_f), plus possibly a self-loop (q_0, r_2, q_0). If there is no self-loop, then we are done, the NFA-reg is A_{r_1}. Otherwise we apply some post-processing:

- If the automaton has a self-loop (q_0, r_2, q_0), remove it, and replace the transition (q_0, r_1, q_f) by $(q_0, r_2^* r_1, q_f)$. The resulting automaton is $A_{r_2^* r_1}$.

Rule 4

The complete algorithm is:

NFAtoRE(A)
Input: NFA-ε $A = (Q, \Sigma, \delta, q_0, F)$
Output: regular expression r with $L(r) = L(A)$

```
1  apply Rule 1
2  while Q' \ (F ∪ {q0}) ≠ ∅ do
3     pick q from Q \ (F ∪ {q0})
4     apply exhaustively Rule 2
5     apply Rule 3 to q
6  apply exhaustively Rule 2
7  if there is a transition from q0 to itself then apply Rule 4
8  return the label of the (unique) transition
```

Example 2.9 Consider the automaton of Figure 2.6(a) on page 26. It is a DFA for the language of all words over the alphabet $\{a, b\}$ that contain an even number of a's and an even number of b's. (The DFA is in the states on the left, respectively on the right, if it has read an even, respectively an odd, number of a's. Similarly, it is in the states at the top, respectively at the bottom, if it has read an even, respectively an odd, number of b's.) The rest of the figure shows some snapshots of the run of *NFAtoRE*() on this automaton. Snapshot (b) is taken right after applying rule 1. Snapshots (c) to (e) are taken after each execution of the body of the while loop. Snapshot (f) shows the final result. □

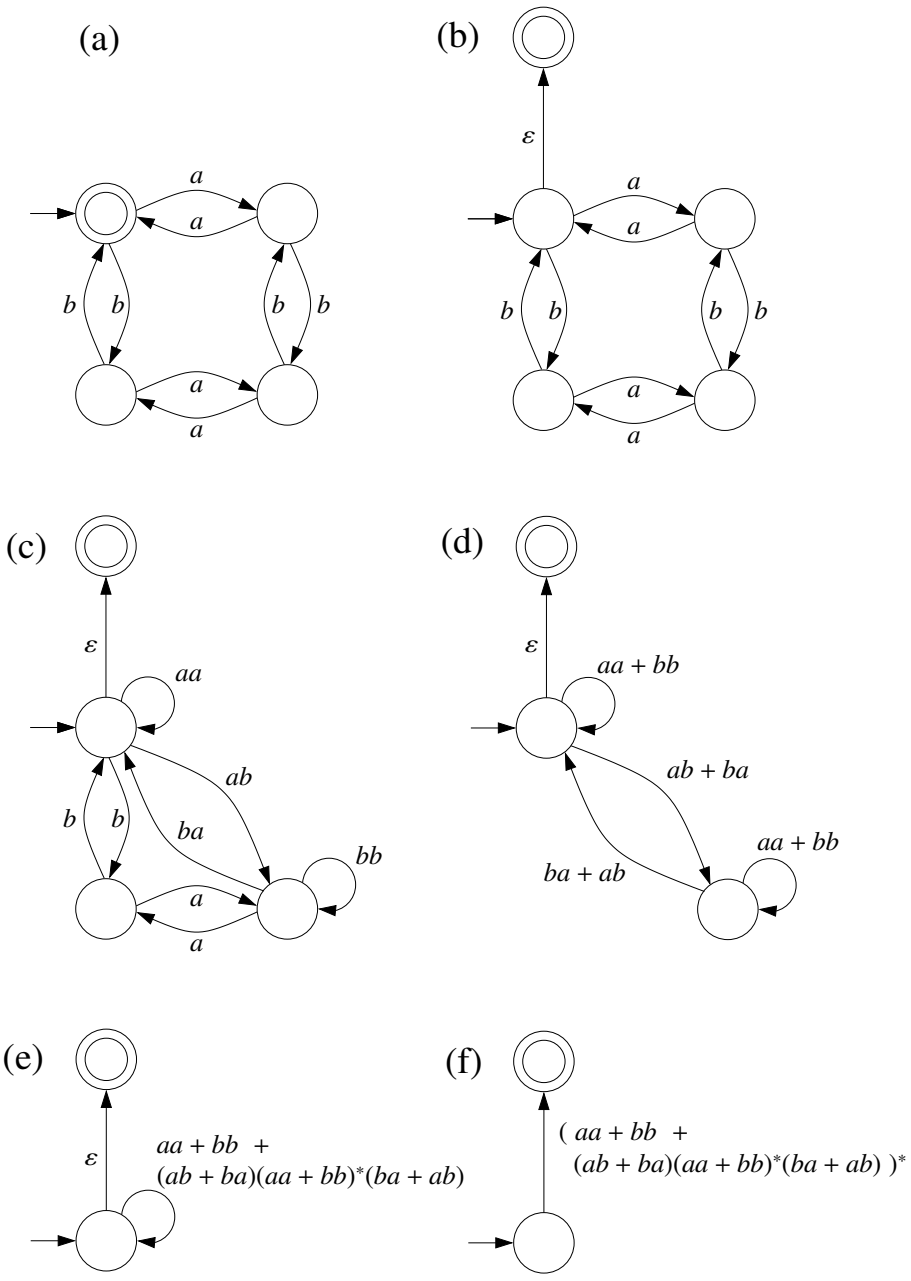


Figure 2.6: Run of *NFA-εtoRE*() on a DFA

Complexity. The complexity of this algorithm depends on the data structure used to store regular expressions. If regular expresions are stored as strings or trees (following the syntax tree of the expression), then the complexity can be exponential. To see this, consider for each $n \geq 1$ the NFA $A = (Q, \Sigma, \delta, q_0, F)$ where $Q = \{q_0, \ldots, q_{n-1}\}$, $\Sigma = \{a_{ij} \mid 0 \leq, i, j \leq n-1\}$, $\delta = \{(q_i, a_{ij}, q_j) \mid 0 \leq, i, j \leq n-1\}$, and $F = \{q_0\}$. By symmetry, the runtime of the algorithm is independent of the order in which states are eliminated. Consider the order $q_1, q_2, \ldots, q_{n-1}$. It is easy to see that after eliminating the state q_i the NFA-reg contains some transitions labeled by regular expressions with 3^i occurrences of letters. The exponental blowup cannot be avoided: It can be shown that every regular expression recognizing the same language as A contains at least $2^{(n-1)}$ occurrences of letters.

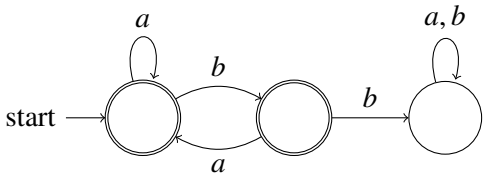
If regular expressions are stored as acyclic directed graphs (the result of sharing common subexpressions in the syntax tree), then the algorithm works in polynomial time, because the label for a new transition is obtained by concatenating aor starring already computed labels.

2.5 A Tour of Conversions

We present an example illustrating all conversions of this chapter. We start with the DFA of Figure 2.6(a) recognizing the words over $\{a, b\}$ with an even number of a's and an even number of b's. The figure converts it into a regular expression. Now we convert this expression into a NFA-ε: Figure 2.7 on page 28 shows four snapshots of the process of applying rules 1 to 4. In the next step we convert the NFA-ε into an NFA. The result is shown in Figure 2.8 on page 29. Finally, we transform the NFA into a DFA by means of the subset construction. The result is shown in Figure 2.9. Observe that we do not recover the DFA we started with, but another one recognizing the same language. A last step allowing us to close the circle is presented in the next chapter.

Exercises

Exercise 1 Transform this DFA



into an equivalent regular expression, then transform this expression into an NFA (with ε-transitions), remove the ε-transitions, and determinize the automaton.

Exercise 2 Prove or disprove: the languages of the regular expressions $(1 + 10)^*$ and $1^*(101^*)^*$ are equal.

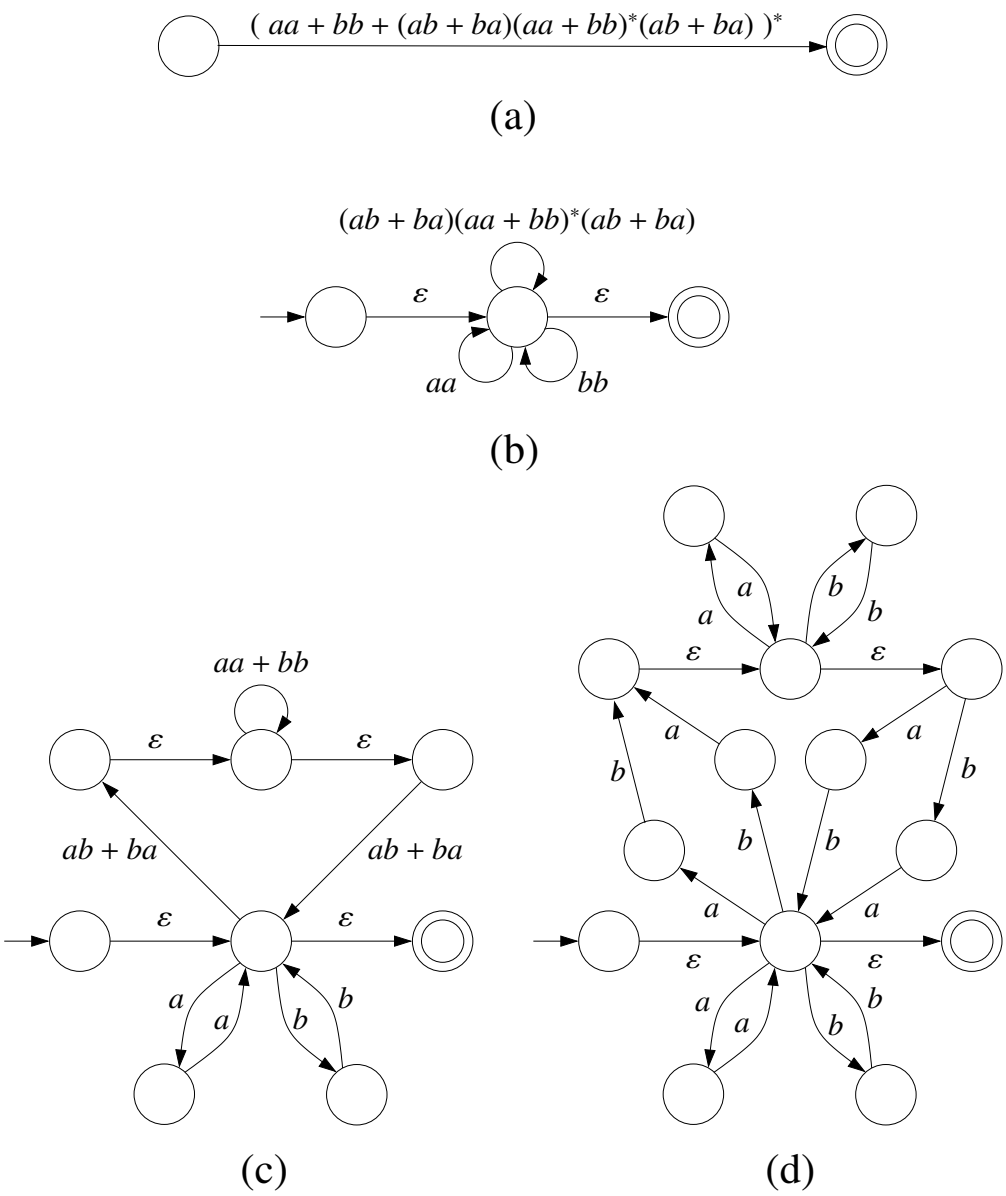


Figure 2.7: Constructing a NFA-ε for $(aa + bb + (ab + ba)(aa + bb)^*(ab + ba))^*$

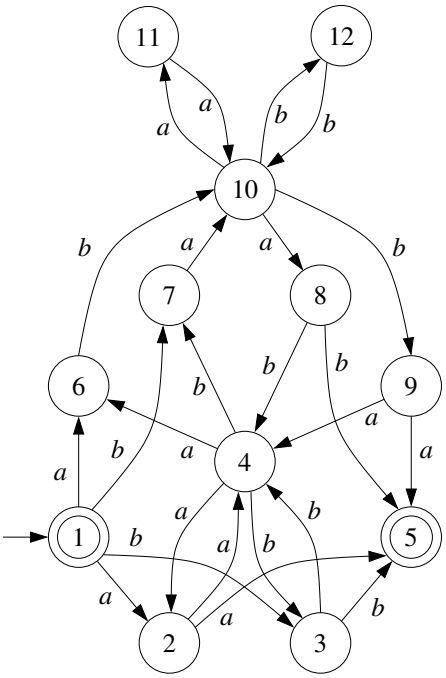


Figure 2.8: NFA for the NFA-ε of Figure 2.7(d)

Exercise 3 Give a regular expression for the words over $\{0, 1\}$ that do not contain 010 as subword.

Exercise 4 Which regular expressions r satisfy the implication $(rr = r \Rightarrow r = r^*)$?

Exercise 5 (R. Majumdar) Consider a deck of cards (with arbitrary many cards) in which black and red cards alternate, and the top card is black. Cut the deck at any point into two piles, and then perform a riffle (also called a dovetail shuffle) to yield a new deck. E.g., we can cut a deck with six cards 123456 into two piles 12 and 3456, and the riffle yields 132456 or 314256, depending on the pile we start the riffle with. Now, take thecards from the new deck two at a time (if the riffle yields 132456, then this exposes the cards 3, 4, and 6; if it yields 314256, then the cards 1, 2, and 6). Prove with the help of regular expressions that all the exposed cards have the same color.

Hint: The expression $(BR)^*(\epsilon + B)$ represents the possible initial decks. Give a regular expression for the set of possible decks after the riffle.

Exercise 6 Extend the syntax and semantics of regular expressions as follows: If r and r' are regular expressions over Σ, then $\overline{r}$ and $r \cap r'$ are also regular expressions, where $L(r) = \overline{L(r)}$ and $L(r \cap r') = L(r) \cap L(r')$. A language $L \subseteq \Sigma^*$ is called *star-free* if there exists an extended regular expression r *without any occurrence of a Kleene star operation* such that $L = L(r)$. For example,

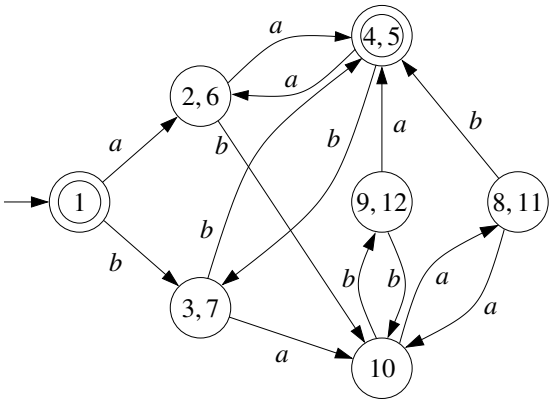


Figure 2.9: DFA for the NFA of Figure 2.8

Σ^* is star-free, because $\Sigma^* = L(\overline{\emptyset})$. Show that $L((01 + 10)^*)$ is star-free.

Exercise 7 Prove or disprove: Every regular language is recognized by a NFA ...

- ... having one single final state.
- ... whose states are all final.
- ... whose initial state has no incoming transitions.
- ... whose final state has no outgoing transitions.

Which of the above hold for DFAs?

Exercise 8 Given a language L, let L_{pref} and L_{suf} denote the languages of all prefixes and all suffixes, respectively, of words in L. E.g. for $L = \{abc, d\}$, we have $L_{\text{pref}} = \{abc, ab, a, \varepsilon, d\}$ and $L_{\text{suf}} = \{abc, bc, c, \varepsilon, d\}$.

1. Given an automaton A, construct automata A_{pref} and A_{suf} so that $\mathcal{L}(A_{\text{pref}}) = \mathcal{L}(A)_{\text{pref}}$ and $\mathcal{L}(A_{\text{suf}}) = \mathcal{L}(A)_{\text{suf}}$.
2. Consider the regular expression $r = (ab + b)^*c$. Give a regular expression r_{pref} so that $L(r_{\text{pref}}) = L(r_{\text{pref}})$.
3. More generally, give an algorithm that receives a regular expression r as input and returns a regular expression r_{pref} so that $L(r_{\text{pref}}) = L(r)_{\text{pref}}$.

Exercise 9 Recall that a nondeterministic automaton A accepts a word w if at least one of the runs of A on w is accepting. This is sometimes called the *existential* accepting condition. Consider the variant in which A accepts w if *all* runs of A on w are accepting (in particular, if A has no run on w then it accepts w). This is called the *universal* accepting condition. Notice that a DFA accepts the same language with both the existential and the universal accepting conditions.

Give an algorithm that transforms an automaton with universal accepting condition into a DFA recognizing the same language.

Exercise 10 Consider the family L_n of languages over the alphabet $\{0, 1\}$ given by $L_n = \{ww \in \Sigma^{2n} \mid w \in \Sigma^n\}$.

- Give an automaton of size $O(n)$ with universal accepting condition that recognizes L_n.
- Prove that every NFA (and so in particular every DFA) recognizing L_n has at least $O(2^n)$ states.

Exercise 11 The existential and universal accepting conditions can be combined, yielding *alternating automata*. The states of an alternating automaton are partitioned into *existential* and *universal* states. An existential state q accepts a word w (i.e., $w \in L(q)$) if $w = \varepsilon$ and $q \in F$ or $w = aw'$ and *there exists* a transition (q, a, q') such that q' accepts w'. A universal state q accepts a word w if $w = \varepsilon$ and $q \in F$ or $w = aw'$ and *for every* transition (q, a, q') the state q' accepts w'. The language recognized by an alternating automaton is the set of words accepted by its initial state.

Give an algorithm that transforms an alternating automaton into a DFA recognizing the same language.

Exercise 12 Let L be an arbitrary language over a 1-letter alphabet. Prove that L^* is regular.

Exercise 13 In algorithm *NFAεtoNFA* for the removal of ϵ-transitions, no transition that has been added to the worklist, processed *and* removed from the worklist is ever added to the worklist again. However, transitions may be added to the worklist more than once. Give a NFA-ε A and a run of *NFAεtoNFA*(A) in which this happens.

Exercise 14 We say that $u = a_1 \cdots a_n$ is a *scattered subword* of w, denoted by $u \triangleleft w$, if there are words $w_0, \cdots, w_n \in \Sigma^*$ such that $w = w_0 a_1 w_1 a_2 \cdots a_n w_n$. The *upward closure* of a language L is the language $L\uparrow = \{u \in \Sigma^* \mid \exists w \in L : w \triangleleft u\}$. The *downward closure* of L is the language $L\downarrow := \{u \in \Sigma^* \mid \exists w \in L : u \triangleleft w\}$. Give algorithms that take a NFA A as input and return NFAs for $L(A)\uparrow$ and $L(A)\downarrow$, respectively.

Exercise 15 Algorithm *NFAtoRE* transforms a finite automaton into a regular expression representing the same language by iteratively eliminating states of the automaton. In this exercise we see that eliminating states corresponds to eliminating variables from a system of *language equations*.

1. Arden's Lemma states that given two languages $A, B \subseteq \Sigma^*$ with $\varepsilon \notin A$, the smallest language $X \subseteq \Sigma^*$ satisfying $X = AX \cup B$ (i.e., the smallest language L such that $AL \cup B$ is again L, where AL is the concatenation of A and L) is the language A^*B. Prove Arden's Lemma.

2. Consider the following system of equations, where the variables X, Y represent languages over the alphabet $\Sigma = \{a, b, c, d, e, f\}$, the operator $\cdot$ represents language concatenation, and $\cup$ represents set union.
$$\begin{aligned} X &= \{a\}X \cup \{b\} \cdot Y \cup \{c\} \\ Y &= \{d\}X \cup \{e\} \cdot Y \cup \{f\}. \end{aligned}$$
(Here $\{a\}X$ denotes the concatenation of the language $\{a\}$ containing only the word a and X.) This system has many solutions. For example, $X, Y = \Sigma^*$ is a solution. But there is a unique minimal solution, i.e., a solution contained in every other solution. Find the smallest solution with the help of Arden's Lemma.
Hint: In a first step, consider X not as a variable, but as a constant language, and solve the equation for Y using Arden's Lemma.

We can associate to any NFA $A = (Q, \Sigma, \delta, q_I, F)$ a system of linear equations as follows. We take as variables the states of the automaton, which we call here $X, Y, Z, \ldots$, with X as initial state. The system has an equation for each state of the form
$$X = \bigcup_{(X,a,Y)\in\delta} \{a\}Y$$
if $X \notin F$, or
$$X = \left(\bigcup_{(X,a,Y)\in\delta} \{a\} \cdot Y \right) \cup \{\varepsilon\}$$
if $X \in F$.

3. Consider the DFA of Figure 2.6(a). Let X, Y, Z, W be the states of the automaton, read from top to bottom and from left to right. The associated system of linear equations is
$$\begin{aligned} X &= \{a\}Y \cup \{b\}Z \cup \{\varepsilon\} \\ Y &= \{a\}X \cup \{b\}W \\ Z &= \{b\}X \cup \{a\}W \\ W &= \{b\}Y \cup \{a\}Z \end{aligned}$$
Calculate the solution of this linear system by iteratively eliminating variables. Start with Y, then eliminate Z, and finally W. Compare with the elimination procedure shown in Figure 2.6.

Exercise 16 Given $n \in \mathbb{N}_0$, let msbf(n) be the set of *most-significant-bit-first* encodings of n, i.e., the words that start with an arbitrary number of leading zeros, followed by n written in binary. For example:

$$\text{msbf}(3) = 0^*11 \quad \text{and} \quad \text{msbf}(9) = 0^*1001 \quad \text{msbf}(0) = 0^*.$$

Similarly, let lsbf(n) denote the set of *least-significant-bit-first* encodings of n, i.e., the set containing for each word $w \in \text{msbf}(n)$ its reverse. For example:

$$\text{lsbf}(6) = \mathcal{L}(0110^*) \quad \text{and} \quad \text{lsbf}(0) = \mathcal{L}(0^*).$$

1. Construct and compare DFAs recognizing the encodings of the even numbers $n \in \mathbb{N}_0$ w.r.t. the unary encoding, where n is encoded by the word 1^n, the msbf-encoding, and the lsbf-encoding.

2. Same for the set of numbers divisible by 3.

3. Give regular expressions corresponding to the languages in (a) and (b).

Chapter 3

Minimization and Reduction

In the previous chapter we showed through a chain of conversions that the two DFAs of Figure 3.1 recognize the same language. Obviously, the automaton on the left of the figure is better as a data structure for this language, since it has smaller size. A DFA (respectively, NFA) is *minimal* if

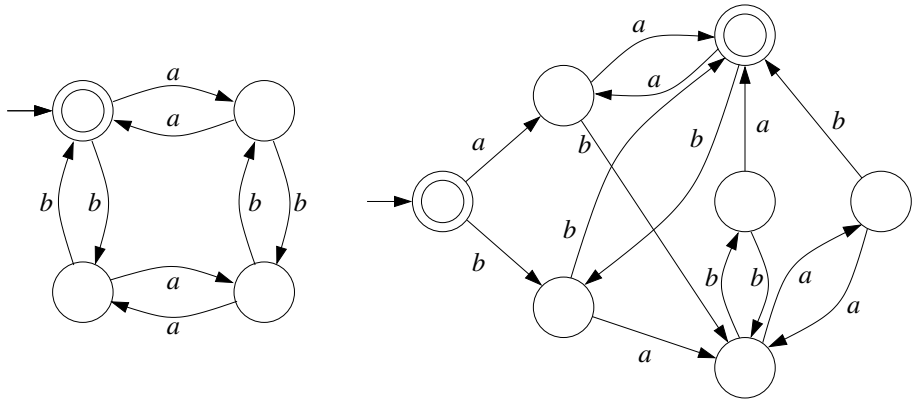


Figure 3.1: Two DFAs for the same language

no other DFA (respectively, NFA) recognizing the same language has fewer states. We show that every regular language has a unique minimal DFA up to isomorphism (i.e., up to renaming of the states). and present an efficient algorithm that "minimizes" a given DFA, i.e., converts it into the unique minimal DFA. In particular, the algorithm converts the DFA on the right of Figure 3.1 into the one on the left.

From a data structure point of view, the existence of a unique minimal DFA has two important consequences. First, as mentioned above, the minimal DFA is the one that can be stored with a minimal amount of memory. Second, the uniqueness of the minimal DFA makes it a *canonical* representation of a regular language. As we shall see, canonicity leads to a fast equality check: In order to decide if two regular languages are equal, we can construct their minimal DFAs, and check

if they are isomorphic .

In the second part of the chapter we show that, unfortunately, computing a minimal NFA is a PSPACE complete problem, for which no efficient algorithm is likely to exist. Moreover, the minimal NFA is not unique. However, we show that a generalization of the minimization algorithm for DFAs can be used to at least reduce the size of an NFA while preserving its language.

3.1 Minimal DFAs

We start with a simple but very useful definition.

Definition 3.1 *Given a language $L \subseteq \Sigma^*$ and $w \in \Sigma^*$, the* w-residual *of L is the language $L^w = \{u \in \Sigma^* \mid wu \in L\}$. A language $L' \subseteq \Sigma^*$ is a* residual *of L if $L' = L^w$ for at least one $w \in \Sigma^*$.*

A language may be infinite but have a finite number of residuals. An example is the language of the DFAs in Figure 3.1. Recall it is the language of all words over $\{a, b\}$ with an even number of a's and an even number of b's, which we denote in what follows by EE. The language has four residuals, namely EE, EO, OE, OO, where EO contains the words with an even number of a's and an odd number of b's, etc. For example, we have $EE^{\varepsilon} = EE$, $EE^{a} = EE^{aaa} = OE$, and $EE^{ab} = OO$.

There is a close connection between the states of a DA (not necessarily finite) and the residuals of its language. In order to formulate it we introduce the following definition:

Definition 3.2 *Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DA and let $q \in Q$. The* language recognized by *q, denoted by $L_A(q)$, or just $L(q)$ if there is no risk of confusion, is the language recognized by A with q as initial state, i.e., the language recognized by the DA $(Q, \Sigma, \delta, q, F)$.*

Figure 3.2 shows the result of labeling the states of the two DFAs of Figure 3.1 with the languages they recognize, which are residuals of EE.

Lemma 3.3 *Let L be a language and let $A = (Q, \Sigma, \delta, q_0, F)$ be a DA recognizing L.*

(1) Every residual of L is recognized by some state of A. Formally: for every $w \in \Sigma^*$ there is $q \in Q$ such that $L_A(q) = L^w$.

(2) Every state of A recognizes a residual of L. Formally: for every $q \in Q$ there is $w \in \Sigma^*$ such that $L_A(q) = L^w$.

Proof: (1) Let $w \in \Sigma^*$, and let q be the state reached by the unique run of A on w. Then a word $wu \in \Sigma^*$ is recognized by A iff u is recognized by A with q as initial state. So $L_A(q) = L^w$.

(2) Since A is in normal form, q can be reached from q_0 by at least a word w. So $L_A(q) = L^w$, and we are done. □

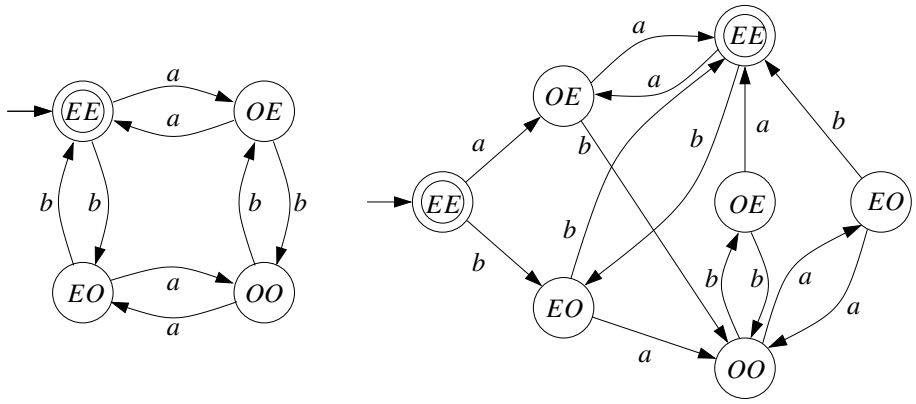


Figure 3.2: Languages recognized by the states of the DFAs of Figure 3.1.

The notion of residuals allows to define the *canonical deterministic automaton* for a language $L \subseteq \Sigma^*$.

Definition 3.4 *Let $L \subseteq \Sigma^*$ be a language. The* canonical DA *for L is the DA $C_L = (Q_L, \Sigma, \delta_L, q_{0L}, F_L)$, where:*

- *Q_L is the set of residuals of L; i.e., $Q_L = \{L^w \mid w \in \Sigma^*\}$;*
- *$\delta(K, a) = K^a$ for every $K \in Q_L$ and $a \in \Sigma$;*
- *$q_{0L} = L$; and*
- *$F_L = \{K \in Q_L \mid \varepsilon \in K\}$.*

Notice that the number of states of C_L is equal to the number of residuals of L, and both may be infinite.

Example 3.5 The canonical DA for the language EE is shown on the left of Figure 3.2. It has four states, corresponding to the four residuals of EE. Since, for instance, we have $EE^a = OE$, we have a transition labeled by a leading from EE to OE. □

It is intuitively clear, and easy to show, that the canonical DA for a language L recognizes L:

Proposition 3.6 *For every language $L \subseteq \Sigma^*$, the canonical DA for L recognizes L.*

Proof: Let C_L be the canonical DA for L. We prove $L(C_L) = L$.

Let $w \in \Sigma^*$. We prove by induction on $|w|$ that $w \in L$ iff $w \in L(C_L)$.

If $|w| = 0$ then $w = \varepsilon$, and we have

$$\begin{array}{lll} & \varepsilon \in L & (w = \epsilon) \\ \Leftrightarrow & L \in F_L & (\text{definition of } F_L) \\ \Leftrightarrow & q_{0L} \in F_L & (q_{0L} = L) \\ \Leftrightarrow & \varepsilon \in L(C_L) & (q_{0L} \text{ is the initial state of } C_L) \end{array}$$

If $|w| > 0$, then $w = aw'$ for some $a \in \Sigma$ and $w' \in \Sigma^*$, and we have

$$\begin{array}{lll} & aw' \in L & \\ \Leftrightarrow & w' \in L^a & (\text{definition of } L^a) \\ \Leftrightarrow & w' \in L(C_{L^a}) & (\text{induction hypothesis}) \\ \Leftrightarrow & aw' \in L(C_L) & (\delta_L(L, a) = L^a) \end{array}$$

□

We now prove that C_L is the unique minimal DFA recognizing a regular language L (up to isomorphism). The informal argument goes as follows. Since every DFA for L has *at least* one state for each residual and C_L has *exactly* one state for each residual, C_L is minimal, and every other minimal DFA for L also has exactly one state for each residual. But the transitions, initial and final states of a minimal DFA are completely determined by the residuals of the states: if state q recognizes residual R, then the a-transition leaving q necessarily leads to the state recognizing R^a; q is final iff $\epsilon \in R$, and q is initial iff $R = L$. So all minimal DFAs are isomorphic. A more formal proof looks as follows:

Theorem 3.7 *If L is regular, then C_L is the unique minimal DFA up to isomorphism recognizing L.*

Proof: Let L be a regular language, and let $A = (Q, \Sigma, \delta, q_0, F)$ be an arbitrary DFA recognizing L. By Lemma 3.3 the number the number of states of A is greater than or equal to the number of states of C_L, and so C_L is a minimal automaton for L. To prove uniqueness of the minimal automaton up to isomorphism, assume A is minimal. By Lemma 3.3(2), $\mathcal{L}_A$ is a mapping that assigns to each state of A a residual of L, and so $\mathcal{L}_A \colon Q \to Q_L$. We prove that $\mathcal{L}_A$ is an isomorphism. $\mathcal{L}_A$ is bijective because it is surjective (Lemma 3.3(2)), and $|Q| = |Q_L|$ (A is minimal by assumption). Moreover, if $\delta(q, a) = q'$, then $L_A(q') = (L_A(q))^a$, and so $\delta_L(L_A(q), a) = L_A(q')$. Also, $\mathcal{L}_A$ maps the initial state of A to the initial state of C_L: $L_A(q_0) = L = q_{0L}$. Finally, $\mathcal{L}_A$ maps final to final and non-final to non-final states: $q \in F$ iff $\varepsilon \in L_A(q)$ iff $L_A(q) \in F_L$. □

The following simple corollary is often useful to establish that a given DFA is minimal:

Corollary 3.8 *A DFA is minimal if and only if $L(q) \neq L(q')$ for every two distinct states q and q'.*

Proof: ($\Rightarrow$): By Theorem 3.7, the number of states of a minimal DFA is equal to the number of residuals of its language. Since every state of recognizes some residual, each state must recognize a different residual.

($\Leftarrow$): If all states of a DFA A recognize different languages, then, since every state recognizes some residual, the number of states of A is less than or equal to the number of residuals. So A has at most as many states as $C_{L(A)}$, and so it is minimal. □

3.2 Minimizing DFAs

We present an algorithm that converts a given DFA into (a DFA isomorphic to) the unique minimal DFA recognizing the same language. The algorithm first partitions the states of the DFA into *blocks*, where a block contains all states recognizing the same residual. We call this partition the *language partition*. Then, the algorithm "merges" the states of each block into one single state, an operation usually called *quotienting* with respect to the partition. Intuitively, this yields a DFA in which every state recognizes a different residual. These two steps are described in Section 3.2.1 and Section 3.2.2.

For the rest of the section we fix a DFA $A = (Q, \Sigma, \delta, q_0, F)$ recognizing a regular language L.

3.2.1 Computing the language partition

We need some basic notions on partitions. A *partition* of Q is a finite set $P = \{B_1, \ldots, B_n\}$ of nonempty subsets of Q, called *blocks*, such that $Q = B_1 \cup \ldots \cup B_n$, and $B_i \cap B_j = \emptyset$ for every $1 \leq i \neq j \leq n$. The block containing a state q is denoted by $[q]_P$. A partition P' *refines* or *is a refinement of* another partition P if every block of P' is contained in some block of P. If P' refines P and $P' \neq P$, then P is *coarser* than P'.

The *language partition*, denoted by P_ℓ, puts two states in the same block if and only if they recognize the same language (i.e, the same residual). Formally, $[q]_{P_\ell} = [q']_{P_\ell}$ if and only if $L(q) = L(q')$. To compute P_ℓ we iteratively refine an initial partition P_0 while maintaining the following invariant:

> Invariant: *States in different blocks recognize different languages.*

P_0 consists of two blocks containing the final and the non-final states, respectively (or just one of the two if all states are final or all states are nonfinal). That is, $P_0 = \{F, Q \setminus F\}$ if F and $Q \setminus F$ are nonempty, $P_0 = \{F\}$ if $Q \setminus F$ is empty, and $P_0 = \{Q \setminus F\}$ if F is empty. Notice that P_0 satisfies the invariant, because every state of F accepts the empty word, but no state of $Q \setminus F$ does.

A partition is refined by splitting a block into two blocks. To find a block to split, we first observe the following:

Fact 3.9 *If $L(q_1) = L(q_2)$, then $L(\delta(q_1, a)) = L(\delta(q_2, a))$ for every $a \in \Sigma$.*

Now, by contraposition, if $L(\delta(q_1, a)) \neq L(\delta(q_2, a))$, then $L(q_1) \neq L(q_2)$, or, rephrasing in terms of blocks: if $\delta(q_1, a)$ and $\delta(q_2, a)$ belong to different blocks, but q_1 and q_2 belong to the same block B, then B can be split, because q_1 and q_2 can be put in different blocks while respecting the invariant.

Definition 3.10 *Let B, B' be (not necessarily distinct) blocks of a partition P, and let $a \in \Sigma$. The pair (a, B')* splits *B if there are $q_1, q_2 \in B$ such that $\delta(q_1, a) \in B'$ and $\delta(q_2, a) \notin B'$. The result of the split is the partition $Ref_P[B, a, B'] = (P \setminus \{B\}) \cup \{B_0, B_1\}$, where*

$$B_0 = \{q \in B \mid \delta(q, a) \notin B'\} \text{ and } B_1 = \{q \in B \mid \delta(q, a) \in B'\} .$$

A partition is unstable *if it contains blocks B, B' such that (a, B') splits B for some $a \in \Sigma$, and* stable *otherwise.*

The partition refinement algorithm *LanPar*(A) iteratively refines the initial partition of A until it becomes stable. The algorithm terminates because every iteration increases the number of blocks by one, and apartition can have at most $|Q|$ blocks.

LanPar(A)
Input: DFA $A = (Q, \Sigma, \delta, q_0, F)$
Output: The language partition P_ℓ.

```
1  if F = ∅ or Q \ F = ∅ then return {Q}
2  else P ← {F, Q \ F}
3  while P is unstable do
4     pick B, B' ∈ P and a ∈ Σ such that (a, B') splits B
5     P ← Ref_P[B, a, B']
6  return P
```

Example 3.11 Figure 3.3 shows a run of *LanPar*() on the DFA on the right of Figure 3.1. States that belong to the same block are given the same color. The initial partition, shown at the top, consists of the yellow and the pink states. The yellow block and the letter a split the pink block into the green block (pink states with an a-transition to the yellow block) and the rest (pink states with an a-transition to other blocks), which stay pink. In the final step, the green block and the letter b split the pink block into the magenta block (pink states with a b transition into the green block) and the rest, which stay pink. □

We prove in two steps that *LanPar*(A) computes the language partition. First, we show that it computes the *coarsest stable refinement of* P_0, denoted by *CSR*; in other words we show that after termination the partition P is coarser than every other stable refinement of P_0. Then we prove that *CSR* is equal to P_ℓ.

Lemma 3.12 *LanPar(A) computes CSR.*

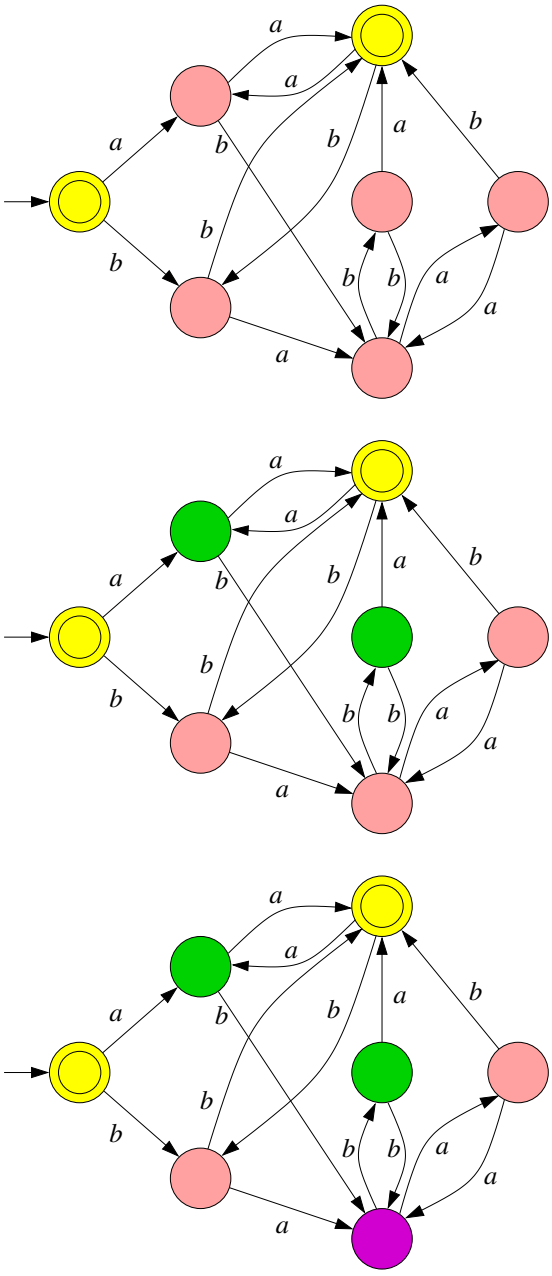


Figure 3.3: Computing the language partition for the DFA on the left of Figure 3.1

Proof: *LanPar*(A) clearly computes a stable refinement of P_0. We prove that after termination P is coarser than any other stable refinement of P_0, or, equivalently, that every stable refinement of P_0 refines P. Actually, we prove more: we show that this is in fact an invariant of the program: it holds not only after termination, but at any time.

Let P' be an arbitrary stable refinement of P_0. Initially $P = P_0$, and so P' refines P. Now, we show that if P' refines P, then P' also refines $Ref_P[B, a, B']$. For this, let q_1, q_2 be two states belonging to the same block of P'. We show that they belong to the same block of $Ref_P[B, a, B']$. Assume the contrary. Since the only difference between P and $Ref_P[B, a, B']$ is the splitting of B into B_0 and B_1, exactly one of q_1 and q_2, say q_1, belongs to B_0, and the other belongs to B_1. So there exists a transition $(q_2, a, q_2') \in \delta$ such that $q_2' \in B'$. Since P' is stable and q_1, q_2 belong to the same block of P', there is also a transition $(q_1, a, q_1') \in \delta$ such that $q_1' \in B'$. But this contradicts $q_1 \in B_0$. □

Theorem 3.13 *CSR is equal to* $P_\ell =$.

Proof: The proof has three parts:

(a) P_ℓ refines P_0. Obvious.

(b) P_ℓ is stable. By Fact 3.9, if two states q_1, q_2 belong to the same block of P_ℓ, then $\delta(q_1, a), \delta(q_2, a)$ also belong to the same block, for every a. So no block can be split.

(c) Every stable refinement P of P_0 refines P_ℓ. Let q_1, q_2 be states belonging to the same block B of P. We prove that they belong to the same block of P_ℓ, i.e., that $L(q_1) = L(q_2)$. By symmetry, it suffices to prove that, for every word w, if $w \in L(q_1)$ then $w \in L(q_2)$. We proceed by induction on the length of w. If $w = \varepsilon$ then $q_1 \in F$, and since P refines P_0 , we have $q_2 \in F$, and so $w \in L(q_2)$. If $w = aw'$, then there is $(q_1, a, q_1') \in \delta$ such that $w' \in L(q_1')$. Let B' be the block containing q_1'. Since P is stable, B' does not split B, and so there is $(q_2, a, q_2') \in \delta$ such that $q_2' \in B'$. By induction hypothesis, $w' \in L(q_1')$ iff $w' \in L(q_2')$. So $w' \in L(q_2')$, which implies $w \in L(q_2)$.

□

3.2.2 Quotienting

It remains to define the quotient of A with respect to a partition. The states of the quotient are the blocks of the partition, and there is a transition (B, a, B') from block B to block B' if A contains some transition (q, a, q') for states q and Q' belonging to B and B', respectively. Formally:

Definition 3.14 *The* quotient *of A with respect to a partition P is the DFA* $A/P = (Q_P, \Sigma, \delta_P, q_{0P}, F_P)$ *where*

- Q_P *is the set of blocks of P;*

- $(B, a, B') \in \delta_P$ *if* $(q, a, q') \in \delta$ *for some* $q \in B$, $q' \in B'$;
- q_{0P} *is the block of* P *containing* q_0; *and*
- F_P *is the set of blocks of* P *that contain at least one state of* F.

Example 3.15 Figure 3.4 shows on the right the result of quotienting the DFA on the left with respect to its language partition. The quotient has as many states as colors, and it has a transition between two colors (say, an a-transition from pink to magenta) if the DFA on the left has such a transition. □

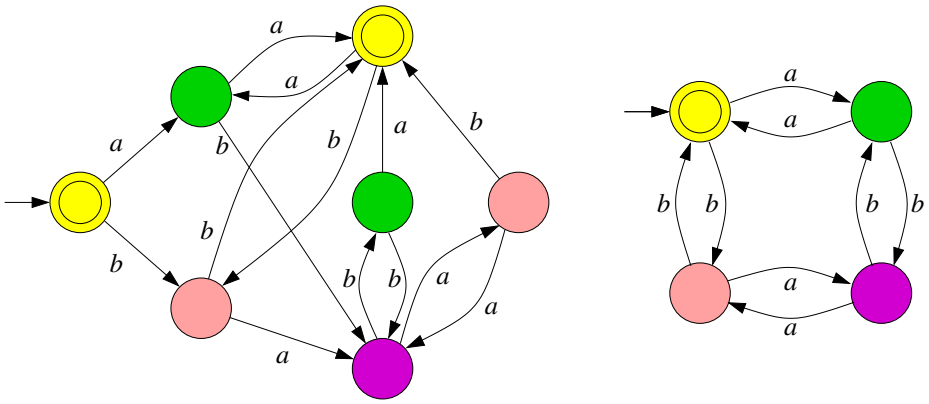


Figure 3.4: Quotient of a DFA with respect to its language partition

We show that A/P_ℓ, the quotient of A with respect to the language partition, is the minimal DFA for L. The main part of the argument is contained in the following lemma. Loosely speaking, it says that any partition in which states of the same block recognize the same language "is good" for quotienting, because the quotient recognizes the same language as the original automaton.

Lemma 3.16 *Let* P *be a refinement of* P_ℓ, *let* q *be a state of* A, *and let* B *be the block of* P *containing* q. *Then* $L_A(q) = L_{A/P}(B)$. *In particular,* $L(A) = L(A/P)$.

Proof: We prove that for every $w \in \Sigma^*$ we have $w \in L_A(q)$ iff $w \in L_{A/P}(B)$.The proof is by induction on $|w|$.

$|w| = 0$. Then $w = \varepsilon$ and we have

$$\begin{array}{lll} & \varepsilon \in L_A(q) & \\ \text{iff} & q \in F & \\ \text{iff} & B \subseteq F & (P \text{ refines } P_\ell, \text{ and so also } P_0) \\ \text{iff} & B \in F_P & \\ \text{iff} & \varepsilon \in L_{A/P}(B) & \end{array}$$

$|w| > 0$. Then $w = aw'$ for some $a \in \Sigma$. So there is a transition $(q, a, q') \in \delta$ such that $w' \in L_A(q')$. Let B' be the block containing q'. By the definition of A/P we have $(B, a, B') \in \delta_P$, and so:

$$\begin{array}{lll} & aw' \in L_A(q) & \\ \text{iff} & w' \in L_A(q') & (A \text{ is a DFA}) \\ \text{iff} & w' \in L_{A/P}(B') & (\text{induction hypothesis}) \\ \text{iff} & aw' \in L_{A/P}(B) & (\ (B, a, B') \in \delta_P\ ) \end{array}$$

□

Proposition 3.17 *The quotient A/P_ℓ is the minimal DFA for L.*

Proof: By Lemma 3.16, the states of A/P_ℓ recognize residuals of L. Moreover, two states of A/P_ℓ recognize different residuals by definition. So A/P_ℓ has as many states as residuals, and we are done. □

3.3 Reducing NFAs

There is no canonical minimal NFA for a given regular language. The simplest witness of this fact is the language aa^*, which is recognized by the two non-isomorphic, minimal NFAs of Figure 3.5.

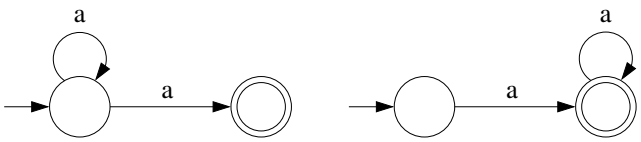


Figure 3.5: Two minimal NFAs for aa^*.

Moreover, computing any of the minimal NFAs equivalent to a given NFA is computationally hard. Recall that the *universality problem* is PSPACE-complete: given a NFA A over an alphabet Σ, decide whether $L(A) = \Sigma^*$. Using this result, we can easily prove that deciding the existence of a small NFA for a language is PSPACE-complete.

Theorem 3.18 *The following problem is PSPACE-complete: given a NFA A and a number $k \geq 1$, decide if there exists an NFA equivalent to A having at most k states.*

Proof: To prove membership in PSPACE, observe first that if A has at most k states, then we can answer A. So assume that A has more than k states. We use NPSPACE = PSPACE = co-PSPACE. Since PSPACE = co-PSPACE, it suffices to give a procedure to decide if no NFA with at most k states is equivalent to A. For this we construct all NFAs with at most k states (over the same alphabet as A), reusing the same space for each of them, and check that none of them is equivalent

to A. Now, since NPSPACE=PSPACE, it suffices to exhibit a nondeterministic algorithm that, given a NFA B with at most k states, checks that B is not equivalent to A (and runs in polynomial space). The algorithm nondeterministically guesses a word, one letter at a time, while maintaining the sets of states in both A and B reached from the initial states by the word guessed so far. The algorithm stops when it observes that the current word is accepted by exactly one of A and B.

PSPACE-hardness is easily proved by reduction from the universality problem. If an NFA is universal, then it is equivalent to an NFA with one state, and so, to decide if a given NFA A is universal we can proceed as follows: Check first if A accepts all words of length 1. If not, then A is not universal. Otherwise, check if some NFA with one state is equivalent to A. If not, then A is not universal. Otherwise, if such a NFA, say B, exists, then, since A accepts all words of length 1, B is the NFA with one final state and a loop for each alphabet letter. So A is universal. □

However, we can reuse part of the theory for the DFA case to obtain an efficient algorithm to possibly reduce the size of a given NFA.

3.3.1 The reduction algorithm

We fix for the rest of the section an NFA $A = (Q, \Sigma, \delta, q_0, F)$ recognizing a language L. A look at Definition 3.14 and Lemma 3.16 shows that the definition of the quotient automaton extends to NFA, and that Lemma 3.16 still holds for NFAs with exactly the same proof. So $L(A) = L(A/P)$ holds for every refinement P of P_ℓ, and so *any* refinement of P_ℓ can be used to reduce A. The largest reduction is obtained for $P = P_\ell$, but P_ℓ is hard to compute for NFA. On the other extreme, the partition that puts each state in a separate block is always a refinement of P_ℓ, but it does not provide any reduction.

To find a reasonable trade-off we examine again Lemma 3.12, which proves that *LanPar*(A) computes *CSR* for deterministic automata. Its proof only uses the following property of stable partitions: if q_1, q_2 belong to the same block of a stable partition and there is a transition $(q_2, a, q_2') \in \delta$ such that $q_2' \in B'$ for some block B', then there is also a transition $(q_1, a, q_1') \in \delta$ such that $q_1' \in B'$. We extend the definition of stability to NFAs so that stable partitions still satisfy this property: we just replace condition

$$\delta(q_1, a) \in B' \text{ and } \delta(q_2, a) \notin B'$$

of Definition 3.10 by

$$\delta(q_1, a) \cap B' \neq \emptyset \text{ and } \delta(q_2, a) \cap B' = \emptyset.$$

Definition 3.19 (Refinement and stability for NFAs) *Let B, B' be (not necessarily distinct) blocks of a partition P, and let $a \in \Sigma$. The pair (a, B')* splits *B if there are $q_1, q_2 \in B$ such that $\delta(q_1, a) \cap B' \neq \emptyset$ and $\delta(q_2, a) \cap B' = \emptyset$. The result of the split is the partition $Ref_P^{NFA}[B, a, B'] = (P \setminus \{B\}) \cup \{B_0, B_1\}$, where*

$$B_0 = \{q \in B \mid \delta(q, a) \cap B' = \emptyset\} \text{ and } B_1 = \{q \in B \mid \delta(q, a) \cap B' \neq \emptyset\}\,.$$

A partition is unstable *if it contains blocks B, B' such that B' splits B, and* stable *otherwise.*

Using this definition we generalize *LanPar*(*A*) to NFAs in the obvious way: allow NFAs as inputs, and replace Ref_P by Ref_P^{NFA} as new notion of refinement. Lemma 3.12 still holds: the algorithm still computes *CSR*, but with respect to the new notion of refinement. Notice that in the special case of DFAs it reduces to *LanPar*(*A*), because Ref_P and Ref_P^{NFA} coincide for DFAs.

CSR(A)
Input: NFA $A = (Q, \Sigma, \delta, q_0, F)$
Output: The partition *CSR* of A.

1 **if** $F = \emptyset$ or $Q \setminus F = \emptyset$ **then return** $\{Q\}$
2 **else** $P \leftarrow \{F, Q \setminus F\}$
3 **while** P is unstable **do**
4 pick $B, B' \in P$ and $a \in \Sigma$ such that (a, B') splits B
5 $P \leftarrow Ref_P^{NFA}[B, a, B']$
6 **return** P

In the case of DFAs we had Theorem 3.13, stating that *CSR* is equal to P_ℓ. The theorem does not hold anymore for NFAs, as we will see later. However, part (c) of the proof, which showed that *CSR* refines P_ℓ, still holds, with exactly the same proof. So we get:

Theorem 3.20 *Let $A = (Q, \Sigma, \delta, q_0, F)$ be a NFA. The partition CSR refines P_ℓ.*

Now, Lemma 3.16 and Theorem 3.13 lead to the final result:

Corollary 3.21 *Let $A = (Q, \Sigma, \delta, q_0, F)$ be a NFA. Then $L(A/CSR) = L(A)$.*

Example 3.22 Consider the NFA at the top of Figure 3.6. *CSR* is the partition indicated by the colors. A possible run of CSR(A) is graphically represented at the bottom of the figure as a tree. Initially we have the partition with two blocks shown at the top of the figure: the block $\{1, \ldots, 14\}$ of non-final states and the block $\{15\}$ of final states. The first refinement uses $(a, \{15\})$ to split the block of non-final states, yielding the blocks $\{1, \ldots, 8, 11, 12, 13\}$ (no a-transition to $\{15\}$) and $\{9, 10, 14\}$ (an a-transition to $\{15\}$). The leaves of the tree are the blocks of *CSR*.

In this example we have $CSR \neq P_\ell$. For instance, states 3 and 5 recognize the same language, namely $(a + b)^*aa(a + b)^*$, but they belong to different blocks of *CSR*.

The quotient automaton is shown in Figure 3.7. □

We finish the section with a remark.

Remark 3.23 If A is an NFA, then A/P_ℓ may not be a minimal NFA for L. The NFA of Figure 3.8 is an example: all states accept different languages, and so $A/P_\ell = A$, but the NFA is not minimal, since, for instance, the state at the bottom can be removed without changing the language.

It is not difficult to show that if two states q_1, q_2 belong to the same block of *CSR*, then they not only recognize the same language, but also satisfy the following far stronger property: for every

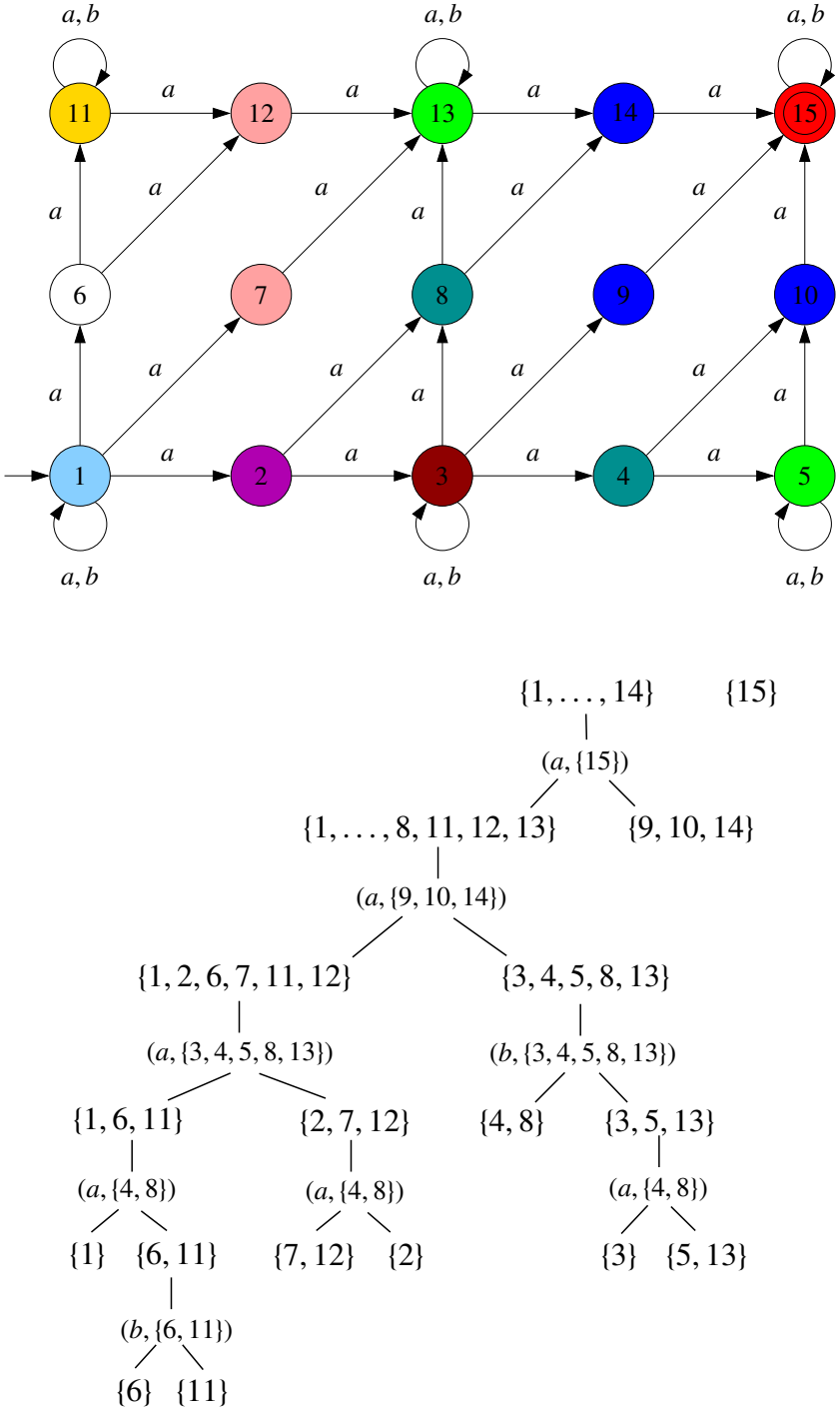


Figure 3.6: An NFA and a run of *CSR*() on it.

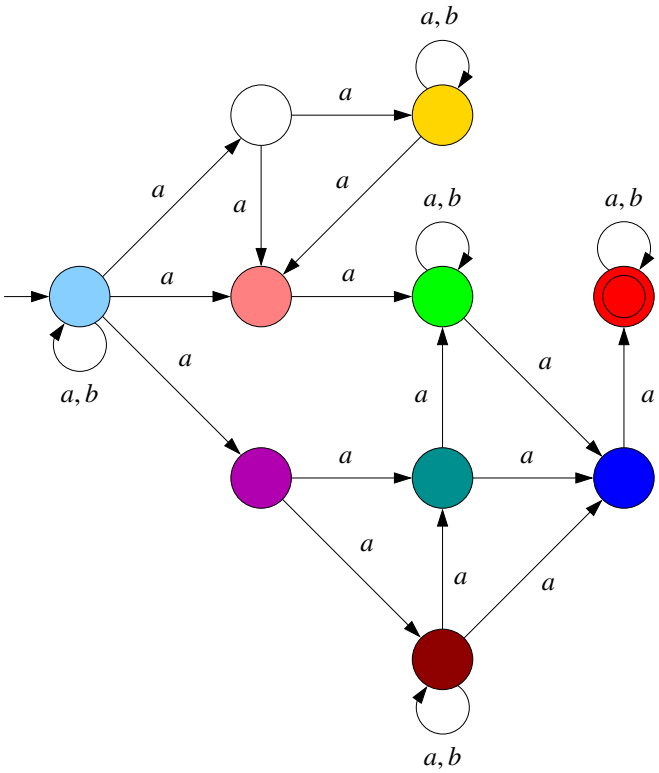


Figure 3.7: The quotient of the NFA of Figure 3.6.

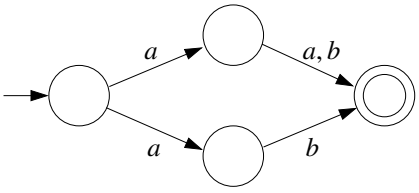


Figure 3.8: An NFA A such that A/P_ℓ is not minimal.

$a \in \Sigma$ and for every $q'_1 \in \delta(q_1, a)$, there exists $q'_2 \in \delta(q_2, a)$ such that $L(q'_1) = L(q'_2)$. This can be used to show that two states belong to different blocks of *CSR*. For instance, consider states 2 and 3 of the NFA on the left of Figure 3.9. They recognize the same language, but state 2 has a c-successor, namely state 4, that recognizes $\{d\}$, while state 3 has no such successor. So states 2 and 3 belong to different blocks of *CSR*. A possible run of of the CSR algorithm on this NFA is shown on the right of the figure. For this NFA, *CSR* has as many blocks as states. □

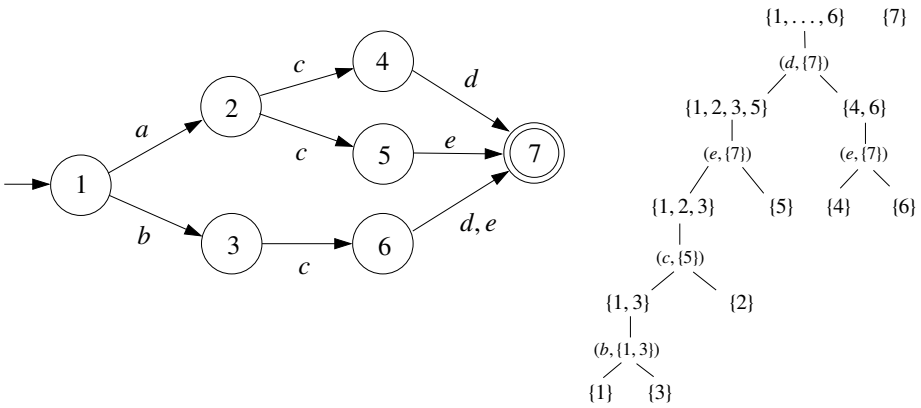


Figure 3.9: An NFA such that $CSR \neq P_\ell$.

3.4 A Characterization of the Regular Languages

We present a useful byproduct of the results of Section 3.1.

Theorem 3.24 *A language L is regular iff it has finitely many residuals.*

Proof: If L is not regular, then no DFA recognizes it. Since, by Proposition 3.6, the canonical automaton C_L recognizes L, then C_L necessarily has infinitely many states, and so L has infinitely many residuals.

If L is regular, then some DFA A recognizes it. By Lemma 3.3, the number of states of A is greater than or equal to the number of residuals of L, and so L has finitely many residuals. □

This theorem provides a useful technique for proving that a given language $L \subseteq \Sigma^*$ is not regular: exhibit an infinite set of words $W \subseteq \Sigma^*$ with pairwise different residuals, i.e., W must satisfy $L^w \neq L^v$ for every two distinct words $w, v \in W$. Let us apply the technique to some typical examples.

- $\{a^n b^n \mid n \geq 0\}$ is not regular. Let $W = \{a^k \mid k \geq 0\}$. For every two distinct words $a^i, a^j \in W$ (i.e., $i \neq j$), we have $b^i \in L^{a^i}$ but $b^i \notin L^{a_j}$.

- $\{ww \mid w \in \Sigma^*\}$ is not regular. Let $W = \Sigma^*$. For every two distinct words $w, v \in W$ (i.e., $w \neq v$), we have $w \in L^w$ but $v \notin L^w$.
- $\{a^{n^2} \mid n \geq 0\}$. Let $W = \{a^{n^2} \mid n \geq 0\}$ ($W = L$ in this case). For every two distinct words $a^{i^2}, a^{j^2} \in W$ (i.e., $i \neq j$), we have that a^{2i+i} belongs to the a^{i^2}-residual of L, because $a^{i^2+2i+1} = a^{(i+1)^2}$, but not to the a^{j^2}-residual, because a^{j^2+2i+1} is only a square number for $i = j$.

Exercises

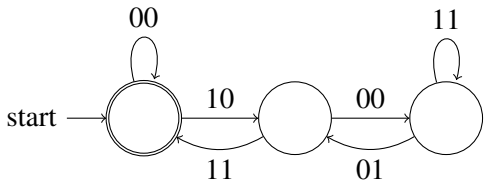
Exercise 17 Consider the most-significant-bit-first encoding (msbf encoding) of natural numbers over the alphabet $\Sigma = \{0, 1\}$. Recall that every number has infinitely many encodings, because all the words of $0 * w$ encode the same number as w. Construct the minimal DFAs accepting the following languages.

- $\{w \mid \text{msbf}^{-1}(w) \mod 3 = 0\} \cap \Sigma^4$.
- $\{w \mid \text{msbf}^{-1}(w) \text{ is a prime }\} \cap \Sigma^4$.

Exercise 18 Consider the family of languages $L_k = \{ww \mid w \in \Sigma^k\}$, where $k \geq 2$.

- Construct the minimal DFA for L_4.
- How many states has the minimal DFA accepting L_k ?

Exercise 19 Consider the following DFA over the alphabet $\Sigma = \{00, 01, 10, 11\}$:



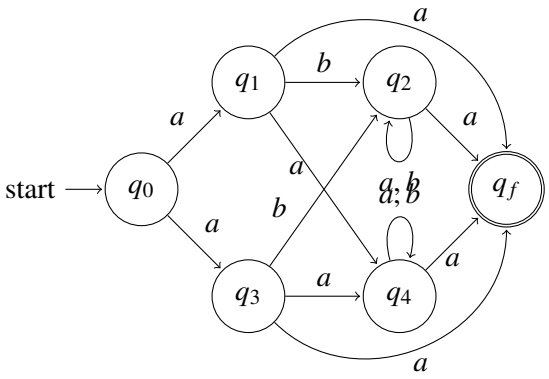
We interpret a word $w\Sigma^*$ as a pair of natural numbers $(X(w), Y(w)) \in \mathbb{N}_0 \times \mathbb{N}_0$ by reading the bits at odd positions as the msbf encoding encoding of $X(w)$, and the bits as even positions as the mbsf encoding of $Y(w)$. For example, if $w = 00001011$ then $X(w) = 3$ abd $Y(w) = 1$, because

$$00001011 \to 0\underline{0}0\underline{0}1\underline{0}1\underline{1} \to (0011, 0001) \to (3, 1)$$

1. Show that a word w is accepted by the DFA iff $X(w) = 3 \cdot Y(w)$.
2. Construct the minimal DFA representing the language $\{w \in \{0, 1\}^* \mid \text{msbf}^{-1}(w) \text{ is divisible by } 3\}$.

Exercise 20 Consider the language partition algorithm *LanPar*. Since every execution of its while loop increases the number of blocks by one, the loop can be executed at most $|Q| - 1$ times. Show that this bound is tight, i.e. give a family of DFAs for which the loops is executed $|Q| - 1$ times. *Hint*: You can take a one-letter alphabet.

Exercise 21 Describe in words the language of the following NFA, and compute CSR, i.e., the coarsest stable refinement of P_0.



1. Describe $\mathcal{L}(\mathcal{A})$.
2. Determine the CSR of $\mathcal{A}$ using the algorithm presented in the lecture.

Exercise 22 Determine the residuals of the following languages over $\Sigma = \{a, b\}$:

- $(ab + ba)^*$,
- $(aa)^*$,
- $\{w \in \{a, b\}^* \mid$ w contains the same number of occurrences of ab and $ba\}$.
- $\{a^n b^n c^n \mid n \geq 0\}$.

Exercise 23 Given a language $L \subseteq \Sigma^*$ and $w \in \Sigma^*$, we denote ${}^wL = \{u \in \Sigma^* \mid uw \in L\}$. A language $L' \subseteq \Sigma^*$ is an *inverse residual* of L if $L' = {}^wL$ for some $w \in \Sigma^*$.

- Determine the inverse residuals of the first two languages in Exercise 22.
- Show that a language is regular iff it has finitely many inverse residuals.
- Does a language always have as many residuals as inverse residuals?

Exercise 24 Given a language $L \subseteq \Sigma^*$ and $w \in \Sigma^*$, the *w-context* of L is the set of pairs $\{(u, v) \in \Sigma^* \mid uwv \in L\}$. A language is a *context* of L if it is a w-context for at least one $w \in \Sigma^*$.

- Determine the contexts of L_2 in Exercise 22.
- Can a language have more residuals than contexts? And more contexts than residuals?

Exercise 25 A DFA with *negative transitions* (DFA-n) is a DFA whose transitions are partitioned into *positive* and *negative* transitions. A run of a DFA-n is accepting if:

- it ends in a final state *and* the number of occurrences of negative transitions is even, *or*
- it ends in a non-final state *and* the number of occurrences of negative transitions is odd.

The intuition is that taking a negative transition "inverts the polarity" of the acceptance condition: after taking the transition we accept iff we would not accept were the transition positive.

- Prove that the languages recognized by DFAs with negative transitions are regular.
- Give a DFA-n for a regular language having fewer states than the minimal DFA for the language.
- Show that the minimal DFA-n for a language is not unique (even for languages whose minimal DFA-n's have fewer states than their minimal DFAs).

Exercise 26 A residual of a regular language L is *composite* if it is the union of other residuals of L. A residual of L is *prime* if it is not composite. Show that every regular language L is recognized by an NFA whose number of states is equal to the number of prime residuals of L.

Exercise 27 Show that the following languages over $\{0, 1, 2\}$ are not regular:

- $\{0^n 1^n 2^m \mid n, m \geq 0\}$
- $\{0^{2n} 1 2^{3n} \mid n \geq 0\}$
- $\{w \mid |w|_0 = |w|_2\}$, where $|w|_\sigma$ denotes the number of occurrences of the letter σ in word w.

Exercise 28 Prove or disprove:

- A subset of a regular language is regular.
- A superset of a regular language is regular.
- If L_1 and $L_1 L_2$ are regular, then L_2 is regular.
- If L_2 and $L_1 L_2$ are regular, then L_1 is regular.

Exercise 29 (T. Henzinger) Which of these languages over the alphabet $\{0, 1\}$ are regular?

- The set of words containing the same number of 0's and 1's.
- The set of words containing the same number of occurrences of the strings 01 and 10. (E.g., 01010001 contains three occurrences of 01 and two occurrences of 10.)
- Same for the pairs of strings 00 and 11, the pair 001 and 110, and the pair 001 and 100.

Exercise 30 A word $w = a_1 \dots a_n$ is a subword of $v = b_1 \dots b_m$, denoted by $w \preceq v$, if there are indices $1 \leq i_1 < i_2 \dots < i_n \leq m$ such that $a_{i_j} = b_j$ for every $j \in \{1, \dots n\}$. Higman's lemma states that every infinite set of words over a finite alphabet contains two words w_1, w_2 such that $w_1 \preceq w_2$.

A language $L \subseteq \Sigma^*$ is *upward-closed* if for every two words $w, v \in \Sigma^*$, if $w \in L$ and $w \preceq v$, then $v \in l$. The *upward-closure* of a language L is the upward-closed language obtained by adding to L all words v such that $w \preceq v$ for some $v \in L$.

- Prove using Higman's lemma that every upward-closed language is regular.
- Give regular expressions for the upward-closures of the languages in Execise 27.
- Give an algorithm that transforms a regular expression for a language into a regular expression for its upward-closure.

Exercise 31 Consider the alphabet $\Sigma = \{up, down, left, right\}$. A word over Σ corresponds to a line in a grid consisting of concatenated segments drawn in the direction specified by the letters. In the same way, a language corresponds to a set of lines.

The set of all *staircases* can be specified as the set of lines given by the regular language $(upright)^*$. It is a regular language.

- Specify the set of all *skylines* as a regular language.
- Show that the set of all *rectangles* is not regular.

Chapter 4

Operations on Sets: Implementations

Recall the list of operations on sets that should be supported by our data structures, where U is the universe of objects, X, Y are subsets of U, x is an element of U:

Member(x, X)	:	returns **true** if $x \in X$, **false** otherwise.
Complement(X)	:	returns $U \setminus X$.
Intersection(X, Y)	:	returns $X \cap Y$.
Union(X, Y)	:	returns $X \cup Y$.
Empty(X)	:	returns **true** if $X = \emptyset$, **false** otherwise.
Universal(X)	:	returns **true** if $X = U$, **false** otherwise.
Included(X, Y)	:	returns **true** if $X \subseteq Y$, **false** otherwise.
Equal(X, Y)	:	returns **true** if $X = Y$, **false** otherwise.

We fix an alphabet Σ, and assume that there exists a bijection between U and Σ^*, i.e., we assume that each object of the universe is encoded by a word, and each word is the encoding of some object. Under this assumption, the operations on sets and elements become operations on languages and words. For instance, the first two operations become

Member(w, L)	:	returns **true** if $w \in L$, **false** otherwise.
Complement(L)	:	returns $\overline{L}$.

The assumption that each word encodes some object may seem too strong. Indeed, the language E of encodings is usually only a subset of Σ^*. However, once we have implemented the operations above under this strong assumption, we can easily modify them so that they work under a much weaker assumption, that almost always holds: the assumption that the language E of encodings is regular. Assume, for instance, that E is a regular subset of Σ^* and L is the language of encodings of a set X. Then, we implement **Complement**(X) so that it returns, not $\overline{L}$, but **Intersection**$(\overline{L}, E)$.

For each operation we present an implementation that, given automata representations of the

operands, returns an automaton representing the result (or a boolean, when that is the return type). Sections 4.1 and 4.2 consider the cases in which the representation is a DFA and a NFA, respectively.

4.1 Implementation on DFAs

In order to evaluate the complexity of the operations we must first make explicit our assumptions on the complexity of basic operations on a DFA $A = (Q, \Sigma, \delta, q_0, F)$. We assume that dictionary operations (lookup, add, remove) on Q and δ can be performed in constant time using hashing. We assume further that, given a state q, we can decide in constant time if $q = q_0$, and if $q \in F$, and that given a state q and a letter $a \in \Sigma$, we can find in constant time the unique state $\delta(q, a)$.

4.1.1 Membership.

To check membership for a word w we just execute the run of the DFA on w. It is convenient for future use to have an algorithm $Member[A](w, q)$ that takes as parameter a DFA A, a state q of A, and a word w, and checks if w is accepted with q as initial state. **Member**(w, L) can then be implemented by $Mem[A](w, q_0)$, where A is the automaton representing L.

Writing $head(aw) = a$ and $tail(aw) = w$ for $a \in \Sigma$ and $w \in \Sigma^*$, the algorithm looks as follows:

MemDFA$[A](w, q)$
Input: DFA $A = (Q, \Sigma, \delta, q_0, F)$, state $q \in Q$, word $w \in \Sigma^*$,
Output: **true** if $w \in \mathcal{L}(q)$, **false** otherwise

1 **if** $w = \epsilon$ **then return** $q \in F$
2 **else return** $Member[A](\ \delta(q, head(w))\ ,\ tail(w)\ )$

The complexity of the algorithm is $\mathcal{O}(|w|)$.

4.1.2 Complement.

Implementing the complement operations on DFAs is easy. Recall that a DFA has exactly one run for each word, and the run is accepting iff it reaches a final state. Therefore, if we swap final and non-final states, the run on a word becomes accepting iff it was non-accepting, and so the new DFA accepts the word iff the new one did not accept it. So we get the following linear-time algorithm:

CompDFA(A)
Input: DFA $A = (Q, \Sigma, \delta, q_0, F)$,
Output: DFA $B = (Q', \Sigma, \delta', q_0', F')$ with $L(B) = \overline{L(A)}$

1 $Q' \leftarrow Q$; $\delta' \leftarrow \delta$; $q_0' \leftarrow q_0$; $F' = \emptyset$
2 **for all** $q \in Q$ **do**
3 **if** $q \notin F$ **then add** q **to** F'

Observe that complementation of DFAs preserves minimality. By construction, each state of $Comp(A)$ recognizes the complement of the language recognized by the same state in A. Therefore, if the states of A recognize pairwise different languages, so do the states of $Comp(A)$. Apply now Corollary 3.8, stating that a DFA is minimal iff their states recognize different languages.

4.1.3 Binary Boolean Operations

Instead of specific implementations for union and intersection, we give a generic implementation for all binary boolean operations. Given two DFAs A_1 and A_2 and a binary boolean operation like union, intersection, or difference, the implementation returns a DFA recognizing the result of applying the operation to $L(A_1)$ and $L(A_2)$. The DFAs for different boolean operations always have the same states and transitions, they differ only in the set of final states. We call this DFA with a yet unspecified set of final states the *pairing* of A_1 and A_2, denoted by $[A_1, A_2]$. Formally:

Definition 4.1 *Let* $A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$ *and* $A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$ *be DFAs. The* pairing $[A_1, A_2]$ *of* A_1 *and* A_2 *is the tuple* (Q, Σ, δ, q_0) *where:*

- $Q = \{\ [q_1, q_2]\ \mid q_1 \in Q_1, q_2 \in Q_2\}$*;*
- $\delta = \{\ ([q_1, q_2], a, [q_1', q_2'])\ \mid (q_1, a, q_1') \in \delta_1, (q_2, a, q_2') \in \delta_2\}$*;*
- $q_0 = [q_{01}, q_{02}]$.

The run of $[A_1, A_2]$ *on a word of* Σ^* *is defined as for DFAs.*

It follows immediately from this definition that the run of $[A_1, A_2]$ over a word $w = a_1 a_2 \dots a_n$ is also a "pairing" of the runs of A_1 and A_2 over w. Formally,

$$\begin{array}{ccccccccc} q_{01} & \xrightarrow{a_1} & q_{11} & \xrightarrow{a_2} & q_{21} & \cdots & q_{(n-1)1} & \xrightarrow{a_n} & q_{n1} \\ q_{02} & \xrightarrow{a_1} & q_{12} & \xrightarrow{a_2} & q_{22} & \cdots & q_{(n-1)2} & \xrightarrow{a_n} & q_{n2} \end{array}$$

are the runs of A_1 and A_2 on w if and only if

$$\begin{bmatrix} q_{01} \\ q_{02} \end{bmatrix} \xrightarrow{a_1} \begin{bmatrix} q_{11} \\ q_{12} \end{bmatrix} \xrightarrow{a_2} \begin{bmatrix} q_{21} \\ q_{22} \end{bmatrix} \cdots \begin{bmatrix} q_{(n-1)1} \\ q_{(n-1)2} \end{bmatrix} \xrightarrow{a_n} \begin{bmatrix} q_{n1} \\ q_{n2} \end{bmatrix}$$

is the run of $[A_1, A_2]$ on w.

DFAs for different boolean operations are obtained by adding an adequate set of final states to $[A_1, A_2]$. Let L_1, L_2 be the languages For intersection, $[A_1, A_2]$ must accept w if and only if A_1 accepts w *and* A_2 accepts w. This is achieved by declaring a state $[q_1, q_2]$ final if and only if $q_1 \in F_1$ *and* $q_2 \in F_2$. For union, we just replace *and* by *or*. For difference, $[A_1, A_2]$ must accept w if and only if A_1 accepts w *and* A_2 does *not* accepts w, and so we declare $[q_1, q_2]$ final if and only if $q_1 \in F_1$ *and not* $q_2 \in F_2$.

Example 4.2 Figure 4.2 shows at the top two DFAs over the alphabet $\Sigma = \{a\}$. They recognize the words whose length is a multiple of 2 and a multiple of three, respectively. We denote these languages by *Mult_2* and *Mult_3*. The Figure then shows the pairing of the two DFAs (for clarity the states carry labels x, y instead of $[x, y]$), and three DFAs recognizing $Mult\_2 \cap Mult\_3$, $Mult\_2 \cup Mult\_3$, and $Mult\_2 \setminus Mult\_3$, respectively. □

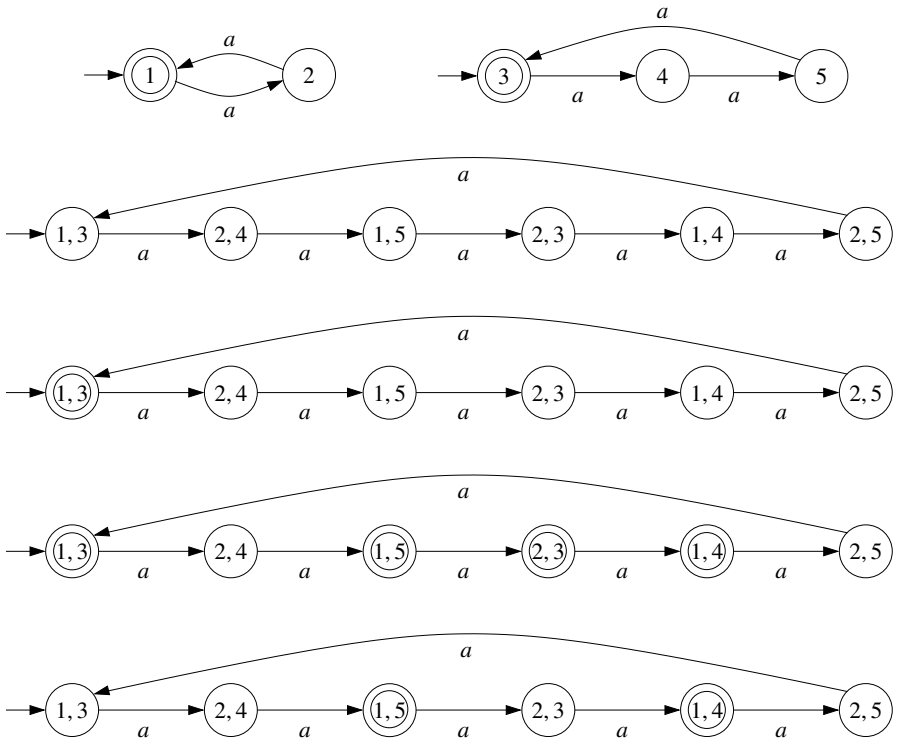


Figure 4.1: Two DFAs, their pairing, and DFAs for the intersection, union, and difference of their languages.

Example 4.3 The tour of conversions of Chapter 2 started with a DFA for the language of all words over $\{a, b\}$ containing an even number of a's and an even number of b's. This language is the intersection of the language of all words containing an even number of a's, and the language of all words containing an even number of b's. Figure 4.2 shows DFAs for these two languages, and the DFA for their intersection. □

We can now formulate a generic algorithm that, given two DFAs recognizing languages L_1, L_2 and a binary boolean operation, returns a DFA recognizing the result of "applying" the boolean operation to L_1, L_2. First we formally define what this means. Given an alphabet Σ and a binary

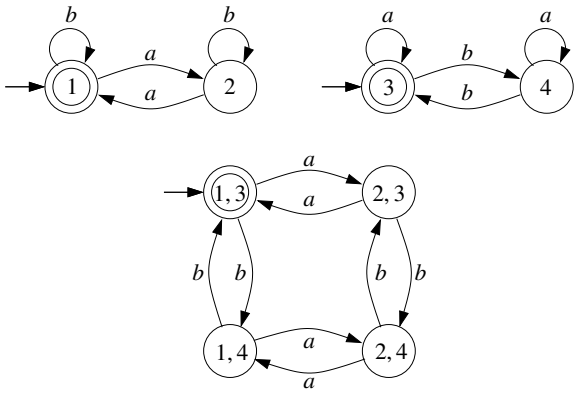


Figure 4.2: Two DFAs and a DFA for their intersection.

boolean operator $\odot$: {**true**, **false**} $\times$ {**true**, **false**} $\to$ {**true**, **false**}, we lift $\odot$ to a function $\widehat{\odot}$: $2^{\Sigma^*} \times 2^{\Sigma^*} \to 2^{\Sigma^*}$ on languages as follows

$$L_1 \widehat{\odot} L_2 \quad = \quad \{w \in \Sigma^* \mid (w \in L_1) \odot (w \in L_2)\}$$

That is, in order to decide if w belongs to $L_1 \widehat{\odot} L_2$, we first evaluate $(w \in L_1)$ and $(w \in L_2)$ to **true** of **false**, and then apply $\widehat{\odot}$ to the results. For instance we have $L_1 \cap L_2 = L_1 \overline{\wedge} L_2$. The generic algorithm, parameterized by $\odot$, looks as follows:

BinOp[$\odot$](A_1, A_2)
Input: DFAs $A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$, $A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$
Output: DFA $A = (Q, \Sigma, \delta, q_0, F)$ with $L(A) = L(A_1) \mathbin{\widehat{\odot}} L(A_2)$

```
1   Q, δ, F ← ∅
2   q0 ← [q01, q02]
3   W ← {q0}
4   while W ≠ ∅ do
5     pick [q1, q2] from W
6     add [q1, q2] to Q
7     if (q1 ∈ F1) ⊙ (q2 ∈ F2) then add [q1, q2] to F
8     for all a ∈ Σ do
9       q1' ← δ1(q1, a); q2' ← δ2(q2, a)
10      if [q1', q2'] ∉ Q then add [q1', q2'] to W
11      add ([q1, q2], a, [q1', q2']) to δ
```

Popular choices of boolean language operations are summarized in the left column below, while the right column shows the corresponding boolean operation needed to instantiate *BinOp*[$\odot$].

Language operation	$b_1 \odot b_2$
Union	$b_1 \vee b_2$
Intersection	$b_1 \wedge b_2$
Set difference ($L_1 \setminus L_2$)	$b_1 \wedge \neg b_2$
Symmetric difference ($L_1 \setminus L_2 \cup L_2 \setminus L_1$)	$b_1 \Leftrightarrow \neg b_2$

The output of *BinOp* is a DFA with $\mathcal{O}(|Q_1| \cdot |Q_2|)$, states, regardless of the boolean operation being implemented. To show that the bound is reachable, let $\Sigma = \{a\}$, and for every $n \geq 1$ let $Mult_n$ denote the language of words whose length is a multiple of n. As in Figure 4.2, the minimal DFA recognizing L_n is a cycle of n states, with the initial state being also the only final state. For any two relatively prime numbers n_1 and n_2, we have $Mult\_n_1 \cap Mult\_n_2 = Mult\_(n_1 \cdot n_2)$. Therefore, any DFA for $Mult\_(n_1 \cdot n_2)$ has at least $n_1 \cdot n_2$ states. In fact, if we denote the minimal DFA for $Mult\_k$ by A_k, then $BinOp[\wedge](A_n, A_m) = A_{n \cdot m}$.

Notice however, that in general minimality is *not* preserved: the product of two minimal DFAs may not be minimal. In particular, given any regular language L, the minimal DFA for $L \cap \overline{L}$ has one state, but the result of the product construction is a DFA with the same number of states as the minimal DFA for L.

4.1.4 Emptiness.

A DFA accepts the empty language if and only if it has no final states (recall our normal form, where all states must be reachable!).

Empty(A)
Input: DFA $A = (Q, \Sigma, \delta, q_0, F)$
Output: **true** if $L(A) = \emptyset$, **false** otherwise

```
1  return F = ∅
```

The runtime depends on the implementation. If we keep a boolean indicating whether the DFA has some final state, then the complexity of *Empty*() is $\mathcal{O}(1)$. If checking $F = \emptyset$ requires a linear scan over Q, then the complexity is $\mathcal{O}(|Q|)$.

4.1.5 Universality.

A DFA accepts Σ^* iff all its states are final, again an algorithm with complexity $\mathcal{O}(1)$ given normal form, and $\mathcal{O}(|Q|)$ otherwise.

UnivDFA(A)
Input: DFA $A = (Q, \Sigma, \delta, q_0, F)$
Output: **true** if $L(A) = \Sigma^*$, **false** otherwise

```
1  return F = Q
```

4.1.6 Inclusion.

Given two regular languages L_1, L_2, the following lemma characterizes when $L_1 \subseteq L_2$ holds.

Lemma 4.4 *Let $A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$ be DFAs. $L(A_1) \subseteq L(A_2)$ if and only if every state $[q_1, q_2]$ of the pairing $[A_1, A_2]$ satisfying $q_1 \in F_1$ also satisfies $q_2 \in F_2$.*

Proof: Let $L_1 = L(A_1)$ and $L_2 = L(A_1)$. We have $L_1 \not\subseteq L_2$ iff $L_1 \setminus L_2 \neq \emptyset$ iff at least one state $[q_1, q_2]$ of the DFA for $L_1 \setminus L_2$ is final iff $q_1 \in F_1$ and $q_2 \notin F_2$. □

The condition of the lemma can be checked by slightly modifying *BinOp*. The resulting algorithm checks inclusion on the fly:

InclDFA(A_1, A_2)
Input: DFAs $A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$, $A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$
Output: **true** if $L(A_1) \subseteq L(A_2)$, **false** otherwise

```
1  Q ← ∅;
2  W ← {[q01, q02]}
3  while W ≠ ∅ do
4     pick [q1, q2] from W
5     add [q1, q2] to Q
6     if (q1 ∈ F1) and (q2 ∉ F2) then return false
7     for all a ∈ Σ do
8        q1' ← δ1(q1, a); q2' ← δ2(q2, a)
9        if [q1', q2'] ∉ Q then add [q1', q2'] to W
10 return true
```

4.1.7 Equality.

For equality, just observe that $L(A_1) = L(A_2)$ holds if and only if the symmetric difference of $L(A_1)$ and $L(A_2)$ is empty. The algorithm is obtained by replacing Line 7 of *IncDFA*(A_1, A_2) by

if $((q_1 \in F_1)$ **and** $q_2 \notin F_2))$ **or** $((q_1 \notin F_1)$ **and** $(q_2 \in F_2))$ **then return false** .

4.2 Implementation on NFAs

For NFAs we make the same assumptions on the complexity of basic operations as for DFAs. For DFAs, however, we had the assumption that, given a state q and a letter $a \in \Sigma$, we can find in constant time the unique state $\delta(q, a)$. This assumption no longer makes sense for NFA, since $\delta(q, a)$ is a set.

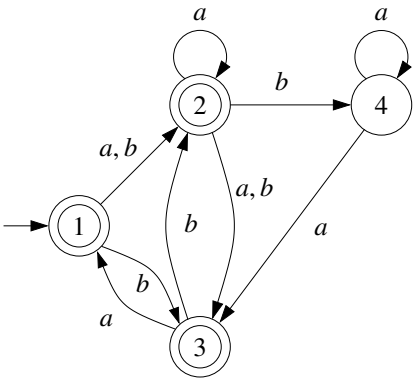
4.2.1 Membership.

Membership testing is slightly more involved for NFAs than for DFAs. An NFA may have many runs on the same word, and examining all of them one after the other in order to see if at least one is accepting is a bad idea: the number of runs may be exponential in the length of the word. The algorithm below does better. For each prefix of the word it computes the *set of states* in which the automaton may be after having read the prefix.

MemNFA$[A](w)$
Input: NFA $A = (Q, \Sigma, \delta, q_0, F)$, word $w \in \Sigma^*$,
Output: **true** if $w \in \mathcal{L}(A)$, **false** otherwise

```
1  W ← {q_0};
2  while w ≠ ε do
3     U ← ∅
4     for all q ∈ W do
5        add δ(q, head(w)) to U
6     W ← U
7     w ← tail(w)
8  return (W ∩ F ≠ ∅)
```

Example 4.5 Consider the NFA of Figure 4.3, and the word $w = aaabba$. The successive values of W, that is, the sets of states A can reach after reading the prefixes of w, are shown on the right. Since the final set contains final states, the algorithm returns **true**. □



Prefix read	W
ϵ	$\{1\}$
a	$\{2\}$
aa	$\{2, 3\}$
aaa	$\{1, 2, 3\}$
$aaab$	$\{2, 3, 4\}$
$aaabb$	$\{2, 3, 4\}$
$aaabba$	$\{1, 2, 3, 4\}$

Figure 4.3: An NFA A and the run of $Mem[A](aaabba)$.

For the complexity, observe first that the **while** loop is executed $|w|$ times. The **for** loop is executed at most $|Q|$ times. Each execution takes at most time $\mathcal{O}(|Q|)$, because $\delta(q, head(w))$ contains at most $|Q|$ states. So the overall runtime is $\mathcal{O}(|w| \cdot |Q|^2)$.

4.2.2 Complement.

Recall that an NFA A may have multiple runs on a word w, and it accepts w if *at least one* is accepting. In particular, an NFA can accept w because of an accepting run ρ_1, but have another non-accepting run ρ_2 on w. It follows that the complementation operation for DFAs cannot be extended to NFAs: after exchanging final and non-final states the run ρ_1 becomes non-accepting, but ρ_2 becomes accepting. So the new NFA still accepts w (at least ρ_2 accepts), and so it does not recognize the complement of $L(A)$.

For this reason, complementation for NFAs is carried out by converting to a DFA, and complementing the result.

CompNFA(A)
Input: NFA A,
Output: DFA $\overline{A}$ with $L(\overline{A}) = \overline{L(A)}$
1 $\overline{A} \leftarrow$ *CompDFA* (*NFAtoDFA*(A))

Since making the NFA deterministic may cause an exponential blow-up in the number of states, the number of states of $\overline{A}$ may be $\mathcal{O}(2^{|Q|})$.

4.2.3 Union and intersection.

On NFAs it is no longer possible to uniformly implement binary boolean operations. The pairing operation can be defined exactly as in Definition 4.1. The runs of a pairing $[A_1, A_2]$ of NFAs on a given word are defined as for NFAs. The difference with respect to the DFA case is that the pairing may have *multiple* runs or *no run at all* on a word. But we still have that

$$\begin{array}{ccccccccccc} q_{01} & \xrightarrow{a_1} & q_{11} & \xrightarrow{a_2} & q_{21} & \cdots & q_{(n-1)1} & \xrightarrow{a_n} & q_{n1} \\ q_{02} & \xrightarrow{a_1} & q_{12} & \xrightarrow{a_2} & q_{22} & \cdots & q_{(n-1)2} & \xrightarrow{a_n} & q_{n2} \end{array}$$

are runs of A_1 and A_2 on w if and only if

$$\begin{bmatrix} q_{01} \\ q_{02} \end{bmatrix} \xrightarrow{a_1} \begin{bmatrix} q_{11} \\ q_{12} \end{bmatrix} \xrightarrow{a_2} \begin{bmatrix} q_{21} \\ q_{22} \end{bmatrix} \cdots \begin{bmatrix} q_{(n-1)1} \\ q_{(n-1)2} \end{bmatrix} \xrightarrow{a_n} \begin{bmatrix} q_{n1} \\ q_{n2} \end{bmatrix}$$

is a run of $[A_1, A_2]$ on w.

Let us now discuss separately the cases of intersection, union, and set difference.

Intersection. Let $[q_1, q_2]$ be a final state of $[A_1, A_2]$ if q_1 is a final state of A_1 *and* q_2 is a final state of q_2. Then it is still the case that $[A_1, A_2]$ has an accepting run on w if and only if A_1 has an accepting run on w *and* A_2 has an accepting run on w. So, with this choice of final states, $[A_1, A_2]$ recognizes $L(A_1) \cap L(A_2)$. So we get the following algorithm:

IntersNFA(A_1, A_2)
Input: NFA $A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$, $A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$
Output: NFA $A_1 \cap A_2 = (Q, \Sigma, \delta, q_0, F)$ with $L(A_1 \cap A_2) = L(A_1) \cap L(A_2)$

```
1  Q, δ, F ← ∅
2  q0 ← [q01, q02]
3  W ← { [q01, q02] }
4  while W ≠ ∅ do
5     pick [q1, q2] from W
6     add [q1, q2] to Q
7     if (q1 ∈ F1) and (q2 ∈ F2) then add [q1, q2] to F
8     for all a ∈ Σ do
9        for all q1' ∈ δ1(q1, a), q2' ∈ δ2(q2, a) do
10          if [q1', q2'] ∉ Q then add [q1', q2'] to W
11          add ([q1, q2], a, [q1', q2']) to δ
```

Notice that we overload the symbol $\cap$, and denote the output by $A_1 \cap A_2$. The automaton $A_1 \cap A_2$ is often called the *product* of A_1 and A_2. It is easy to see that, as operation on NFAs, $\cap$ is associative and commutative in the following sense:

$$\begin{array}{rcccl} L((A_1 \cap A_2) \cap A_3) & = & L(A_1) \cap L(A_2) \cap L(A_3) & = & L(A_1 \cap (A_2 \cap A_3)) \\ L(A_1 \cap A_2) & = & L(A_1) \cap L(A_2) & = & L(A_2 \cap A_1) \end{array}$$

For the complexity, observe that in the worst case the algorithm must examine all pairs $[t_1, t_2]$ of transitions of $\delta_1 \times \delta_2$, but every pair is examined at most once. So the runtime is $\mathcal{O}(|\delta_1||\delta_2|)$.

Example 4.6 Consider the two NFAs of Figure 4.4 over the alphabet $\{a, b\}$. The first one recognizes the words containing at least two blocks with two consecutive a's each, the second one those containing at least one block. The result of applying *IntersNFA*() is the NFA of Figure 3.6 in page 47. Observe that the NFA has 15 states, i.e., all pairs of states are reachable.

Observe that in this example the intersection of the languages recognized by the two NFAs is equal to the language of the first NFA. So there is an NFA with 5 states that recognizes the intersection, which means that the output of *IntersNFA*() is far from optimal in this case. Even after applying the reduction algorithm for NFAs we only obtain the 10-state automaton of Figure 3.7. □

Union. The argumentation for intersection still holds if we replace *and* by *or*, and so an algorithm obtained from *IntersNFA*() by substituting **or** for **and** correctly computes a NFA for $L(A_1) \cup L(A_2)$. However, there is an algorithm that produces a smaller NFA. Observe first that a NFA-ε for $L(A_1) \cup L(A_2)$ can be constructed by putting A_1 and A_2 "side by side", and adding a new initial state q_0, together with ε-transitions from q_0 to q_{01} and q_{02}. To get an NFA, we then apply the algorithm for removing ε-transitions.

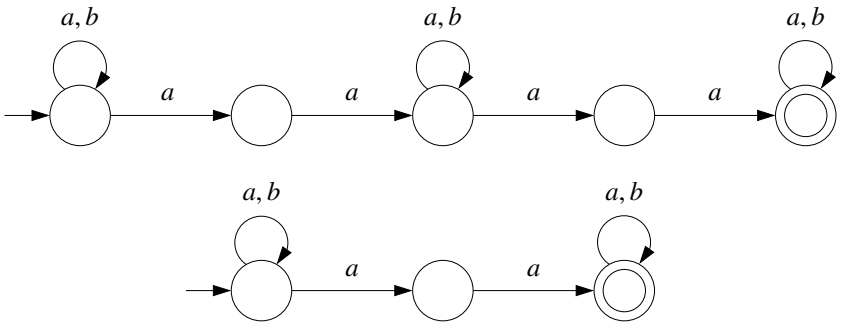


Figure 4.4: Two NFAs

The algorithm below directly construct the final result, with the assumption that q_0 is a fresh state, not belonging to Q_1 or Q_2. Removing the ϵ-transitions may make the old initial states q_{01}, q_{02} unreachable from the new initial state q_0, and in this case we remove them. Notice that the states become unreachable if and only if they had no incoming transitions in A_1 and A_2, respectively. We denote by $\delta_i^{-1}(q_{0i})$ the set of states q such that (q, a, q_{0i}) for some $a \in \Sigma$.

UnionNFA(A_1, A_2)
Input: NFA $A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$, $A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$
Output: NFA $A_1 \cup A_2$ with $L(A_1 \cup A_2) = L(A_1) \cup L(A_2)$

```
1  Q ← Q1 ∪ Q2 ∪ {q0}
2  δ ← δ1 ∪ δ2
3  F ← F1 ∪ F2
4  for all i = 1, 2 do
5     if q0i ∈ Fi then add q0 to F
6     for all (q0i, a, q) ∈ δi do
7        add (q0, a, q) to δ
8     if δi^{-1}(q0i) = ∅ then
9        remove q0i from Q
10       for all a ∈ Σ, q ∈ δi(q0i, a) do
11          remove (q0i, a, q) from δi
12 return (Q, Σ, δ, q0, F)
```

Figure 4.5 is a graphical representation of how the algorithm works (up to removal of the old initial states). If emptiness of $\delta_i^{-1}(q_{0i})$ can be checked in constant time, then the complexity is $\mathcal{O}(m_1 + m_2)$, where m_i is the number of transitions of A_i starting at q_{0i}.

Set difference. The generalization of the procedure for DFAs fails. Let $[q_1, q_2]$ be a final state of $[A_1, A_2]$ if q_1 is a final state of A_1 *and* q_2 *is not* a final state of q_2. Then $[A_1, A_2]$ has an accepting

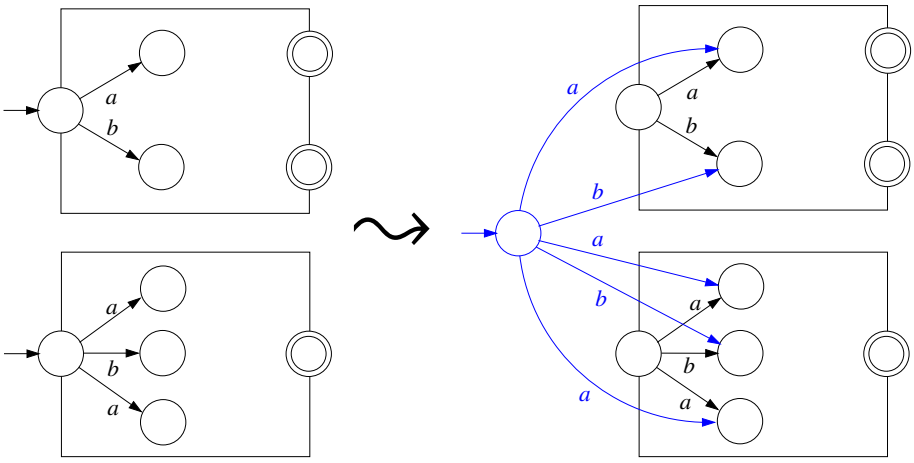


Figure 4.5: Union for NFAs

run on w if and only if A_1 has an accepting run on w *and* A_2 has a non-accepting run on w. But "A_2 has a non-accepting run on w" is no longer equivalent to "A_2 has no accepting run on w": this only holds in the DFA case. An algorithm producing an NFA $A_1 \setminus A_2$ recognizing $L(A_1) \setminus L(A_2)$ can be obtained from the algorithms for complement and intersection through the equality $L(A_1) \setminus L(A_2) = L(A_1) \cap \overline{L(A_2)}$.

4.2.4 Emptiness and Universality.

Emptiness for NFAs is decided using the same algorithm as for DFAs: just check if the NFA has at least one final state.

Universality requires a new algorithm. Since an NFA may have multiple runs on a word, an NFA may be universal even if some states are non-final: for every word having a run that leads to a non-final state there may be another run leading to a final state. An example is the NFA of Figure 4.3, which, as we shall show in this section, is universal.

A language L is universal if and only if $\overline{L}$ is empty, and so universality of an NFA A can be checked by applying the emptiness test to $\overline{A}$. Since complementation, however, involves a worst-case exponential blowup in the size of A, the algorithm requires exponential time and space.

We show that the universality problem is PSPACE-complete. That is, the superpolynomial blowup cannot be avoided unless $P = PSPACE$,which is unlikely.

Theorem 4.7 *The universality problem for NFAs is PSPACE-complete*

Proof: We only sketch the proof. To prove that the problem is in PSPACE, we show that it belongs to NPSPACE and apply Savitch's theorem. The polynomial-space nondeterministic algorithm for universality looks as follows. Given an NFA $A = (Q, \Sigma, \delta, q_0, F)$, the algorithm guesses a run of $B = NFAtoDFA(A)$ leading from $\{q_0\}$ to a non-final state, i.e., to a set of states of A containing no

final state (if such a run exists). The algorithm only does not store the whole run, only the current state, and so it only needs linear space in the size of A.

We prove PSPACE-hardness by reduction from the acceptance problem for linearly bounded automata. A linearly bounded automaton is a deterministic Turing machine that always halts and only uses the part of the tape containing the input. A configuration of the Turing machine on an input of length k is coded as a word of length k. The run of the machine on an input can be encoded as a word $c_0\#c_1 \ldots \#c_n$, where the c_i's are the encodings of the configurations.

Let Σ be the alphabet used to encode the run of the machine. Given an input x, M accepts if there exists a word w of Σ^* satisfying the following properties:

(a) w has the form $c_0\#c_1 \ldots \#c_n$, where the c_i's are configurations;

(b) c_0 is the initial configuration;

(c) c_n is an accepting configuration; and

(d) for every $0 \leq i \leq n-1$: c_{i+1} is the successor configuration of c_i according to the transition relation of M.

The reduction shows how to construct in polynomial time, given a linearly bounded automaton M and an input x, an NFA $A(M, x)$ accepting all the words of Σ^* that do *not* satisfy at least one of the conditions (a)-(d) above. We then have

- If M accepts x, then there is a word $w(M, x)$ encoding the accepting run of M on x, and so $L(A(M, x)) = \Sigma^* \setminus \{w(M, x)\}$.
- If M rejects x, then no word encodes an accepting run of M on x, and so $L(A(M, x)) = \Sigma^*$.

So M accepts x if and only if $L(A(M, x)) = \Sigma^*$, and we are done. □

A Subsumption Test. We show that it is not necessary to completely construct $\overline{A}$. First, the universality check for DFA only examines the states of the DFA, not the transitions. So instead of *NFAtoDFA*(A) we can apply a modified version that only stores the states of $\overline{A}$, but not its transitions. Second, it is not necessary to store all states.

Definition 4.8 *Let A be a NFA, and let B = NFAtoDFA(A). A state Q' of B is* minimal *if no other state Q'' satisfies $Q'' \subset Q'$.*

Proposition 4.9 *Let A be a NFA, and let B = NFAtoDFA(A). A is universal iff every minimal state of B is final.*

Proof: Since A and B recognize the same language, A is universal iff B is universal. So A is universal iff every state of B is final. But a state of B is final iff it contains some final state of A, and so every state of B is final iff every minimal state of B is final. □

Example 4.10 Figure 4.6 shows a NFA on the left, and the equivalent DFA obtained through the application of *NFAtoDFA*() on the right. Since all states of the DFA are final, the NFA is universal. Only the states {1}, {2}, and {3, 4} (shaded in the picture), are minimal. □

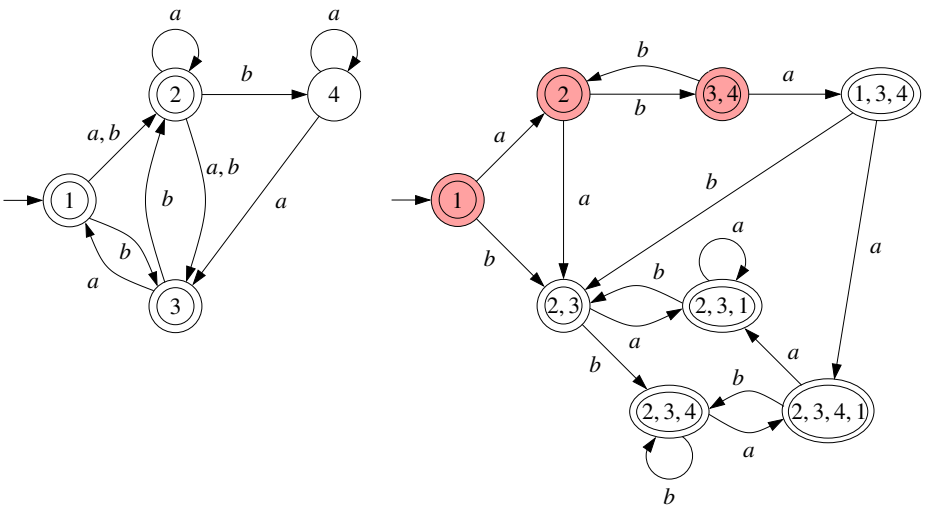


Figure 4.6: An NFA, and the result of converting it into a DFA, with the minimal states shaded.

Proposition 4.9 shows that it suffices to construct and store the minimal states of B. Algorithm *UnivNFA*(A) below constructs the states of B as in *NFAtoDFA*(A), but introduces at line 8 a *subsumption test*: it checks if some state $Q'' \subseteq \delta(Q', a)$ has already been constructed. In this case either $\delta(Q', a)$ has already been constructed (case $Q'' = \delta(Q', a)$) or is non-minimal (case $Q'' \subset \delta(Q', a)$). In both cases, the state is not added to the worklist.

UnivNFA(A)
Input: NFA $A = (Q, \Sigma, \delta, q_0, F)$
Output: **true** if $L(A) = \Sigma^*$, **false** otherwise

```
1  Ω ← ∅;
2  W ← { {q0} }
3  while W ≠ ∅ do
4     pick Q' from W
5     if  Q' ∩ F = ∅  then return false
6     add Q' to Ω
7     for all a ∈ Σ do
8        if W ∪ Ω contains no Q'' ⊆ δ(Q', a) then add δ(Q', a) to W
9  return true
```

The next proposition shows that *UnivNFA*(A) constructs *all* minimal states of B. If *UnivNFA*(A) would first generate all states of $\overline{A}$ and then would remove all non-minimal states, the proof would be trivial. But the algorithm removes non-minimal states whenever they appear, and we must show that this does not prevent the future generation of other minimal states.

Proposition 4.11 *Let $A = (Q, \Sigma, \delta, q_0, F)$ be a NFA, and let $B =$ NFAtoDFA(A). After termination of UnivNFA(A), the set $\mathcal{Q}$ contains all minimal states of B.*

Proof: Let $\mathcal{Q}_t$ be the value of $\mathcal{Q}$ after termination of *UnivNFA*(A). We show that no path of B leads from a state of $\mathcal{Q}_t$ to a minimal state of B not in $\mathcal{Q}_t$. Since $\{q_0\} \in \mathcal{Q}_t$ and all states of B are reachable from $\{q_0\}$, it follows that every minimal state of B belongs to $\mathcal{Q}_t$.

Assume there is a path $\pi = Q_1 \xrightarrow{a_1} Q_2 \dots Q_{n-1} \xrightarrow{a_n} Q_n$ of B such that $Q_1 \in \mathcal{Q}_t$, $Q_n \notin \mathcal{Q}_t$, and Q_n is minimal. Assume further that π is as short as possible. This implies $Q_2 \notin \mathcal{Q}_t$ (otherwise $Q_2 \dots Q_{n-1} \xrightarrow{a_n} Q_n$ is a shorter path satisfying the same properties), and so Q_2 is never added to the worklist. On the other hand, since $Q_1 \in \mathcal{Q}_t$, the state Q_1 is eventually added to and picked from the worklist. When Q_1 is processed at line 7 the algorithm considers $Q_2 = \delta(Q_1, a_1)$, but does not add it to the worklist in line 8. So at that moment either the worklist or $\mathcal{Q}$ contain a state $Q'_2 \subset Q_2$. This state is eventually added to $\mathcal{Q}$ (if it is not already there), and so $Q'_2 \in \mathcal{Q}_t$. So B has a path $\pi' = Q'_2 \xrightarrow{a_2} Q'_3 \dots Q'_{n-1} \xrightarrow{a_n} Q'_n$ for some states $Q'_3, \dots, Q'_n$. Since $Q'_2 \subset Q_2$ we have $Q'_2 \subset Q_2, Q'_3 \subseteq Q_3, \dots, Q'_n \subseteq Q_n$ (notice that we may have $Q'_3 = Q_3$). By the minimality of Q_n, we get $Q'_n = Q_n$, and so π' leads from Q'_2, which belongs to $\mathcal{Q}_t$, to Q_n, which is minimal and not in to $\mathcal{Q}_t$. This contradicts the assumption that π is as short as possible. □

Notice that the complexity of the subsumption test may be considerable, because the new set $\delta(Q', a)$ must be compared with every set in $\mathcal{W} \cup \mathcal{Q}$. Good use of data structures (hash tables or radix trees) is advisable.

4.2.5 Inclusion and Equality.

Recall Lemma 4.4: given two DFAs A_1, A_2, $L(A_1) \subseteq L(A_2)$ holds if and only if every state $[q_1, q_2]$ of $[A_1, A_2]$ satisfying $q_1 \in F_1$ also satisfies $q_2 \notin F_2$. This lemma no longer holds for NFAs. To see why, let A be any NFA having two runs for some word w, one of them leading to a final state q_1, the other to a non-final state q_2. We have $L(A) \subseteq L(A)$, but the pairing $[A, A]$ has a run on w leading to $[q_1, q_2]$.

To obtain an algorithm for checking inclusion, we observe that $L_1 \subseteq L_2$ holds if and only if $L_1 \cap \overline{L_2} = \emptyset$. This condition can be checked using the constructions for intersection and for the emptiness check. However, as in the case of universality, we can apply a subsumption check.

Definition 4.12 *Let A_1, A_2 be NFAs, and let $B_2 =$ NFAtoDFA(A_2). A state $[q_1, Q_2]$ of $[A_1, B_2]$ is* minimal *if no other state $[q'_1, Q'_2]$ satisfies $q'_1 = q_1$ and $Q'_2 \subset Q'$.*

Proposition 4.13 *Let $A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1), A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$ be NFAs, and let $B_2 = NFAtoDFA(A_2)$. $L(A_1) \subseteq L(A_2)$ iff every minimal state $[q_1, Q_2]$ of $[A_1, B_2]$ satisfying $q_1 \in F_1$ also satisfies $Q_2 \cap F_2 \neq \emptyset$.*

Proof: Since A_2 and B_2 recognize the same language, $L(A_1) \subseteq L(A_2)$ iff $L(A_1) \cap \overline{L(A_2)} = \emptyset$ iff $L(A_1) \cap \overline{L(B_2)} = \emptyset$ iff $[A_1, B_2]$ has a state $[q_1, Q_2]$ such that $q_1 \in F_1$ and $Q_2 \cap F_2 = \emptyset$. But $[A_1, B_2]$ has some state satisfying this condition iff it has some minimal state satisfying it. □

So we get the following algorithm to check inclusion:

InclNFA(A_1, A_2)
Input: NFAs $A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$,
Output: **true** if $L(A_1) \subseteq L(A_2)$, **false** otherwise

```
1  Q ← ∅;
2  W ← { [q01, {q02}] }
3  while W ≠ ∅ do
4     pick [q1, Q2] from W
5     if (q1 ∈ F1) and (Q2 ∩ F2 = ∅) then return false
6     add [q1, Q2] to Q
7     for all a ∈ Σ, q1' ∈ δ1(q1, a) do
8        Q2' ← δ2(Q2, a)
9        if W ∪ Q contains no [q1'', Q2''] s.t. q1'' = q1' and Q2'' ⊆ Q2' then
10          add [q1', Q2'] to W
11 return true
```

Notice that in unfavorable cases the overhead of the subsumption test may not be compensated by a reduction in the number of states. Without the test, the number of pairs that can be added to the worklist is at most $|Q_1|2^{|Q_2|}$. For each of them we have to execute the **for** loop $\mathcal{O}(|Q_1|)$ times, each of them taking $\mathcal{O}(|Q_2|^2)$ time. So the algorithm runs in $|Q_1|^2 2^{\mathcal{O}(|Q_2|)}$ time and space.

As was the case for universality, the inclusion problem is PSPACE-complete, and so the exponential cannot be avoided unless $P = PSPACE$.

Proposition 4.14 *The inclusion problem for NFAs is PSPACE-complete*

Proof: We first prove membership in PSPACE. Since PSPACE=co-PSPACE=NPSPACE, it suffices to give a polynomial space nondeterministic algorithm that decides non-inclusion. Given NFAs A_1 and A_2, the algorithm guesses a word $w \in L(A_1) \setminus L(A_2)$ letter by letter, maintaining the sets Q_1', Q_2' of states that A_1 and A_2 can reached by the word guessed so far. When the guessing ends, the algorithm checks that Q_1' contains some final state of A_1, but Q_2' does not.

Hardness follows from the fact that A is universal iff $\Sigma \subseteq L(A)$, and so the universality problem, which is PSPACE-complete, is a subproblem of the inclusion problem. □

There is however an important case with polynomial complexity, namely when A_2 is a DFA. The number the number of pairs that can be added to the worklist is then at most $|Q_1||Q_2|$. The **for** loop is still executed $\mathcal{O}(|Q_1|]$ times, but each of them takes $\mathcal{O}(1)$ time. So the algorithm runs in $\mathcal{O}(|Q_1|^2|Q_2|)$ time and space.

Equality. Equality of two languages is decided by checking that each of them is included in the other. The equality problem is again PSPACE-complete. The only point worth observing is that, unlike the inclusion case, we do not get a polynomial algorithm when A_2 is a DFA.

Exercises

Exercise 32 Consider the following languages over the alphabet $\Sigma = \{a, b\}$:

- L_1 is the set of all words where between any two occurrences of b's there is at least one a.
- L_2 is the set of all words where every non-empty maximal sequence of consecutive a's has odd length.
- L_3 is the set of all words where a occurs only at even positions.
- L_4 is the set of all words where a occurs only at odd positions.
- L_5 is the set of all words of odd length.
- L_6 is the set of all words with an even number of a's.

Remark : For this exercise we assume that the first letter of a nonempty word is at position 1, e.g., $a \in L_4$, $a \notin L_3$.

Construct an NFA for the language

$$(L_1 \setminus L_2) \cup \overline{(L_3 \triangle L_4) \cap L_5 \cap L_6}$$

where $L \triangle L'$ denotes the symmetric difference of L and L', while sticking to the following rules:

- Start from NFAs for $L_1, \ldots, L_6$.
- Any further automaton must be constructed from already existing automata via an algorithm introduced in the chapter, e.g. *Comp*, *BinOp*, *UnionNFA*, *NFAtoDFA*, etc.

Try to find an order on the construction steps such that the intermediate automata and the final result have as few states as possible.

Exercise 33 Prove or disprove: the minimal DFAs recognizing a language and its complement have the same number of states.

Exercise 34 Let r be the regular expression $((0 + 1)(0 + 1))^*$ over $\Sigma = \{0, 1\}$

- Give a regular expression r' such that $L(r') = \overline{L(r)}$.
- *Construct* a regular expression r' such that $L(r') = \overline{L(r)}$ using the algorithms of this and the preceding chapters.

Exercise 35 Let $A = (Q, \Sigma, \delta, q_0, F)$ be an NFA. Show that with the universal accepting condition of Exercise 9 the automaton $A' = (Q, \Sigma, \delta, q_0, Q \setminus F)$ recognizes the complement of the language recognized by A.

Exercise 36 Recall the alternating automata introduced in Exercise **??**.

- Let $A = (Q_1, Q_2, \Sigma, \delta, q_0, F)$ be an alternating automaton, where Q_1 and Q_2 are the sets of existential and universal states, respectively, and $\delta\colon (Q_1 \cup Q_2) \times \Sigma \to \mathcal{P}(Q_1 \cup Q_2)$. Show that the alternating automaton $A = (Q_2, Q_1, \Sigma, \delta, q_0, Q \setminus F)$ recognizes the complement of the language recognized by A. I.e., show that alternating automata can be complemented by exchanging existential and universal states, and final and non-final states.
- Give linear time algorithms that take two alternating automata recognizing languages L_1, L_2 and deliver a third alternating automaton recognizing $L_1 \cup L_2$ and $L_1 \cap L_2$.
  Hint: The algorithms are very similar to *UnionDFA*.
- Show that the emptiness problem for alternating automata is PSPACE-complete.
  Hint: Using Exercise 39, prove that the emptiness problem for alternating automata is PSPACE-complete.

Exercise 37 Find a family $\{A_n\}_{n=1}^{\infty}$ of NFAs with $O(n)$ states such that *every* NFA recognizing the complement of $L(A_n)$ has at least 2^n states.

Hint: See Exercise 10.

Exercise 38 Consider again the regular expressions $(1 + 10)^*$ and $1^*(101^*)^*$ of Exercise 2.

- Construct NFAs for the expressions and use *InclNFA* to check if their languages are equal.
- Construct DFAs for the expressions and use *InclDFA* to check if their languages are equal.
- Construct minimal DFAs for the expressions and check whether they are isomorphic.

Exercise 39

- Prove that the following problem is PSPACE-complete:

  Given: DFAs $A_1, \ldots, A_n$ over the same alphabet Σ.
  Decide: Is $\bigcap_{i=1}^{n} L(A_i) = \emptyset$?

  Hint: Reduction from the acceptance problem for deterministic, linearly bounded automata.

- Prove that if Σ is a 1-letter alphabet then the problem is "only" NP-complete.
  Hint: reduction from 3-SAT.

- Prove that if the DFAs are acyclic (but the alphabet arbitrary) then the problem is again NP-complete.

Exercise 40 Consider the variant of *IntersNFA* in which line 7

$$\textbf{if } (q_1 \in F_1) \textbf{ and } (q_2 \in F_2) \textbf{ then add } [q_1, q_2] \textbf{ to } F$$

is replaced by

$$\textbf{if } (q_1 \in F_1) \textbf{ or } (q_2 \in F_2) \textbf{ then add } [q_1, q_2] \textbf{ to } F$$

Let $A_1 \otimes A_2$ be the result of applying this variant to two NFAs A_1, A_2. We call $A_1 \otimes A_2$ the *or-product* of A_1 and A_2.

An NFA $A = (Q, \Sigma, \delta, q_0, F)$ is *complete* if $\delta(q, a) \neq \emptyset$ for every $q \in Q$ and every $a \in \Sigma$.

- Prove: If A_1 and A_2 are complete NFAs, then $L(A_1 \otimes A_2) = L(A_1) \cup L(L_2)$.

- Give NFAs A_1, A_2 such that $L(A_1 \otimes A_2) = L(A_1) \cup L(L_2)$ but neither A_1 nor A_2 are complete.

Exercise 41 Given a word $w = a_1 a_2 \ldots a_n$ over an alphabet Σ, we define the *even part* of w as the word $a_2 a_4 \ldots a_{\lfloor n/2 \rfloor}$. Given an NFA for a language L, construct an NFA recognizing the even parts of the words of L.

Exercise 42 Given regular languages L_1, L_2 over an alphabet Σ, the *left quotient* of L_1 by L_2 is the language

$$L_2 \backslash L_1 := \{v \in \Sigma^* \mid \exists u \in L_2 : \ uv \in L_1\}$$

(Note that $L_2 \backslash L_1$ is different from the set difference $L_2 \setminus L_1$.)

1. Given NFA A_1, A_2, construct an NFA A such that $L(A) = L(A_1) \backslash L(A_2)$.
2. Do the same for the *right quotient* of L_1 by L_2, defined as $L_1 / L_2 := \{u \in \Sigma^* \mid \exists v \in L_2 : \ uv \in L_1\}$.
3. Determine the inclusion relations between the following languages: L_1, $(L_1 / L_2) L_2$, and $(L_1 L_2) / L_2$

Exercise 43 We have shown in Exercise 30 that every upward-closed language is regular. A language $L \subseteq \Sigma^*$ is *downward-closed* if for every two words $w, v \in \Sigma^*$, if $w \in L$ and $v \preceq w$, then $v \in l$.

- Prove that every downward-closed language is regular.
- Give an algorithm that transforms a regular expression for a language into a regular expression for its downward-closure.

Exercise 44 (Abdulla, Bouajjani, and Jonsson) An *atomic expression* over an alphabet Σ^* is an expression of the form $\emptyset$, ϵ, $(a + \epsilon)$ or $(a_1 + \ldots + a_n)^*$, where $a, a_1, \ldots, a_n \in \Sigma$. A *product* is a concatenation $e_1\, e_2 \ldots\, e_n$ of atomic expressions. *A simple regular expression* is a sum $p_1 + \ldots + p_n$ of products.

- Prove that the language of a simple regular expression is downward-closed.
- Prove that every downward-closed language can be represented by a simple regular expression.
  Hint: since every downward-closed language is regular, it is represented by a regular expression. Prove that this expression is equivalent to a simple regular expression.

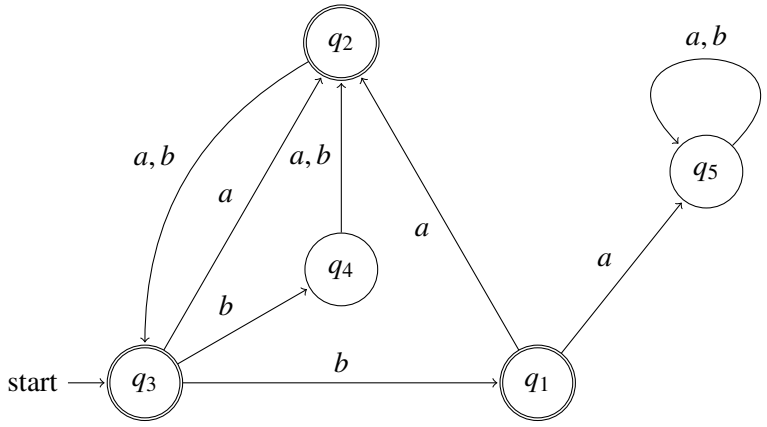
Exercise 45 Let $L_i = \{w \in \{a\}^* \mid$ the length of w is divisible by $i\}$.

1. Construct an NFA for $L := L_4 \cup L_6$ with at most 11 states.
2. Construct the minimal DFA for L.

Exercise 46

- Modify algorithm *Empty* so that when the DFA or NFA is nonempty it returns a witness, i.e., a word accepted by the automaton.
- Same for a *shortest* witness.

Exercise 47 Check by means of *UnivNFA* whether the following NFA is universal.



Exercise 48 Let Σ be an alphabet, and define the *shuffle operator* $\| \ : \ \Sigma^* \times \Sigma^* \to 2^{\Sigma^*}$ as follows, where $a, b \in \Sigma$ and $w, v \in \Sigma^*$:

$$\begin{aligned} w \,\|\, \varepsilon &= \{w\} \\ \varepsilon \,\|\, w &= \{w\} \\ aw \,\|\, bv &= \{a\}\,(w \,\|\, bv) \cup \{b\}\,(aw \,\|\, v) \cup \\ &\{bw \mid w \in au \| v\} \end{aligned}$$

For example we have:

$$b \| d = \{bd, db\}, \quad ab \| d = \{abd, adb, dab\}, \quad ab \| cd = \{cabd, acbd, abcd, cadb, acdb, cdab\}.$$

Given DFAs recognizing languages $L_1, L_2 \subseteq \Sigma^*$ construct an NFA recognizing their *shuffle*

$$L_1 \,\|\, L_2 := \bigcup_{u \in L_1, v \in L_2} u \,\|\, v \ .$$

Exercise 49 Let Σ_1, Σ_2 be two alphabets. A *homomorphism* is a map $h \ : \ \Sigma_1^* \to \Sigma_2^*$ such that $h(\varepsilon) = \varepsilon$ and $h(w_1 w_2) = h(w_1)h(w_2) for every w_1, w_2 \in \Sigma_1^*$. Observe that if $\Sigma_1 = \{a_1, \ldots, a_n\}$ then h is completely determined by the values $h(a_1), \ldots, h(a_n)$.

1. Let $h \ : \ \Sigma_1^* \to \Sigma_2^*$ be a homomorphism and let A be a NFA over Σ_1. Describe how to constuct a NFA for the language
$$h(L(A)) := \{h(w) \mid w \in L(A)\}.$$

2. Let $h \ : \ \Sigma_1^* \to \Sigma_2^*$ be a homomorphism and let A be a NFA over Σ_2. Describe how to construct a NFA for the language
$$h^{-1}(L(A)) := \{w \in \Sigma_1^* \mid h(w) \in L(A)\}.$$

3. Recall that the language $\{0^n1^n \mid n \in \mathbb{N}\}$ is not regular. Use the preceding results to show that $\{(01^k2)^n3^n \mid k, n \in \mathbb{N}\}$ is also not regular.

Exercise 50 Given alphabets Σ and Δ, a *substitution* is a map $f : \Sigma \to 2^{\Delta^*}$ assigning to each letter $a \in \Sigma$ a language $L_a \subseteq \Delta^*$.A subsitution f can be canonically extended to a map $2^{\Sigma^*} \to 2^{\Delta^*}$ by defining $f(\varepsilon) = \varepsilon$, $f(wa) = f(w)f(a)$, and $f(L) = \bigcup_{w \in L} f(w)$. Note that a homomorphism can be seen as the special case of a substitution in which all L_a's are singletons.

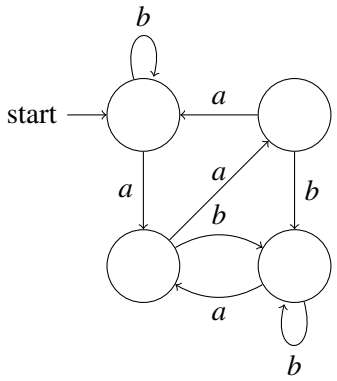
Let $\Sigma = \{\texttt{Name}, \texttt{Tel}, \texttt{:}, \#\}$, let $\Delta = \{A, \ldots, Z, 0, 1, \ldots, 9, :, \#\}$, and let f be the substitution f given by

$$\begin{aligned} f(\texttt{Name}) &= (A + \cdots + Z)^* \\ f(\texttt{:}) &= \{:\} \\ f(\texttt{Tel}) &= 0049(1 + \ldots + 9)(0 + 1 + \ldots + 9)^{10} + 00420(1 + \ldots + 9)(0 + 1 + \ldots + 9)^8 \\ f(\#) &= \{\#\} \end{aligned}$$

1. Draw a DFA recognizing $L = \texttt{Name:Tel(\#Telephone)}^*$.
2. Sketch an NFA-reg recognizing $f(L)$.
3. Give an algorithm that takes as input an NFA A, a substitution f, and for every $a \in \Sigma$ an NFA recognizing $f(a)$, and returns an NFA recognizing $f(L(A))$.

Exercise 51 A DFA is *synchronizing* if there is a word w and a state q such that after reading w from *any* state the DFA is always in state q.

1. Show that the following DFA is synchronizing.



2. Give an algorithm to decide if a given DFA is synchronizing.
3. Give a *polynomial time* algorithm to decide if a given DFA is synchronizing.

Chapter 5

Operations on Relations: Implementations

We show how to implement operations on *relations* over a (possibly infinite) universe U. Even though we will encode the elements of U as words, when implementing relations it is convenient to think of U as an abstract universe, and not as the set Σ^* of words over some alphabet Σ. The reason, as we shall see, is that for some operations we encode an element of X not by one word, but by many, actually by infinitely many. In the case of operations on sets this is not necessary, and one can safely identify the object and its encoding as word.

We are interested in a number of operations. A first group contains the operations we already studied for sets, but lifted to relations. For instance,we consider the operation **Membership**$((x, y), R)$ that returns **true** if $(x, y) \in R$, and **false** otherwise, or **Complement**(R), that returns $\overline{R} = (X \times X) \setminus R$. Their implementations will be very similar to those of the language case. A second group contains three fundamental operations proper to relations. Given $R, R_1, R_2 \subseteq U \times U$:

Projection_1(R)	:	returns the set $\pi_1(R) = \{x \mid \exists y \ (x, y) \in R\}$.
Projection_2(R)	:	returns the set $\pi_2(R) = \{y \mid \exists x \ (x, y) \in R\}$.
Join(R_1, R_2)	:	returns $R_1 \circ R_2 = \{(x, z) \mid \exists y \ \ (x, y) \in R_1 \wedge (y, z) \in R_2\}$

Finally, given $X \subseteq U$ we are interested in two derived operations:

Post(X, R)	:	returns $post_R(X) = \{y \in U \mid \exists x \in X : (x, y) \in R\}$.
Pre(X, R)	:	returns $pre_R(X) = \{y \mid \exists x \in X : (y, x) \in R\}$.

Observe that **Post**(X, R) = **Projection_2**(**Join**(Id_X, R)), and **Pre**(X, R) = **Projection_1**(**Join**(R, Id_x)), where $Id_X = \{(x, x) \mid x \in X\}$.

5.1 Encodings

We encode elements of U as words over an alphabet Σ. It is convenient to assume that Σ contains a padding letter #, and that an element x is encoded not only by a word $s_x \in \Sigma^*$, but by all the words $s_x\#^n$ with $n \geq 0$. That is, an element x has a shortest encoding s_x, and other encodings are obtained by appending to the shortest encoding an arbitrary number of padding letters. We assume that the shortest encodings of two distinct elements are also distinct, and that for every $x \in U$ the last letter of s_x is different from #. It follows that the sets of encodings of two distinct elements are disjoint.

The advantage is that for any two elements x, y there is a number n (in fact infinitely many) such that both x and y have encodings of length n. We say that (w_x, w_y) encodes the pair (x, y) if w_x encodes x, w_y encodes y, and w_x, w_y *have the same length*. Notice that if (w_x, w_y) encodes (x, y), then so does $(w_x\#^k, w_y\#^k)$ for every $k \geq 0$. If s_x, s_y are the shortest encodings of x and y, and $|s_x| \leq |s_y|$, then the shortest encoding of (x, y) is $(s_x\#^{|s_y|-|s_y|}, s_y)$.

Example 5.1 We encode the number 6 not only by its small end binary representation 011, but by any word of $L(0110^*)$. In this case we have $\Sigma = \{0, 1\}$ with 0 as padding letter. Notice, however, that taking 0 as padding letter requires to take the empty word as the shortest encoding of the number 0 (otherwise the last letter of the encoding of 0 is the padding letter).

In the rest of this chapter, we will use this particular encoding of natural numbers without further notice. We call it the *least significant bit first* encoding and write *lsbf*(6) to denote the language $L(0110^*)$ □

If we encode an element of U by more than one word, then we have to define when is an element accepted or rejected by an automaton. Does it suffice that the automaton accepts(rejects) *some* encoding, or does it have to accept (reject) *all* of them? Several definitions are possible, leading to different implementations of the operations. We choose the following option:

Definition 5.2 *Assume an encoding of the universe U over Σ^* has been fixed. Let A be an NFA.*

- *A accepts $x \in U$ if it accepts all encodings of x.*
- *A rejects $x \in U$ if it accepts no encoding of x.*
- *A recognizes a set $X \subseteq U$ if*

$$L(A) = \{w \in \Sigma^* \mid \textit{ } w \textit{ encodes some element of } X\} .$$

A set is regular *(with respect to the fixed encoding) if it is recognized by some NFA.*

Notice that if A recognizes $X \subseteq U$ then, as one would expect, A accepts every $x \in X$ and rejects every $x \notin X$. Observe further that with this definition a NFA may neither accept nor reject a given x. An NFA is *well-formed* if it recognizes some set of objects, and *ill-formed* otherwise.

5.2 Transducers and Regular Relations

We assume that an encoding of the universe U over the alphabet Σ has been fixed.

Definition 5.3 *A* transducer *over Σ is an NFA over the alphabet $\Sigma \times \Sigma$.*

Transducers are also called *Mealy machines*. According to this definition a transducer accepts sequences of pairs of letters, but it is convenient to look at it as a machine accepting pairs of words:

Definition 5.4 *Let T be a transducer over Σ. Given words $w_1 = a_1 a_2 \dots a_n$ and $w_2 = b_1 b_2 \dots b_n$, we say that T* accepts the pair (w_1, w_2) *if it accepts the word $(a_1, b_1)...(a_n, b_n) \in (\Sigma \times \Sigma)^*$.*

In other words, we identify the sets

$$\bigcup_{i \geq 0} (\Sigma^i \times \Sigma^i) \quad \text{and} \quad (\Sigma \times \Sigma)^* = \bigcup_{i \geq 0} (\Sigma \times \Sigma)^i \ .$$

We now define when a transducer accepts a pair $(x, y) \in X \times X$, which allows us to define the relation recognized by a transducer. The definition is completely analogous to Definition 5.2

Definition 5.5 *Let T be a transducer.*

- *T accepts a pair $(x, y) \in X \times X$ if it accepts all encodings of (x, y).*
- *T rejects a pair $(x, y) \in X \times X$ if it accepts no encoding of (x, y).*
- *T recognizes a relation $R \subseteq X \times X$ if*

$$L(T) = \{(w_x, w_y) \in (\Sigma \times \Sigma)^* \mid (w_x, w_y) \text{ encodes some pair of } R\} \ .$$

A relation is regular if it is recognized by some transducer.

It is important to emphasize that not every transducer recognizes a relation, because it may recognize only some, but not all, the encodings of a pair (x, y). As for NFAs, we say a transducer if *well-formed* if it recognizes some relation, and *ill-formed* otherwise.

Example 5.6 The *Collatz function* is the function $f : \mathbb{N} \to \mathbb{N}$ defined as follows:

$$f(n) = \begin{cases} 3n + 1 & \text{if } n \text{ is odd} \\ n/2 & \text{if } n \text{ is even} \end{cases}$$

Figure 5.1 shows a transducer that recognizes the relation $\{(n, f(n)) \mid n \in \mathbb{N}\}$ with *lsbf* encoding and with $\Sigma = \{0, 1\}$. The elements of $\Sigma \times \Sigma$ are drawn as column vectors with two components. The transducer accepts for instance the pair $(7, 22)$ because it accepts the pairs $(111000^k, 011010^k)$, that is, it accepts

$$\begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}\left(\begin{bmatrix}0\\0\end{bmatrix}\right)^k$$

for every $k \geq 0$, and we have $lsbf(7) = L(1110^*)$ and $lsbf(22) = L(011010^*)$. □

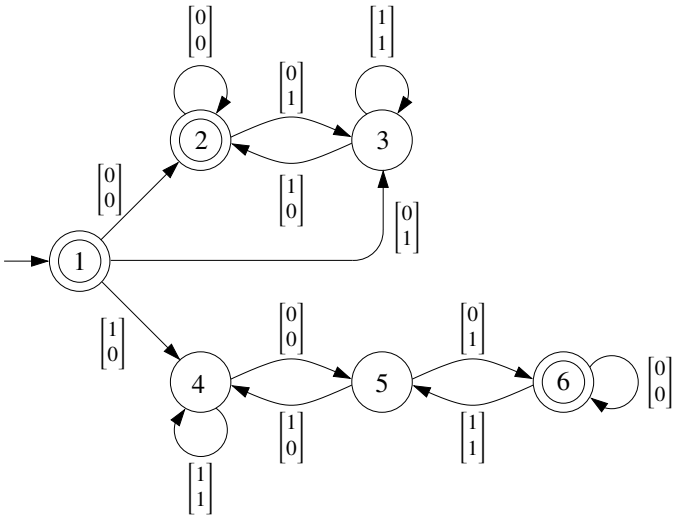


Figure 5.1: A transducer for Collatz's function.

Determinism A transducer is *deterministic* if it is a DFA. In particular, a state of a deterministic transducer over the alphabet $\Sigma\times\Sigma$ has exactly $|\Sigma|^2$ outgoing transitions. The transducer of Figure 5.1 is deterministic in this sense, when an appropriate sink state is added.

There is another possibility to define determinism of transducers, in which the letter (a,b) is interpreted as "the transducer receives the input a and produces the output b". In this view, a transducer is called deterministic if for every state q and every letter a there is exactly one transition of the form $(q,(a,b),q')$. Observe that these two definitions of determinism are *not* equivalent.

We do not give separate implementations of the operations for deterministic and nondeterministic transducers. The new operations (projection and join) can only be reasonably implemented on nondeterministic transducers, and so the deterministic case does not add anything new to the discussion of Chapter 4.

5.3 Implementing Operations on Relations

In Chapter 4 we made two assumptions on the encoding of objects from the universe U as words:

- every word is the encoding of some object, and
- every object is encoded by exactly one word.

We have relaxed the second assumption, and allowed for multiple encodings (in fact, infinitely many), of an object. Fortunately, as long as the first assumption still holds, the implementations of the boolean operations remain correct, in the following sense: If the input automata are well formed then the output automaton is also well-formed. Consider for instance the complementation operation on DFAs. Since every word encodes some object, the set of all words can be partitioned in

equivalence classes, each of them containing all the encodings of an object. If the input automaton A is well-formed, then for every object x from the universe, A either accepts all the words in an equivalence class, or all of them. The complement automaton then satisfies the same property, but accepting a class iff the original automaton does not accept it.

Notice further that membership of an object x in a set represented by a well-formed automaton can be checked by taking any encoding w_x of x, and checking if the automaton accepts w_x.

5.3.1 Projection

Given a transducer T recognizing a relation $R \subseteq X \times X$, we construct an automaton over Σ recognizing the set $\pi_1(R)$. The initial idea is very simple: loosely speaking, we go through all transitions, and replace their labels (a, b) by a. This transformation yields a NFA, and this NFA has an accepting run on a word $a_1 \dots a_n$ iff the transducer has an accepting run on some pair (w, w'). Formally, this step is carried out in lines 1-4 of the following algorithm (line 5 is explained below):

Proj_1(T)
Input: transducer $T = (Q, \Sigma \times \Sigma, \delta, q_0, F)$
Output: NFA $A = (Q', \Sigma, \delta', q_0', F')$ with $L(A) = \pi_1(L(T))$

1 $Q' \leftarrow Q$; $q_0' \leftarrow q_0$; $F'' \leftarrow F$
2 $\delta' \leftarrow \emptyset$;
3 **for all** $(q, (a, b), q') \in \delta$ **do**
4 **add** (q, a, q') **to** δ'
5 $F' \leftarrow \mathit{PadClosure}((Q', \Sigma, \delta', q_0', F''), \#)$

However, this initial idea is not fully correct. Consider the relation $R = \{(1, 4)\}$ over $\mathbb{N}$. A transducer T recognizing R recognizes the language

$$\{(10^{n+2}, 0010^n) \mid n \geq 0\}$$

and so the NFA constructed after lines 1-4 recognizes $\{10^{n+2} \mid n \geq 0\}$. However, it does not recognize the number 1, because it does not accept *all* its encodings: the encodings 1 and 10 are rejected.

This problem can be easily repaired. We introduce an auxiliary construction that "completes" a given NFA: the *padding closure* of an NFA is another NFA A' that accepts a word w if and only if the first NFA accepts $w\#^n$ for some $n \geq 0$ and a padding symbol #. Formally, the padding closure augments the set of final states and return a new such set. Here is the algorithm constructing the padding closure:

PadClosure$(A, \#)$
Input: NFA $A = (\Sigma \times \Sigma, Q, \delta, q_0, F)$
Output: new set F' of final states

```
1  W ← F; F' ← ∅;
2  while W ≠ ∅ do
3     pick q from W
4     add q to F'
5     for all (q', #, q) ∈ δ do
6        if q' ∉ F' then add q' to W
7  return F'
```

Projection onto the second component is implemented analogously. The complexity of *Proj_i*() is clearly $\mathcal{O}(|\delta| + |Q|)$, since every transition is examined at most twice, once in line 3, and possibly a second time at line 5 of *PadClosure*().

Observe that projection does *not* preserve determinism, because two transitions leaving a state and labeled by two different (pairs of) letters (a, b) and (a, c), become after projection two transitions labeled with the same letter a: In practice the projection of a transducer is hardly ever deterministic. Since, typically, a sequence of operations manipulating transitions contains at least one projection, deterministic transducers are relatively uninteresting.

Example 5.7 Figure 5.2 shows the NFA obtained by projecting the transducer for the Collatz function onto the first and second components. States 4 and 5 of the NFA at the top (first component) are made final by *PadClosure*(), because they can both reach the final state 6 through a chain of 0s (recall that 0 is the padding symbol in this case). The same happens to state 3 for the NFA at the bottom (second component), which can reach the final state 2 with 0.
Recall that the transducer recognizes the relation $R = \{(n, f(n)) \mid n \in \mathbb{N}\}$, where f denotes the Collatz function. So we have $\pi_1(R) = \{n \mid n \in \mathbb{N}\} = \mathbb{N}$ and $\pi_2(R) = \{f(n) \mid n \in \mathbb{N}\}$, and a moment of thought shows that $\pi_2(R) = \mathbb{N}$ as well. So both NFAs should be universal, and the reader can easily check that this is indeed the case. Observe that both projections are nondeterministic, although the transducer is deterministic. □

5.3.2 Join, Post, and Pre

We give an implementation of the **Join** operation, and then show how to modify it to obtain implementations of **Pre** and **Post**.

Given transducers T_1, T_2 recognizing relations R_1 and R_2, we construct a transducer $T_1 \circ T_2$ recognizing $R_1 \circ R_2$. We first construct a transducer T with the following property: T accepts (w, w') iff there is a word w'' such that T_1 accepts (w, w'') and T_2 accepts (w'', w'). The intuitive idea is to slightly modify the product construction. Recall that the pairing $[A_1, A_2]$ of two NFA A_1, A_2 has a transition $[q, r] \xrightarrow{a} [q', r']$ if and only if A_1 has a transition $q \xrightarrow{a} r$ and A_2 has a transition $q' \xrightarrow{a} r'$. Similarly, $T_1 \circ T_2$ has a transition $[q, r] \xrightarrow{(a,b)} [q', r']$ if *there is a letter* c such that T_1 has a transition

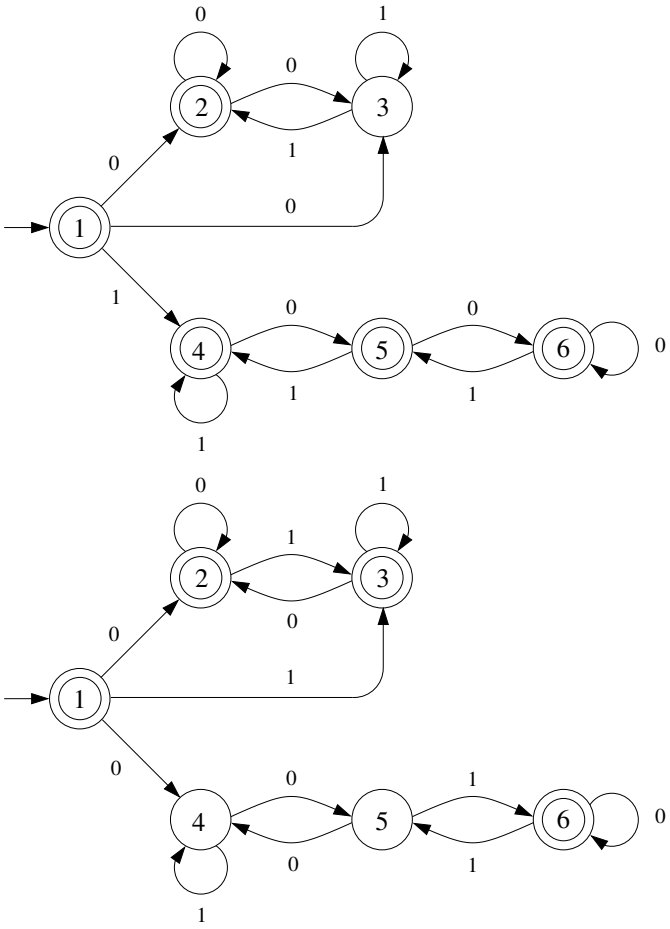


Figure 5.2: Projections of the transducer for the Collatz function onto the first and second components.

$q \xrightarrow{(a,c)} r$ and A_2 has a transition $q' \xrightarrow{(c,b)} r'$. Intuitively, T can output b on input a if there is a letter c such that T_1 can output c on input a, and T_2 can output b on input c. The transducer T has a run

$$\begin{bmatrix} q_{01} \\ q_{02} \end{bmatrix} \xrightarrow{\begin{bmatrix} a_1 \\ b_1 \end{bmatrix}} \begin{bmatrix} q_{11} \\ q_{12} \end{bmatrix} \xrightarrow{\begin{bmatrix} a_2 \\ b_2 \end{bmatrix}} \begin{bmatrix} q_{21} \\ q_{22} \end{bmatrix} \quad \cdots \quad \begin{bmatrix} q_{(n-1)1} \\ q_{(n-1)2} \end{bmatrix} \xrightarrow{\begin{bmatrix} a_n \\ b_n \end{bmatrix}} \begin{bmatrix} q_{n1} \\ q_{n2} \end{bmatrix}$$

iff T_1 and T_2 have runs

$$\begin{array}{ccccccccccc} & \begin{bmatrix} a_1 \\ c_1 \end{bmatrix} & & \begin{bmatrix} a_2 \\ c_2 \end{bmatrix} & & & & \begin{bmatrix} a_n \\ c_n \end{bmatrix} & \\ q_{01} & \longrightarrow & q_{11} & \longrightarrow & q_{21} & \cdots & q_{(n-1)1} & \longrightarrow & q_{n1} \\ & \begin{bmatrix} c_1 \\ b_1 \end{bmatrix} & & \begin{bmatrix} c_2 \\ b_2 \end{bmatrix} & & & & \begin{bmatrix} c_n \\ b_n \end{bmatrix} & \\ q_{02} & \longrightarrow & q_{12} & \longrightarrow & q_{22} & \cdots & q_{(n-1)2} & \longrightarrow & q_{n2} \end{array}$$

Formally, if $T_1 = (Q_1, \Sigma \times \Sigma, \delta_1, q_{01}, F_1)$ and $T_2 = (Q_2, \Sigma \times \Sigma, \delta_2, q_{02}, F_2)$, then $T = (Q, \Sigma \times \Sigma, \delta, q_0, F')$ is the transducer generated by lines 1–9 of the algorithm below:

Join(T_1, T_2)
Input: transducers $T_1 = (Q_1, \Sigma \times \Sigma, \delta_1, q_{01}, F_1)$, $T_2 = (Q_2, \Sigma \times \Sigma, \delta_2, q_{02}, F_2)$
Output: transducer $T_1 \circ T_2 = (Q, \Sigma \times \Sigma, \delta, q_0, F)$

1 $Q, \delta, F' \leftarrow \emptyset$; $q_0 \leftarrow [q_{01}, q_{02}]$
2 $W \leftarrow \{[q_{01}, q_{02}]\}$
3 **while** $W \neq \emptyset$ **do**
4 **pick** $[q_1, q_2]$ **from** W
5 **add** $[q_1, q_2]$ **to** Q
6 **if** $q_1 \in F_1$ and $q_2 \in F_2$ **then add** $[q_1, q_2]$ **to** F'
7 **for all** $(q_1, (a, c), q_1') \in \delta_1, (q_2, (c, b), q_2') \in \delta_2$ **do**
8 **add** $([q_1, q_2], (a, b), [q_1', q_2'])$ **to** δ
9 **if** $[q_1', q_2'] \notin Q$ **then add** $[q_1', q_2']$ **to** W
10 $F \leftarrow$ **PadClosure**$((Q, \Sigma \times \Sigma\delta, q_0, F'), (\#, \#))$

However, T is not yet the transducer we are looking for. The problem is similar to the one of the projection operation. Consider the relations on numbers $R_1 = \{(2, 4)\}$ and $R_2 = \{(4, 2)\}$. Then T_1 and T_2 recognize the languages $\{(010^{n+1}, 0010^n) \mid n \geq 0\}$ and $\{(0010^n, 010^{n+1}) \mid n \geq 0\}$ of word pairs. So T recognizes $\{(010^{n+1}, 010^{n+1}) \mid n \geq 0\}$. But then, according to our definition, T does not accept the pair $(2, 2) \in \mathbb{N} \times \mathbb{N}$, because it does not accept *all* its encodings: the encoding $(01, 01)$ is missing. So we add a padding closure again at line 10, this time using $[\#, \#]$ as padding symbol.

The number of states of *Join*(T_1, T_2) is $\mathcal{O}(|Q_1| \cdot |Q_2|)$, as for the standard product construction.

Example 5.8 Recall that the transducer of Figure 5.1, shown again at the top of Figure 5.3, recognizes the relation $\{(n, f(n)) \mid n \in \mathbb{N}\}$, where f is the Collatz function. Let T be this transducer. The bottom part of Figure 5.3 shows the transducer $T \circ T$ as computed by *Join*(T,T). For example, the transition leading from $[2, 3]$ to $[3, 2]$, labeled by $(0, 0)$, is the result of "pairing" the transition from 2 to 3 labeled by $(0, 1)$, and the one from 3 to 2 labeled by $(1, 0)$. Observe that $T \circ T$ is not deterministic, because for instance $[1, 1]$ is the source of two transitions labeled by $(0, 0)$, even

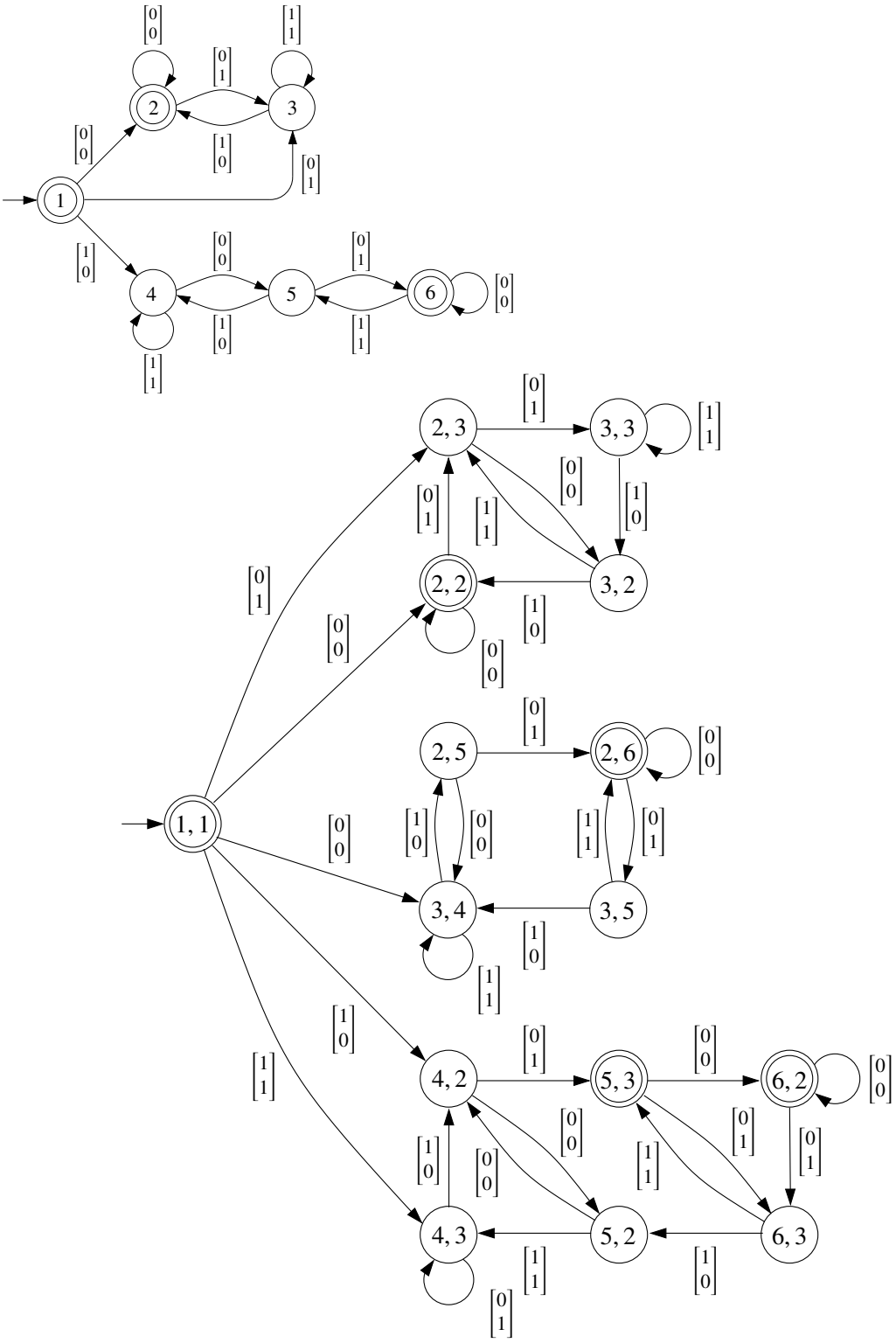


Figure 5.3: A transducer for $f(f(n))$.

though T is deterministic. This transducer recognizes the relation $\{(n, f(f(n))) \mid n \in \mathbb{N}\}$. A little calculation gives

$$f(f((n)) = \begin{cases} n/4 & \text{if } n \equiv 0 \, mod \, 4 \\ 3n/2 + 1 & \text{if } n \equiv 2 \, mod \, 4 \\ 3n/2 + 1/2 & \text{if } n \equiv 1 \, mod \, 4 \text{ or } n \equiv 3 \, mod \, 4 \end{cases}$$

The three components of the transducer reachable from the state $[1, 1]$ correspond to these three cases. □

Post(X,R) and Pre(X, R) Given an NFA $A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$ recognizing a regular set $X \subseteq U$ and a transducer $T_2 = (Q_2, \Sigma \times \Sigma, \delta_2, q_{02}, F_2)$ recognizing a regular relation $R \subseteq U \times U$, we construct an NFA B recognizing the set $post_R(U)$. It suffices to slightly modify the join operation. The algorithm *Post*(A_1,T_2) is the result of replacing lines 7-8 of *Join*() by

```
7    for all (q1, c, q1') ∈ δ1, (q2, (c, b), q2') ∈ δ2 do
8      add ([q1, q2], b, [q1', q2']) to δ
```

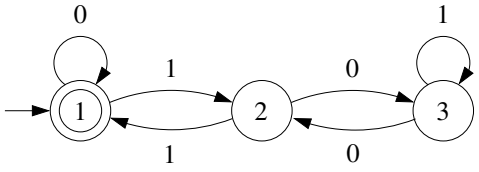
As for the join operation, the resulting NFA has to be postprocessed, closing it with respect to the padding symbol.

In order to construct an NFA recognizing $pre_R(X)$, we replace lines 7-8 by

```
7    for all (q1, (a, c), q1') ∈ δ1, (q2, c, q2') ∈ δ2 do
8      add δ to ([q1, q2], a, [q1', q2'])
```

Notice that both post and pre are computed with the same complexity as the pairing construction, namely, the product of the number of states of transducer and NFA.

Example 5.9 We construct an NFA recognizing the image under the Collatz function of all multiples of 3, i.e., the set $\{f(3n) \mid n \in \mathbb{N}\}$. For this, we first need an automaton recognizing the set Y of all *lsbf* encodings of the multiples of 3. The following DFA does the job:



For instance, this DFA recognizes 0011 (encoding of 12) and 01001 (encoding of 18), but not 0101 (encoding of 10). We now compute $post_R(Y)$, where, as usual, $R = \{(n, f(n)) \mid n \in \mathbb{N}\}$. The result is the NFA shown in Figure 5.4. For instance, the transition $[1, 1] \xrightarrow{1} [1, 3]$ is generated by the transitions $1 \xrightarrow{0} 1$ of the DFA and $1 \xrightarrow{(0,1)} 3$ of the transducer for the Collatz function. State $[2, 3]$ becomes final due to the closure with respect to the padding symbol 0.

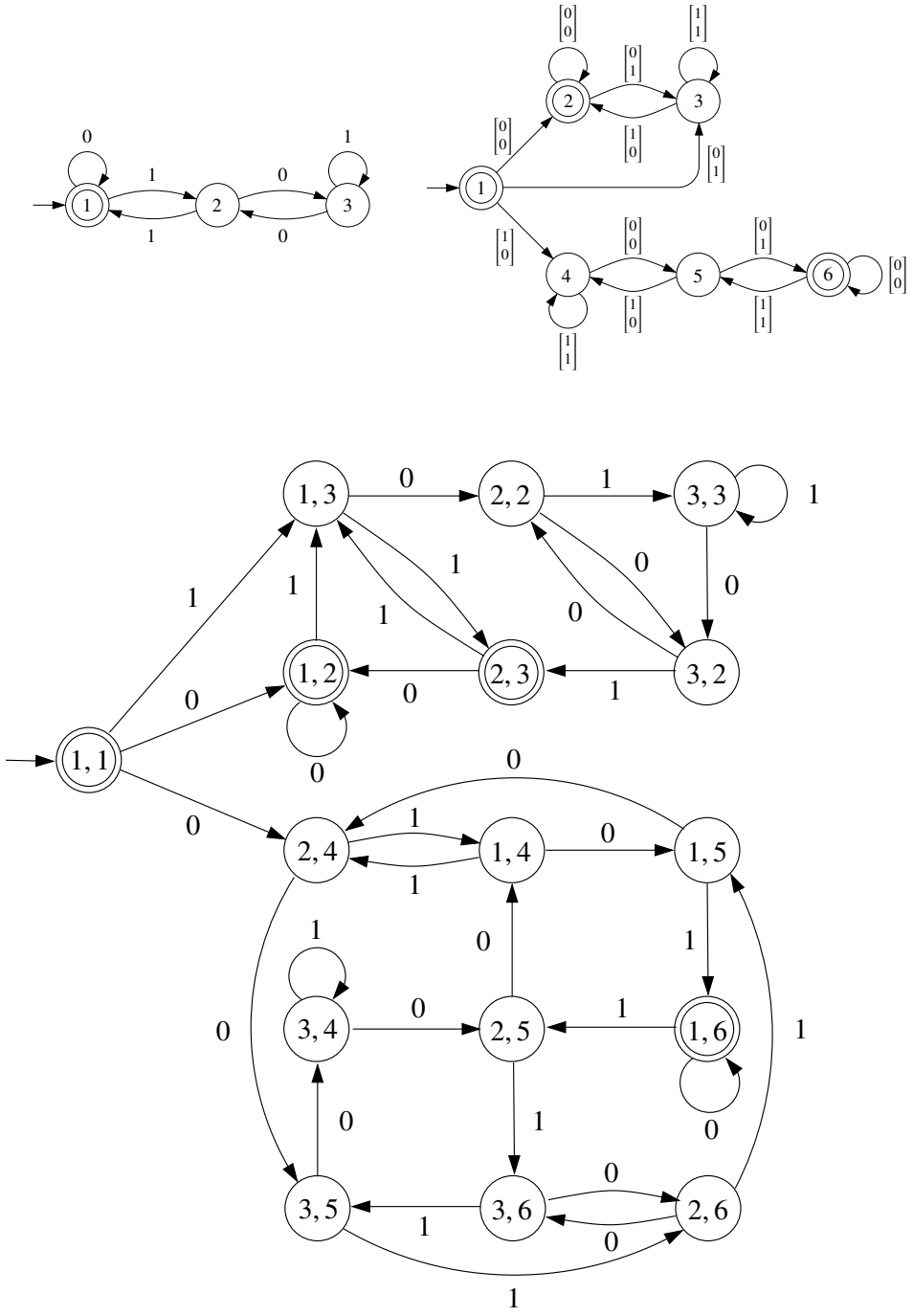


Figure 5.4: Computing $f(n)$ for all multiples of 3.

The NFA of Figure 5.4 is not difficult to interpret. The multiples of 3 are the union of the sets $\{6k \mid k \geq 0\}$, all whose elements are even, and the set $\{6k + 3 \mid k \geq 0\}$, all whose elements are odd. Applying f to them yields the sets $\{3k \mid k \geq 0\}$ and $\{18k + 10 \mid k \geq 0\}$. The first of them is again the set of all multiples of 3, and it is recognized by the upper part of the NFA. (In fact, this upper part is a DFA, and if we minimize it we obtain exactly the DFA given above.) The lower part of the NFA recognizes the second set. The lower part is minimal; it is easy to find for each state a word recognized by it, but not by the others.

It is interesting to observe that an explicit computation of the set $\{f(3k) \mid k \geq 0\}$) in which we apply f to each multiple of 3 does not terminate, because the set is infinite. In a sense, our solution "speeds up" the computation by an infinite factor! □

5.4 Relations of Higher Arity

The implementations described in the previous sections can be easily extended to relations of higher arity over the universe U. We briefly describe the generalization.

Fix an encoding of the universe U over the alphabet Σ with padding symbol #. A tuple $(w_1, \ldots, w_k)$ of words over Σ encodes the tuple $(x_1, \ldots, x_k) \in U^k$ if w_i encodes x_i for every $1 \leq i \leq k$, and $w_1, \ldots, w_k$ have the same length. A k-transducer over Σ is an NFA over the alphabet Σ^k. Acceptance of a k-transducer is defined as for normal transducers.

Boolean operations are defined as for NFAs. The projection operation can be generalized to projection over an arbitrary subset of components. For this, given an index set $I = \{i_1, \ldots, i_n\} \subseteq \{1, \ldots, k\}$, let $\vec{x}_I$ denote the projection of a tuple $\vec{x} = (x_1, \ldots, x_k) \in U^k$ over I, defined as the tuple $(x_{i_1}, \ldots, x_{i_n}) \in U^n$. Given a relation $R \subseteq U$, we define

Projection_I(R): returns the set $\pi_I(R) = \{\vec{x}_I \mid \vec{x} \in R\}$.

The operation is implemented analogously to the case of a binary relation. Given a k-transducer T recognizing R, the n-transducer recognizing **Projection_P**(R) is computed as follows:

- Replace every transition $(q, (a_1, \ldots, a_k), q')$ of T by the transition $(q, (a_{i_1}, \ldots, a_{i_n}), q')$.
- Compute the PAD-closure of the result: for every transition $(q, (\#, \ldots, \#), q')$, if q' is a final state, then add q to the set of final states.

The join operation can also be generalized. Given two tuples $\vec{x} = (x_1, \ldots, x_n)$ and $\vec{y} = (y_1, \ldots, y_m)$ of arities n and m, respectively, we denote the tuple $(x_1, \ldots, x_n, y_1, \ldots, y_m)$ of dimension $n + m$ by $\vec{x} \cdot \vec{y}$. Given relations $R_1 \subseteq U^{k_1}$ and $R_2 \subseteq U^{k_2}$ of arities k_1 and k_2, respectively, and index sets $I_1 \subseteq \{1, \ldots, k_1\}$, $I_2 \subseteq \{1, \ldots, k_2\}$ *of the same cardinality*, we define

Join_I(R_1, R_2): returns $R_1 \circ_{I_1, I_2} R_2 = \{\vec{x}_{K_1 \setminus I_1} \, \vec{x}_{K_2 \setminus I_2} \mid \exists \vec{x} \in R_1, \vec{y} \in R_2 : \quad \vec{x}_{I_1} = \vec{y}_{I_2}\}$

The arity of **Join_I**(R_1, R_2) is $k_1 + k_2 - |I_1|$. The operation is implemented analogously to the case of binary relations. We proceed in two steps. The first step constructs a transducer according to the following rule:

> If the transducer recognizing R_1 has a transition $(q, \vec{a}, q')$, the transducer recognizing R_2 has a transition $(r, \vec{b}, r')$, and $\vec{a}_{I_1} = \vec{b_2}$, then add to the transducer for **Join_I**(R_1, R_2) a transition $([q, r], \vec{a}_{K_1 \setminus I_1} \cdot \vec{b}_{K_2 \setminus I_2}, [q', r'])$.

In the second step, we compute the PAD-closure of the result. The generalization of the **Pre** and **Post** operations is analogous.

Exercises

Exercise 52 As we have seen, the application of the **Post**, **Pre** operations to transducers requires to compute the padding closure in order to guarantee that the resulting automaton accepts either all or none of the encodings of a object. The padding closure has been defined for encodings where padding occurs *on the right*, i.e., if w encodes an object, then so does $w\#^k$ for every $k \geq 0$. However, in some natural encodings, like the *most-significant-bit-first* encoding of natural numbers, padding occurs *on the left*. Give an algorithm for calculating the padding closure of a transducer when padding occurs on the left.

Exercise 53 Let A be an NFA over the alphabet Σ.

- Show how to construct a transducer T over the alphabet $\Sigma \times \Sigma$ such that $(w, v) \in \mathcal{L}(T)$ iff $wv \in L(A)$ and $|w| = |v|$.
- Give an algorithm that accepts an NFA A as input and returns an NFA $A/2$ such that $L(A/2) = \{w \in \Sigma^* \mid \exists v \in \Sigma^* \ : \ wv \in L(A) \wedge |w| = |v|\}$.

Exercise 54 In phone dials letters are mapped into digits as follows:

ABC $\mapsto$ 2	DEF $\mapsto$ 3	GHI $\mapsto$ 4	JKL $\mapsto$ 5
MNO $\mapsto$ 6	PQRS $\mapsto$ 7	TUV $\mapsto$ 8	WXYZ $\mapsto$ 9

This map can be used to assign a telephone number to a given word. For instance, the number for AUTOMATON is 288662866.

Consider the problem of, given a telephone number, finding the set of English words that are mapped into it. For instance, the set of words mapping to 233 contains at least ADD, BED, and BEE. Assume a DFA N over the alphabet $\{A, \ldots, Z\}$ recognizing the set of all English words is given. Show how to construct a nondeterministic transducer over $\Sigma \times \Sigma$ for $\Sigma = \{A, \ldots, Z, 0, \ldots, 9\}$ recognizing the set of pairs (n, w) such that $n \in \{0, \ldots, 9\}^*$ and $w \in \{A, ldots, Z\}^*$ and w is mapped to n.

Exercise 55 We have defined transducers as NFAs whose transitions are labeled by pairs of symbols $(a, b) \in \Sigma \times \Sigma$. With this definition transducers can only accept pairs of words $(a_0 a_1 \ldots a_l, b_0 b_1 \ldots b_l)$ of the same length.

A *ε-transducer* is a NFA whose transitions are labeled by elements of $(\Sigma \cup \{\varepsilon\}) \times (\Sigma \cup \{\varepsilon\})$. An ε-transducer accepts a pair (w, w') of words if it has a run

$$q_0 \xrightarrow{(a_0,b_0)} q_1 \xrightarrow{(a_1,b_1)} \ldots \xrightarrow{(a_n,b_n)} q_n \text{ with } a_i, b_i \in \Sigma \cup \{\varepsilon\}$$

such that $w = a_0 a_1 \ldots a_n$ and $w' = b_0 b_1 \ldots b_n$. Note that $|w| \leq n$ and $|w'| \leq n$. The relation accepted by the ε-transducer T is denoted by $L(T)$.

1. Construct ε-transducers T_1, T_2 recognizing the relations $R_1 = \{(a^n b^m, c^{2n}) \mid n, m \geq 0\}$, and $R_2 = \{(a^n b^m, c^{2m}) \mid n, m \geq 0\}$.

2. Apply *IntersNFA* to T_1 and T_2. Which is the language recognized by the resulting ε-transducer?

3. Show that no ε-transducer recognizes $R_1 \cap R_2$.

Exercise 56 Transducers can be used to capture the behaviour of simple programs. Let P be the following program:

```
1   x ←?
2   write x
3   while true do
4     do
5       read y
6     until x = y
7     if eof then
8       write y
9       end
10    do
11      x ← x − 1
        or
12      y ← y + 1
13    until x ≠ y
```

P communicates with the environment through the boolean variables `x` and `y`, both with 0 as initial value. Let $[i, x, y]$ denote the state of P in which the next instruction to be executed is the one at line i, and the values of `x` and `y` are x and y, respectively. The initial state of P is $[1, 0, 0]$. By executing the first instruction P moves nondeterministically to one of the states $[2, 0, 0]$ and $[2, 1, 0]$; no input symbol is read and no output symbol is written, and so the transition relation

δ for P contains the transitions $([1,0,0],(\varepsilon,\varepsilon),[2,0,0])$ and $([1,0,0],(\varepsilon,\varepsilon),[2,1,0])$. Similarly, by executing its second instruction, the program P moves from $[2,1,0]$ to $[3,1,0]$ while reading nothing and writing 1. Hence, δ also contains the transition rule $([2,1,0],(\varepsilon,1),[3,1,0])$.

1. Draw an ε-transducer modelling the behaviour of P.
2. Can an overflow error occur?
3. What are the possible values of `x` and `y` upon termination, i.e. upon reaching **end**?
4. Is there an execution during which P reads 101 and writes 01?
5. Let I and O be regular sets of inputs and outputs, respectively. Think of O as a set of dangerous outputs that we want to avoid. We wish to prove that the inputs from I are safe, i.e. that none of the dangerous outputs can occur. Describe an algorithm that decides, given I and O, whether there are $i \in I$ and $o \in O$ such that (i,o) is accepted.

Exercise 57 (Inspired by a paper by Galwani al POPL'11.) Consider transducers whose transitions are labeled by elements of $(\Sigma \cup \{\varepsilon\}) \times (\Sigma^* \cup \{\varepsilon\})$. Intuitively, at each transition these transducers read one letter or no letter, an write a string of arbitrary length. These transducers can be used to perform operations on strings like, for instance, capitalizing all the words in the string: if the transducer reads, say, "singing in the rain", it writes "Singing In The Rain". Give transducers for the following operations, each of which is informally defined by means of two examples. In each example, if the transducer reads the string on the left then it writes the string on the right.

```
            Company\Code\index.html    Company\Code
          Company\Docs\Spec\specs.doc  Company\Docs\Spec\

    International Business Machines    IBM
Principles Of Programming Languages    POPL

                Oege  De     Moor      Oege De Moor
      Kathleen  Fisher   AT&T Labs     Kathleen Fisher AT&T Labs

                     Eran Yahav       Yahav, E.
                     Bill Gates       Gates, B.

                  004989273452        +49 89 273452
                 (00)4989273452       +49 89 273452
                        273452        +49 89 273452
```

Chapter 6

Finite Universes

In Chapter 3 we proved that every regular language has a unique minimal DFA. A natural question is whether the operations on languages and relations described in Chapters 4 and 5 can be implemented using minimal DFAs and minimal deterministic transducers as data structure.

The implementations of (the first part of) Chapter 4 accept and return DFAs, but do not preserve minimality: even if the arguments are minimal DFAs, the result may be non-minimal (the only exception was complementation). So, in order to return the minimal DFA for the result an extra minimization operation must be applied. The situation is worse for the projection and join operations of Chapter 5, because they do not even preserve determinacy: the result of projecting a deterministic transducer or joining two of them may be nondeterministic. In order to return a minimal DFA it is necessary to first determinize, at exponential cost in the worst case, and then minimize.

In this chapter we present implementations that *directly* yield the minimal DFA, with no need for an extra minimization step, for the case in which the universe U of objects is finite. When the universe is finite, all objects can be encoded by words *of the same length*, and this common length is known *a priori*. Every subset of objects can then be encoded by a *fixed-length* language.

Definition 6.1 *A language $L \subseteq \Sigma^*$ has* length $n \geq 0$ *if it is empty and $n = 0$, or if it is nonempty and all words of L have length n. If L has length n for some $n \geq 0$, then we say that L is a* fixed-length *language, or that L* has fixed-length.

Notice that every fixed-length language contains only finitely many words, and so it is automatically regular. Observe also that there are exactly two languages of length 0: the empty language $\emptyset$, and the language $\{\varepsilon\}$ containing only the empty word.

6.1 The Master Automaton

The *master automaton* over an alphabet Σ is a deterministic automaton with an infinite number of states, but no initial state. As in the case of canonical DAs, the states are languages.

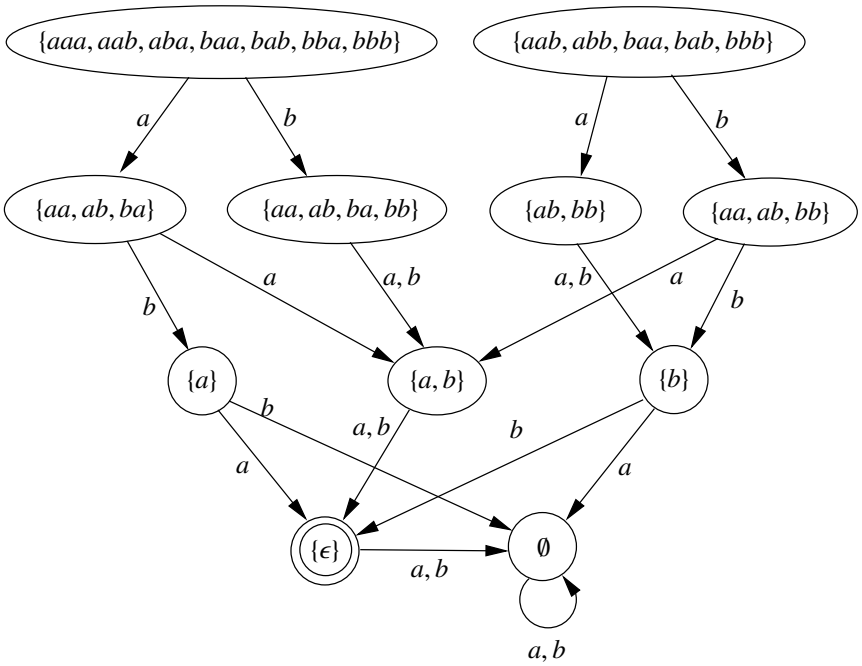


Figure 6.1: A fragment of the master automaton for the alphabet $\{a, b\}$

For the definition, recall the notion of *residual* with respect to a letter: given a language $L \subseteq \Sigma^*$ and $a \in \Sigma$, its residual with respect to a is the language $L^a = \{w \in \Sigma^* \mid aw \in L\}$. Recall that, in particular, we have $\emptyset^a = \{\epsilon\}^a = \emptyset$. A simple but important observation is that if L has fixed-length, then so does L^a.

Definition 6.2 *The* master automaton *over the alphabet Σ is the tuple $M = (Q_M, \Sigma, \delta_M, F_M)$, where*

- *Q_M is is the set of all fixed-length languages over Σ;*
- *$\delta\colon Q_M \times \Sigma \to Q_M$ is given by $\delta(L, a) = L^a$ for every $q \in Q_M$ and $a \in \Sigma$;*
- *F_M is the singleton set containing the language $\{\epsilon\}$ as only element.*

Example 6.3 Figure 6.1 shows a small fragment of the master automaton for the alphabet $\Sigma = \{a, b\}$. Notice that M is almost acyclic. More precisely, the only cycles of M are the self-loops corresponding to $\delta_M(\emptyset, a) = \emptyset$ for every $a \in \Sigma$. □

The following proposition was already proved in Chapter 3, but with slightly different terminology.

Proposition 6.4 *Let L be a fixed-length language. The language recognized from the state L of the master automaton is L.*

Proof: By induction on the length n of L. If $n = 0$, then $L = \{\epsilon\}$ or $L = \emptyset$, and the result is proved by direct inspection of the master automaton. For $n > 0$ we observe that the successors of the initial state L are the languages L^a for every $a \in \Sigma$. Since, by induction hypothesis, the state L^a recognizes the language L^a, the state L recognizes the language L. □

By this proposition, we can look at the master automaton as a structure containing DFAs recognizing all the fixed-length languages. To make this precise, each fixed-length language L determines a DFA $A_L = (Q_L, \Sigma, \delta_L, q_{0L}, F_L)$ as follows: Q_L is the set of states of the master automaton reachable from the state L; q_{0L} is the state L; δ_L is the projection of δ_M onto Q_L; and $F_L = F_M$. It is easy to show that A_L is the *minimal* DFAs recognizing L:

Proposition 6.5 *For every fixed-language L, the automaton A_L is the minimal DFA recognizing L.*

Proof: By definition, distinct states of the master automaton are distint languages. By Proposition 6.4, distinct states of A_L recognize distinct languages. By Corollary 3.8 (a DFA is minimal if and only if distinct states recognized different languages) A_L is minimal. □

6.2 A Data Structure for Fixed-length Languages

Proposition 6.5 allows to define a data structure for representing *finite sets of fixed-length languages*, all of them *of the same length*. Loosely speaking, the structure representing the languages $\mathcal{L} = \{L_1, \ldots, L_m\}$ is the fragment of the master automaton containing the states recognizing $L_1, \ldots, L_n$ and their descendants. It is a DFA with multiple initial states, and for this reason we call it *the multi-DFA for* $\mathcal{L}$. Formally:

Definition 6.6 *Let $\mathcal{L} = \{L_1, \ldots, L_n\}$ be a set of languages of the same length over the same alphabet Σ. The* multi-DFA *$A_{\mathcal{L}}$ is the tuple $A_{\mathcal{L}} = (Q_{\mathcal{L}}, \Sigma, \delta_{\mathcal{L}}, Q_{0\mathcal{L}}, F_{\mathcal{L}})$, where $Q_{\mathcal{L}}$ is the set of states of the master automaton reachable from at least one of the states $L_1, \ldots, L_n$; $Q_{0\mathcal{L}} = \{L_1, \ldots, L_n\}$; $\delta_{\mathcal{L}}$ is the projection of δ_M onto $Q_{\mathcal{L}}$; and $F_{\mathcal{L}} = F_M$.*

Example 6.7 Figure 6.2 shows (a DFA isomorphic to) the multi-DFA for the set $\{L_1, L_2, L_3\}$, where $L_1 = \{aa, ba\}$, $L_2 = \{aa, ba, bb\}$, and $L_3 = \{ab, bb\}$. In order to simplify the picture the state for the empty language has been omitted, as well as the transitions leading to it. □

In order to manipulate multi-DFAs we represent them as a *table of nodes*. Assume $\Sigma = \{a_1, \ldots, a_m\}$. A *node* is a pair $\langle q, s \rangle$, where q is a *state identifier* and $s = (q_1, \ldots, q_m)$ is the *successor tuple* of the node. The multi-DFA is represented by a table containing a node for each state, but the state corresponding to the empty language[1]. The table for the multi-DFA of Figure 6.2 is

[1]The reason for this is technical convenience: it ensures that different rows of the table have different successor tuples.

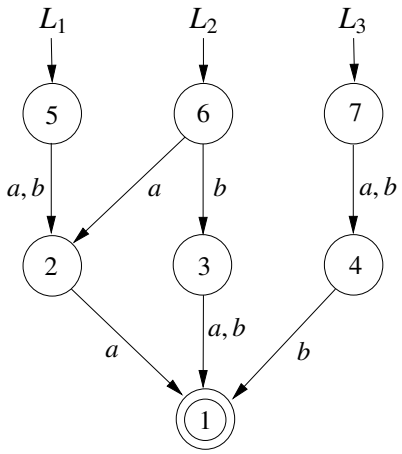


Figure 6.2: The multi-DFA for $\{L_1, L_2, L_3\}$ with $L_1 = \{aa, ba\}$, $L_2 = \{aa, ba, bb\}$, and $L_3 = \{ab, bb\}$.

Ident.	a-succ	b-succ
1	0	0
2	1	0
3	1	1
4	0	1
5	2	2
6	2	3
7	4	4

The procedure $make[T](s)$**.** The algorithms on multi-DFAs use a procedure $make[T](s)$ that returns the state of T having s as successor tuple, if such a state exists; otherwise, it adds a new node $\langle q, s\rangle$ to T, where q is a fresh state identifier (different from all other state identifiers in T) and returns q. The procedure assumes that all the states of the tuple s appear in T.[2] For instance, if T is the table above, then $make[T](2, 2)$ returns 5, but $make[T](3, 2)$ adds a new row, say $8, 3, 2$, and returns 8.

6.3 Operations on fixed-length languages

All operations assume that the input fixed-length language(s) is (are) given as multi-DFAs represented as a table of nodes. Nodes are pairs of state identifier and successor tuple.

The key to all implementations is the fact that if L is a language of length $n \geq 1$, then L^a is a language of length $n - 1$ or of length 0. This allows to design recursive algorithms that directly

[2]Notice that the procedure makes use of the fact that there is at most one node with a given successor tuple.

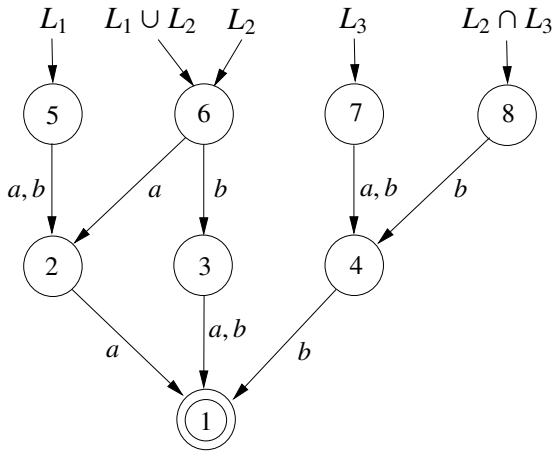


Figure 6.3: The multi-DFA for $\{L_1, L_2, L_3, L_1 \cup L_2, L_2 \cap L_3\}$

compute the result for languages of length 0, and reduce the problem of computing the result for languages of length n to the same problem for languages of smaller length.

Fixed-length membership. The operation is implemented as for DFAs. The complexity is linear in the size of the input.

Fixed-length binary boolean operations. Implementing a boolean operation on multi-DFAs corresponds to possibly extending the multi-DFA, and returning the state corresponding to the result of the operation. This is best explained by means of an example. Consider again the multi-DFA of Figure 6.2. An operation like **Union**(L_1, L_2) gets the initial states 5 and 6 as input, and returns the state recognizing $L_1 \cup L_2$; since $L_1 \cup L_2 = L_2$, the operation returns state 6. However, if we take **Intersection**(L_2, L_3), then the multi-DFA does not contain any state recognizing it. In this case the operation extends the multi-DFA for $\{L_1, L_2, L_3\}$ to the multi-DFA for $\{L_1, L_2, L_3, L_2 \cup L_3\}$, shown in Figure 6.3, and returns state 8. So **Intersection**(L_2, L_3) not only returns a state, but also has a side effect on the multi-DFA underlying the operations.

Given two fixed-length languages L_1, L_2 *of the same length* and a boolean operation $\odot$, there is a generic algorithm that returns the state of the master automaton recognizing $L_1 \mathbin{\widehat{\odot}} L_2$. Let us consider the case of intersection, the other cases being similar. The properties

(1) if $L_1 = \emptyset$, then $L_1 \cap L_2 = \emptyset$;

(2) if $L_2 = \emptyset$, then $L_1 \cap L_2 = \emptyset$;

(3) if $L_1 \neq \emptyset \neq L_2$, then $(L_1 \cap L_2)^a = L_1^a \cap L_2^a$ for every $a \in \Sigma$;

lead to the recursive algorithm $inter[T](q_1, q_2)$ shown in Table 6.1. Assume the states q_1, q_2 recognize the languages L_1, L_2. The algorithm returns the state identifier $q_{L_1 \cup L_2}$. If $q_1 = q_\emptyset$, then $L_1 = \emptyset$,

which implies $L_1 \cap L_2 = \emptyset$. So the algorithm returns the state identifier $q_\emptyset$. If $q_2 = q_\emptyset$, the algorithm also returns $q_\emptyset$. If $q_1 \neq q_\emptyset \neq q_2$, then the algorithm computes the state identifiers $r_1, \ldots, r_m$ recognizing the languages $(L_1 \cap L_2)^{a_1}, \ldots, (L_1 \cap L_2)^{a_m}$, and returns $make[T](r_1, \ldots, r_n)$ (creating a new node if no node of T has $(r_1, \ldots, r_n)$ as successor tuple). But how does the algorithm compute the state identifier of $(L_1 \cap L_2)^{a_i}$? By equation (3) above, we have $(L_1 \cap L_2)^{a_i} = L_1^{a_i} \cap L_2^{a_i}$, and so the algorithm computes the state identifier of $L_1^{a_i} \cap L_2^{a_i}$ by a recursive call $inter[T](q_1^{a_i}, q_2^{a_i})$.

The only remaining point is the rôle of the table G. The algorithm uses memoization to avoid recomputing the same object. The table G is initially empty. When $inter[T](q_1, q_2)$ is computed for the first time, the result is memoized in $G(q_1, q_2)$. In any subsequent call the result is not recomputed, but just read from G. For the complexity, let n_1, n_2 be the number of states of T reachable from the state q_1, q_2. It is easy to see that every call to *inter* receives as arguments states reachable from q_1 and q_2, respectively. So *inter* is called with at most $n_1 \cdot n_2$ possible arguments, and the complexity is $O(n_1 \cdot n_2)$.

inter$[T](q_1, q_2)$
Input: table T, states q_1, q_2 of T
Output: state recognizing $L(q_1) \cap L(q_2)$

```
1  if G(q1, q2) is not empty  then return G(q1, q2)
2  if q1 = q∅ ∨ q2 = q∅ then return q∅
3  if q1 ≠ q∅ ∧ q2 ≠ q∅ then
4     for all i = 1, ..., m do r_i ← inter[T](q1^{a_i}, q2^{a_i})
5     G(q1, q2) ← make[T](r1, ..., rm)
6     return G(q1, q2)
```

Table 6.1: $inter[T](q_1, q_2)$

Algorithm *inter*() is generic: in order to obtain an algorithm for another binary operator it suffices to change lines 2 and 3. If we are only interested in intersection, then we can easily gain a more efficient version. For instance, we know that $inter[T](q_1, q_2)$ and $inter[T](q_2, q_1)$ return the same state, and so we can improve line 1 by checking not only if $G(q_1, q_2)$ is nonempty, but also if $G(q_2, q_1)$ is. Also, $inter[T](q, q)$ always returns q, we do not need to compute anything either.

Example 6.8 Consider the multi-DFA at the top of Figure 6.4, but without the blue states. State 0, accepting the empty language, is again not shown. Let T be the table for this multi-DFA. The tree at the bottom of the figure graphically describes the run of $inter[T](12, 13)$ (that is, we compute the node for the intersection of the languages recognized from states 12 and 13). A node $q, q' \mapsto q''$ of the tree corresponds to a recursive call of *inter*() with arguments q and q', and q'' is the state returned by the call. For instance, the node 2,4 $\mapsto$ 2 indicates that *inter*() is called with arguments 2 and 4 and the call returns state 2. Let us see why. The call *inter*(2, 4) produces two recursive calls, first *inter*(1, 1) (the a-successors of 2 and 4), and then *inter*(0, 1). The first call returns 1, and the second 0. Therefore the call *inter*(2, 4) returns a state with state 1 as a-successor and state 0 as

b-successor. This state already exists, it is state 2. So *inter*(2, 4) returns state 2. On the other hand, *inter*(9,10) creates and returns a new state. The two "children calls" return states 5 and 6, and so a new state with state 5 and 6 as a- and b-successors must be created.

The pink nodes correspond to calls that have already been computed, and for which *inter*() just looks up the result in G. The green nodes correspond to calls that are computed by *inter*(), but not by the more efficient version. For instance, the result of the call *inter*(4,4) at the bottom right can be returned immediately. □

Fixed-length complement. The complement $\Sigma^* \setminus L$ of a fixed-length language L is *not* a fixed-length language, and so the complement operation is not well defined. We consider its fixed-length version, which returns the state recognizing the language $\Sigma^n \setminus L$, called the *fixed-length complement*. We abuse language and denote the fixed-length complement of L by $\overline{L}$. (In this chapter $\overline{L}$ always has this meaning.) Notice that $\Sigma^0 = \{\epsilon\}$. We have the following equations:

- if $L = \emptyset$, then $\overline{L} = \{\epsilon\}$;
- if $L = \{\epsilon\}$, then $\overline{L} = \emptyset$; and
- if $\emptyset \neq L \neq \{\epsilon\}$, then $\left(\overline{L}\right)^a = \overline{L^a}$.
  (Observe that $w \in \left(\overline{L}\right)^a$ iff $aw \notin L$ iff $w \notin L^a$ iff $w \in \overline{L^a}$.)

Proceeding as for the binary boolean operations, we get the algorithm of Table 6.2. If the master automaton has n states reachable from q, then the operation has complexity $\mathcal{O}(n)$.

comp$[T](q)$
Input: table T, state q of T
Output: state recognizing $\overline{L(q)}$

```
1  if G(q) is not empty then return G(q)
2  if q = q_∅ then return q_ε
3  else if q = q_ε then return q_∅
4  else  / * q ≠ q_∅ and q ≠ q_ε * /
5     for all i = 1,...,m do r_i ← comp[T](q^{a_i})
6     G(q) ← make[T](r_1,...,r_m)
7     return G(q)
```

Table 6.2: *comp*$[T](q)$

Fixed-length emptiness. A fixed-language language is empty if and only if the node representing it has $q_\emptyset$ as state identifier, and so we get the algorithm of Table 6.3.

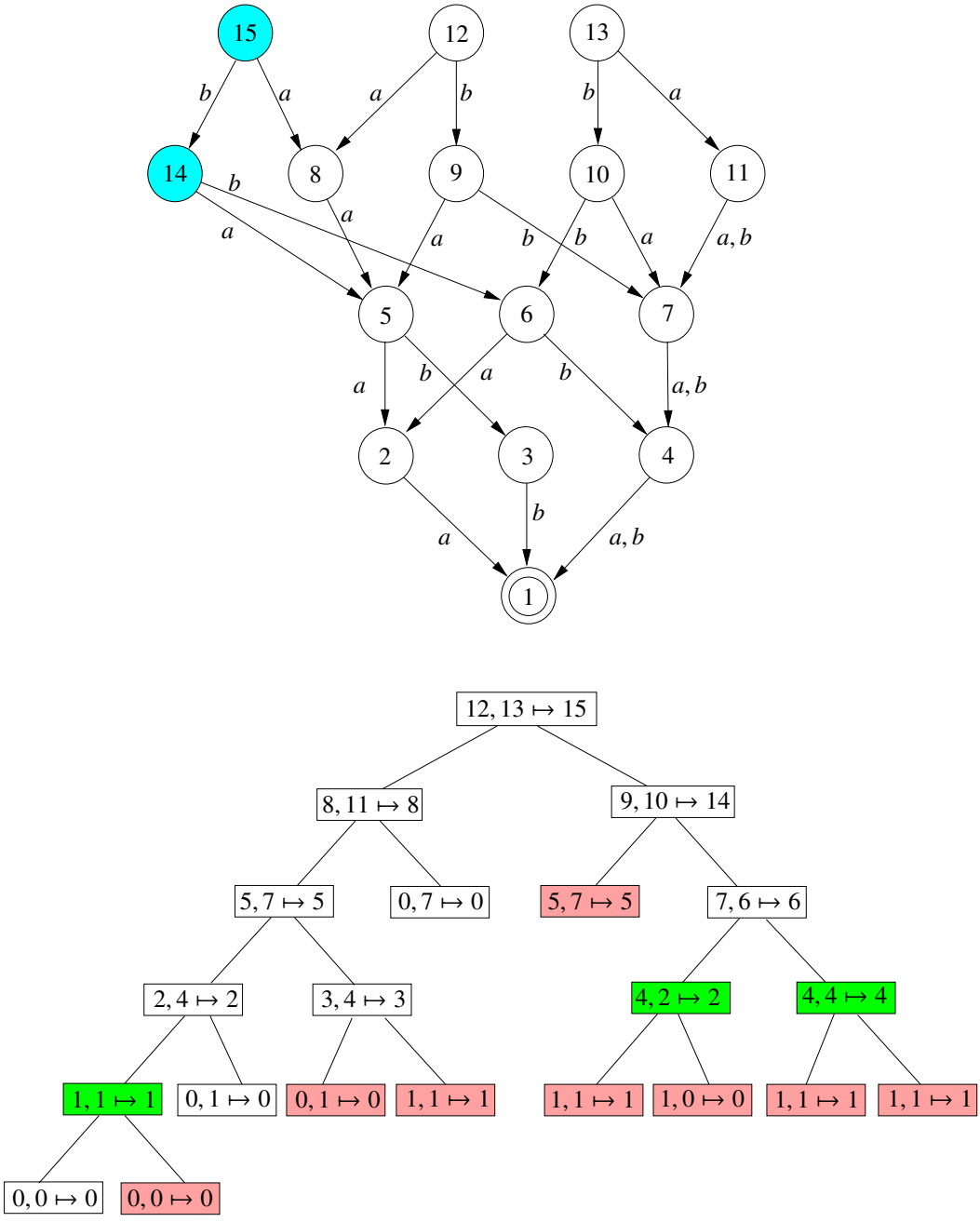


Figure 6.4: An execution of *inter*().

empty$[T](q)$
Input: table T, state q of T
Output: **true** if $L(q) = \emptyset$, **false** otherwise
1 **return** $q = q_\emptyset$

Table 6.3: *empty*$[T](q)$

Fixed-length universality. Like in the case of complement, a fixed-length language cannot be universal by definition. However, we may ask whether it is *fixed-length universal*. A language L of length n is fixed-length universal if $L = \Sigma^n$. The universality of a language of length n recognized by a state q can be checked in time $\mathcal{O}(n)$. It suffices to check for all states reachable from q, with the exception of $q_\emptyset$, that no transition leaving them leads to $q_\emptyset$. More systematically, the equations

- if $L = \emptyset$, then L is not universal;
- if $L = \{\epsilon\}$, then L is universal;
- if $\emptyset \neq L \neq \{\epsilon\}$, then L is universal iff L^a is universal for every $a \in \Sigma$.

lead to the algorithm of Table 6.4.

univ$[T](q)$
Input: table T, state q of T
Output: **true** if $L(q)$ is fixed-length universal,
false otherwise
1 **if** $G(q)$ is not empty **then return** $G(q)$
2 **if** $q = q_\emptyset$ **then return false**
3 **else if** $q = q_\epsilon$ **then return true**
4 **else** / $*$ $q \neq q_\emptyset$ and $q \neq q_\epsilon$ $*$ /
5 $G(q) \leftarrow$ **and**$(univ[T](q^{a_1}), \ldots, univ[T](q^{a_m}))$
6 **return** $G(q)$

Table 6.4: The algorithm *univ*$[T](q)$

Fixed-length inclusion. Given two languages $L_1, L_2 \subseteq \Sigma^n$, in order to check $L_1 \subseteq L_2$ we compute $L_1 \cap L_2$ and check whether it is equal to L_1 using the equality check shown next. The complexity is dominated by the complexity of computing the intersection.

Fixed-length equality. Since the minimal DFA recognizing a language is unique, two languages are equal if and only if the nodes representing them have the same state identifier, leading to the

constant time algorithm at the top of Figure 6.5. This algorithm, however, assumes that the two input nodes come from the same table T. If they come from two different tables T_1, T_2, then, since state identifiers can be assigned in both tables in different ways, it is necessary to check if the DFA rooted at the states q_1 and q_2 are isomorphic. This is done by the implementation $eq[T_1, T_2](q_1, q_2)$ at the bottom of the figure, which assumes that q_i belongs to the table T_i, and that both tables assign state identifiers $q_{\emptyset 1}$ and $q_{\emptyset 2}$ to the empty language.

$eq[T](q_1, q_2)$
Input: table T, states q_1, q_2 of T
Output: **true** if $L(q_1) = L(q_2)$, **false** otherwise
1 **return** $q_1 = q_2$

$eq[T_1, T_2](q_1, q_2)$
Input: tables T_1, T_2, states q_1 of T_1, q_2 of T_2
Output: **true** if $L(q_1) = L(q_2)$, **false** otherwise
1 **if** $G(q_1, q_2)$ is not empty **then return** $G(q_1, q_2)$
2 **if** $q_1 = q_{\emptyset 1}$ and $q_2 = q_{\emptyset 2}$ **then** $G(q_1, q_2) \leftarrow$ `true`
3 **else if** $q_1 = q_{\emptyset 1}$ and $q_2 \neq q_{\emptyset 2}$ **then** $G(q_1, q_2) \leftarrow$ `false`
4 **else if** $q_1 \neq q_{\emptyset 1}$ and $q_2 = q_{\emptyset 2}$ **then** $G(q_1, q_2) \leftarrow$ `false`
5 **else** / * $q_1 \neq q_{\emptyset 1}$ and $q_2 \neq q_{\emptyset 2}$ * /
6 $G(q_1, q_2) \leftarrow$ **and**$(\mathtt{eq}(q_1^{a_1}, q_2^{a_1}), \ldots, \mathtt{eq}(q_1^{a_m}, q_2^{a_m}))$
7 **return** $G(q_1, q_2)$

Table 6.5: The algorithms $eq[T](q_1, q_2)$ and $eq[T_1, T_2](q_1, q_2)$

6.4 Determinization and Minimization

Let L be a fixed-length language, and let $A = (Q, \Sigma, \delta, q_0, F)$ be a NFA recognizing L. The algorithm *det&min*(A) shown in Table 6.6 returns the state q_L of the master automaton. In other words, *det&min*(A) simultaneously determinizes and minimizes A.

The algorithm actually solves a more general problem. Given a set S of states of A, all recognizing languages *of the same length*, the language $L(S) = \bigcup_{q \in S} L(q)$ has also this common length. The heart of the algorithm is a procedure *state*(S) that returns the state $q_{L(S)}$. Since $L = L(\{q_0\})$, *det&Min*(A) just calls *state*$(\{q_0\})$.

We make the assumption that for every state q of A there is a path leading from q to some final state. This assumption can be enforced by suitable preprocessing, but usually it is not necessary; in applications, NFAs for fixed-length languages usually satisfy the property by construction. With this assumption, $L(S)$ satisfies:

- if $S = \emptyset$ then $L(S) = \emptyset$;

- if $S \cap F \neq \emptyset$ then $L(S) = \{\epsilon\}$; (since all states of S recognize fixed-length languages length, the states of F necessarily recognize $\{\epsilon\}$; since all the states of S recognize languages of the same length, and $S \cap F \neq \emptyset$, we have $L(S) = \{\epsilon\}$) NFA recognizes a fixed-length language);
- if $S \neq \emptyset$ and $S \cap F = \emptyset$, then $L(S) = \bigcup_{i=1}^{n} a_i \cdot L(S_i)$, where $S_i = \delta(S, a_i)$.

These properties lead to the recursive algorithm of Table 6.6. The procedure $state[A](S)$ uses a table G of results, initially empty. When $state[A](S)$ is computed for the first time, the result is memoized in $G(S)$, and any subsequent call directly reads the result from G.

$state[A](S)$
Input: NFA $A = (Q, \Sigma, \delta, q_0, F)$, set $S \subseteq Q$
Output: master state recognizing $L(S)$
1 **if** $G(S)$ is not empty **then return** $G(S)$
2 **else if** $S = \emptyset$ **then return** $q_\emptyset$
3 **else if** $S \cap F \neq \emptyset$ **then return** q_ϵ
4 **else** $/* S \neq \emptyset$ and $S \cap F = \emptyset */$
5 **for all** $i = 1, \ldots, m$ **do** $S_i \leftarrow \delta(S, a_i)$
6 $G(S) \leftarrow make(state[A](S_1), \ldots, state[A](S_m))$;
7 **return** $G(S)$

$det\&min(A)$
Input: NFA $A = (Q, \Sigma, \delta, q_0, F)$
Output: master state recognizing $L(A)$
1 **return** $state[A](\{q_0\})$

Table 6.6: The algorithm $det\&min(A)$.

The algorithm has exponential complexity, because, in the worst case, it may call $state[A]$(S) for every set $S \subseteq Q$.

Example 6.9 Figure 6.5 shows a NFA (upper left) and the result of applying *det&min* to it. The run of *det&min* is shown at the bottom of the figure, where, for the sake of readability, sets of states are written without the usual parenthesis (e.g. β, γ instead of $\{\beta, \gamma\}$. Observe, for instance, that the algorithm assigns to $\{\gamma\}$ the same node as to $\{\beta, \gamma\}$, because both have the states 2 and 3 as a-successor and b-successor, respectively. □

6.5 Operations on Fixed-length Relations

Fixed-length relations can be manipulated very similarly to fixed-length languages. Boolean operations are as for fixed-length languages. The join, pre, and post operations can be however

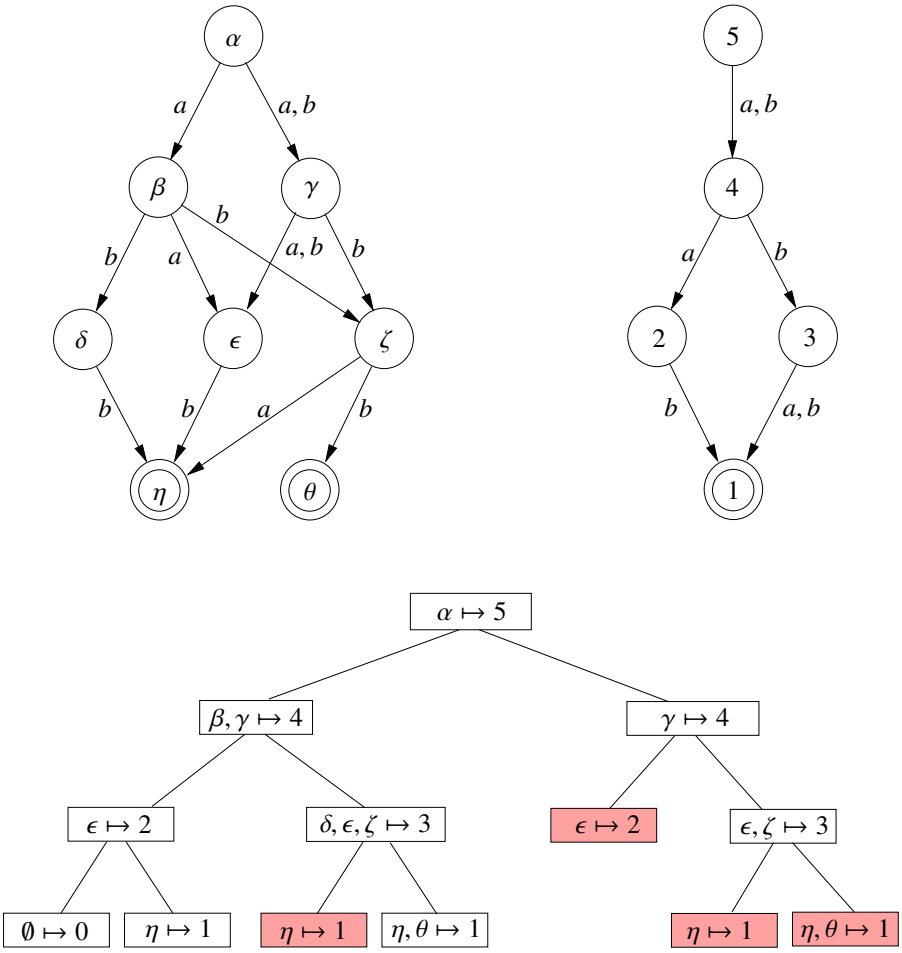


Figure 6.5: Run of *det&min*() on an NFA for a fixed-length language

implemented more efficiently as in Chapter 5.

We start with an observation on encodings. In Chapter 5 we assumed that if an element of X is encoded by $w \in \Sigma^*$, then it is also encoded by $w\#$, where # is the padding letter. This ensures that every pair $(x, y) \in X \times X$ has an encoding (w_x, w_y) such that w_x and w_y have the same length. Since in the fixed-length case all shortest encodings have the same length, the padding symbol is no longer necessary. So in this section we assume that each word or pair has exactly one encoding.

Definition 6.10 *A word relation $R \subseteq \Sigma^* \times \Sigma^*$ has* length $n \geq 0$ *if it is empty and $n = 0$, or if it is nonempty and for all pairs (w_1, w_2) of R the words w_1 and w_2 have length n. If R has length n for some $n \geq 0$, then we say that R is a* fixed-length *word relation, or that R* has fixed-length.

Recall that a transducer T accepts a pair $(w_1, w_2) \in \Sigma^* \times \Sigma^*$ if $w_1 = a_1 \dots a_n$, $w_2 = b_1 \dots b_n$, and T accepts the word $(a_1, b_1) \dots (a_n, b_n) \in \Sigma^* \times \Sigma^*$.

Definition 6.11 *A fixed-length transducer accepts a relation $R \subseteq X \times X$ if it recognizes the word relation $\{(w_x, w_y) \mid (x, y) \in R\}$.*

We define the *master transducer* over the alphabet $\Sigma \times \Sigma$. Given a language $R \subseteq \Sigma^* \times \Sigma^*$ and $a, b \in \Sigma$, we define $R^{[a,b]} = \{(w_1, w_2) \in \Sigma^* \times \Sigma^* \mid (aw_1, bw_2) \in R\}$. Notice that in particular, $\emptyset^{[a,b]} = \emptyset$, and that if R has fixed-length, then so does $R^{[a,b]}$.

Definition 6.12 *The* master transducer *over the alphabet Σ is the tuple $MT = (Q_M, \Sigma \times \Sigma, \delta_M, F_M)$, where*

- *Q_M is is the set of all fixed-length relations;*
- *$\delta_M \colon Q_M \times (\Sigma \times \Sigma) \to Q_M$ is given by $\delta_M(R, [a, b]) = R^{[a,b]}$ for every $q \in Q_M$ and $a, b \in \Sigma$;*
- *$F_M = \{(\varepsilon, \varepsilon)\}$.*

As in the language case, each fixed-length word relation R determines a deterministic transducer $T_R = (Q_R, \Sigma \times \Sigma, \delta_R, q_{0R}, F_R)$ as follows: Q_R is the set of states of the master transducer reachable from the state R; q_{0R} is the state R; δ_R is the projection of δ_M onto Q_R; and $F_R = F_M$. T_R is the *minimal* deterministic transducer recognizing R:

Proposition 6.13 *For every fixed-length word relation R, the transducer T_R is the minimal deterministic transducer recognizing R.*

Like minimal DFA, minimal deterministic transducers are represented as tables of nodes. However, a remark is in order: since a state of a deterministic transducer has $|\Sigma|^2$ successors, one for each letter of $\Sigma \times \Sigma$, a row of the table has $|\Sigma|^2$ entries, too large when the table is only sparsely filled. Sparse transducers over $\Sigma \times \Sigma$ are better encoded as NFAs over Σ by introducing auxiliary states: a transition $q \xrightarrow{[a,b]} q'$ of the transducer is "simulated" by two transitions $q \xrightarrow{a} r \xrightarrow{b} q'$, where r is an auxiliary state with exactly one input and one output transition.

Fixed-length join. We give a recursive definition of $R_1 \circ R_2$. Let $(a,b)\cdot R$ denote the set $\{(aw_1, bw_2) \mid (w_1,w_2) \in R\}$. We have the following identities:

- $\emptyset \circ R = R \circ \emptyset = \emptyset$;
- $\{(\varepsilon,\varepsilon)\} \circ \{(\varepsilon,\varepsilon)\} = \{(\varepsilon,\varepsilon)\}$;
- $R_1 \circ R_2 = \bigcup_{a,b,c\in\Sigma} [a,b] \cdot \left(R_1^{[a,c]} \circ R_2^{[c,b]}\right)$.

Exploiting the identities we arrive at the algorithm of Figure 6.6, where *union* is defined similarly to *inter*.

Input: transducer table T, states q_1, q_2 of T
Output: state recognizing $L(q_1) \circ L(q_2)$

1 $join[T](q_1,q_2)$
2 **if** $G(q_1,q_2)$ is not empty **then return** $G(q_1,q_2)$
3 **if** $q_1 = q_\emptyset$ **or** $q_2 = q_\emptyset$ **then return** $q_\emptyset$
4 **else if** $q_1 = q_\epsilon$ **and** $q_2 = q_\epsilon$ **then return** q_ϵ
5 **else** $/* \, q_\emptyset \neq q_1 \neq q_\epsilon, q_\emptyset \neq q_2 \neq q_\epsilon \, */$
6 **for all** $(a_i, a_j) \in \Sigma \times \Sigma$ **do**
7 $q_{a_i,a_j} \leftarrow union[T]\left(join\left(q_1^{[a_i,a_1]}, q_2^{[a_1,a_j]}\right), \ldots, join\left(q_1^{[a_i,a_m]}, q_2^{[a_m,a_j]}\right)\right)$
8 $G(q_1,q_2) = make(q_{a_1,a_1}, \ldots, q_{a_1,a_m}, \ldots, q_{a_m,a_m})$
9 **return** $G(q_1,q_2)$

Figure 6.6: Algorithm $join[T](q_1,q_2)$

The complexity is $\mathcal{O}(n_1 \cdot n_2)$ for transducers of size n_1, n_2, since that is the maximal possible number of calls to *join*.

Fixed-length *pre*. Recall that in the fixed-length case we do not need any padding symbol. Then, given a fixed-length language L, $pre(L)$ admits an inductive definition that we now derive. Define

$$emb(L) = \{[v_1,v_2] \in (\Sigma\times\Sigma)^n \mid v_2 \in L\}$$

and define $pre_S(L)$, where $S \in (\Sigma\times\Sigma)^*$ and $L \in \Sigma^*$, as follows:

$$pre_S(L) = \{w_1 \in \Sigma^n \mid \exists [v_1,v_2] \in S : v_1 = w_1 \text{ and } v_2 \in L\}$$

Proposition 6.14 *The sets $pre_S(L)$ have the following properties:*

(1) if $S = \emptyset$ or $L = \emptyset$, then $pre_S(L) = \emptyset$;

(2) if $S \neq \emptyset \neq L$ *then* $pre_S(L) = \bigcup_{a,b\in\Sigma} a \cdot pre_{S^{[a,b]}}(L^b)$,
where $S^{[a,b]} = \{w \in (\Sigma \times \Sigma)^* \mid [a,b]w \in S\}$.

Proof: (1) is obvious. For (2), observe first that $pre_S(L) = \bigcup_{a\in\Sigma} a \cdot (pre_S(L))^a$, and so it suffices to show $(pre_S(L))^a = \bigcup_{b\in\Sigma} pre_{S^{[a,b]}}(L^b)$. We have:

$$\begin{aligned}
(pre_S(L))^a &= (proj_1(S \cap emb(L)))^a \\
&= \left(proj_1\left(\bigcup_{b\in\Sigma} [a,b] \cdot (S \cap emb(L))^{[a,b]}\right)\right)^a \\
&= \left(\bigcup_{b\in\Sigma} proj_1\left([a,b] \cdot (S \cap emb(L))^{[a,b]}\right)\right)^a \\
&= \left(\bigcup_{b\in\Sigma} a \cdot proj_1\left((S \cap emb(L))^{[a,b]}\right)\right)^a \\
&= \bigcup_{b\in\Sigma} proj_1\left((S \cap emb(L))^{[a,b]}\right) \\
&= \bigcup_{b\in\Sigma} proj_1\left(S^{[a,b]} \cap emb\left(L^b\right)\right) \\
&= \bigcup_{b\in\Sigma} pre_{S^{[a,b]}}(L^b)
\end{aligned}$$

The second step in this chain is the crucial one. It holds because # does not appear in the words of S, and so the word of S encoding a pair of the form $[aw_1, w_2] \in \Sigma^n \times \Sigma^n$ necessarily starts with a letter of the form $[a,b]$ for some $b \in \Sigma$. □

Proposition 6.14 leads to the recursive algorithm of Figure 6.5, which accepts as inputs a state of the transducer table recognizing a relation S, a state of the automaton table recognizing a language L, and returns the state of the automaton table recognizing $pre_S(L)$. The transducer table is not changed by the algorithm.

6.6 Decision Diagrams

Binary Decision Diagrams, BDDs for short, are a very popular data structure for the representation and manipulation of boolean functions. In this section we show that they can be seen as minimal automata of a certain kind.

Given a boolean function $f(x_1, \det \&minldots, x_n) : \{0,1\}^n \to \{0,1\}$, let L_f denote the set of strings $b_1 b_2 \dots b_n \in \{0,1\}^n$ such that $f(b_1, \dots b_n) = 1$. The minimal DFA recognizing L_f is very

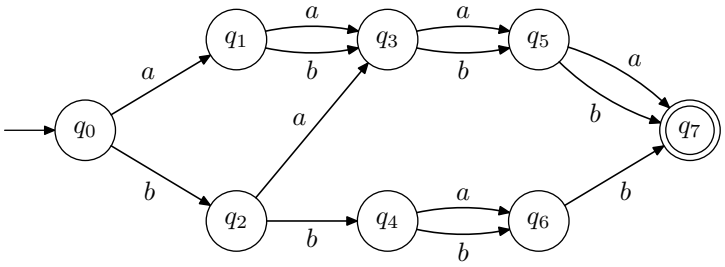
Input: transducer table TT, table T, state r of TT, state q of T
Output: state of T recognizing $pre_{L(r)}(L(q))$

```
1  pre[TT, T](r, q)
2     if G(r, q) is not empty then return G(r, q)
3     if r = r_∅ or q = q_∅ then return q_∅
4     else if r = r_ε and q = q_ε then return q_ε
5     else
6        for all a_i ∈ Σ do
7           q_{a_i} ← union(pre[TT, T](q^{[a_i, a_1]}, r^{a_1}), ..., pre[TT, T](q^{[a_i, a_m]}, r^{a_m}))
8        G(r, q) ← make(q_{a_1}, ..., q_{a_m});
9        return G(r, q)
```

Table 6.7: The algorithm $pre[TT, T](r, q)$.

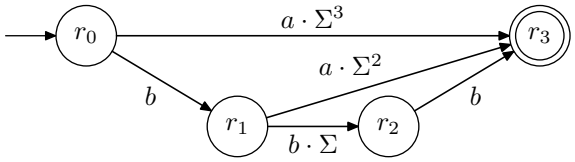
similar to the BDD representing f, but not completely equal. We modify the constructions of the last section to obtain an exact match.

Consider the following minimal DFA for a language of length four:

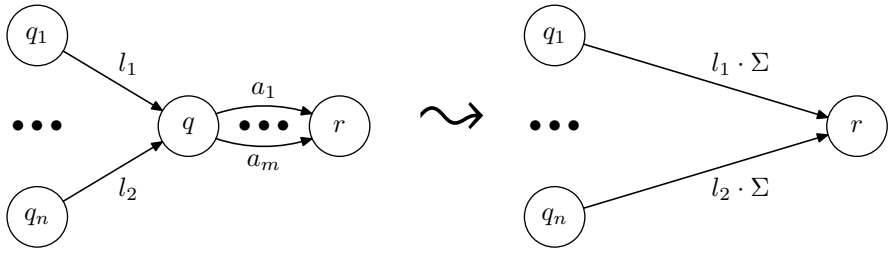


Its language can be described as follows: after reading an a, accept any word of length three; after reading ba, accept any word of length 2; after reading bb, accept any two-letter word whose last letter is a b.

Following this description, the language can also be more compactly described by an automaton with regular expressions as transitions:



Sections 6.6.1 and 6.6.2 show that this *z-automaton* (as we call it, see the definition below) is unique, and can be obtained by repeatedly applying to the minimal DFA the following reduction rule:



The converse direction also works: the minimal DFA can be recovered from the z-automaton by "reversing" the rule. This already allows to use z-automata as a data structure for fixed-length languages, but only through conversion to minimal DFAs: to compute an operation using z-automata, expand them to minimal DFAs, conduct the operation, and convert the result back. Sections 6.6.3 and 6.6.4show how to do better by directly defining the operations on z-automata, bypassing the minimal DFAs.

6.6.1 Z-automata and Kernels

A *zip* over an alphabet Σ is a regular expression of the form $a\Sigma^n = a\underbrace{\Sigma\Sigma\Sigma\ldots\Sigma\Sigma}_{n}$ (which looks a bit like a zip). The set of all zips over Σ is denoted by $Z(\Sigma)$. We introduce z-automata as automata whose transitions are labeled by zips.

Definition 6.15 *A* z-automaton *is a tuple $A = (Q, \Sigma, \delta, q_0, F)$, where Q, Σ, q_0, and F are as for NFAs, and $\delta\colon Q \times Z(\Sigma) \to Q$. The accepting runs of a z-automaton are defined as for NFAs-regs.*

A z-automaton is finite *if Q is finite, and* deterministic *if for every $a \in \Sigma$ there is a unique $k \in \mathbb{N}$ such that $\delta(q, a\Sigma^k) \neq \emptyset$.*

We abbreviate deterministic finite z-automaton to zDFA. We use zDFAs to recognize *kernels* of fixed-length languages:

Definition 6.16 *Let $L \subseteq \Sigma^n$ be a nonempty, fixed-length language, and let k be the largest number such that $L = \Sigma^k \cdot K$ for some $K \subseteq \Sigma^{(n-k)}$. The language K is the* kernel *of L, denoted $K = \ker(L)$. If $\ker L = L$, then L is a* kernel.

For convenience we also declare $\emptyset$ to be a kernel with $ker(\emptyset) = \emptyset$ and $k = 0$. A fixed-length language is completely determined by its length and its kernel. Since applications manipulate fixed-length languages of a given length known *a priori*, a z-automaton recognizing the kernel of a language can be used to represent the language itself.

The language recognized by the minimal DFA at the beginning of the chapter is an example of a kernel.

6.6.2 The Master Z-automaton

We define a master z-automaton "containing" the minimal zDFAs for all kernels.

Definition 6.17 *The* master z-automaton *over* Σ *is the tuple* $MZ = (Q_M, \Sigma, \delta_M, F_M)$*, where*

- Q_M *is the set of all kernels;*
- $(K, a\Sigma^k, K') \in \delta$ *iff* $K' = \ker(K^a)$ *and* $K^a = \Sigma^k \cdot K'$*; and*
- F_M *is the singleton set containing the language* $\{\epsilon\}$ *as only element.*

Example 6.18 Figure 6.7 shows the fragment of the master z-automaton. corresponding to the fragment of the master automaton in Figure 6.1. The languages $\{a, b\}$, $\{aa, ab, ba, bb\}$, and $\{ab, bb\}$, which appeared in Figure 6.1, are not kernels, and so no longer appear here. □

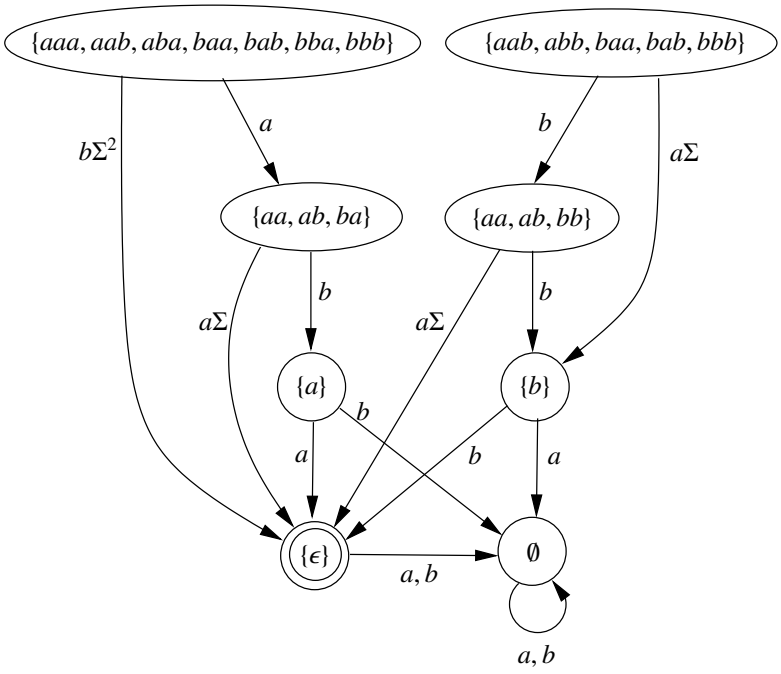


Figure 6.7: A fragment of the master z-automaton

Given a kernel K we define the zDFA $A_K = (Q_K, \Sigma, \delta_K, q_{0K}, F_K)$ as follows: Q_K is the set of states of the master z-automaton reachable from the state K; $q_{0K} = K$; δ_K is the projection of δ_M onto Q_K; and $F_K = F_M$. Using arguments similar to those for the master automaton, it is easy to see that A_K recognizes K.

A zDFA for a language is *minimal* if no other zDFA for the same language has fewer states. The following proposition shows that minimal zDFAs have very similar properties to minimal DFAs:

Proposition 6.19 *(1) A zDFA A is minimal if and only if (a) every state of A recognizes a kernel, and (b) distinct states of A recognize distinct kernels.*

(2) A_K is the unique minimal zDFA recognizing a kernel K.

(3) The result of exhaustively applying the reduction rule to the minimal DFA recognizing a language L is the minimal zDFA recognizing ker(L).

Proof: ($1 \Rightarrow$): For (a), assume that the language L_q recognized from a state q of A is not a kernel. Then A has a transition $(q, a \cdot \Sigma^{k_a}, q_a)$ for every $a \in \Sigma$, and moreover $(L(q))^a = (L(q))^b$ for every $a, b \in \Sigma$. Since $(L(q))^a = \Sigma^{k_a} \cdot L(q_a)$ for every $a \in \Sigma$, we have $\Sigma^{k_a} \cdot L(q_a) = \Sigma^{k_b} \cdot L(q_b)$ for every $a, b \in \Sigma$. Let m be a letter of Σ such that k_m is minimal. Then, there is a number k such that $L(q_a) = \Sigma^k \cdot L(q_m)$ for every $a \in \Sigma$, and so $L(q) = \Sigma^{k+1} \cdot L(q_m)$. Consider now the zDFA A' obtained from A by applying the transformation rule at the beginning of the section. A and A' recognize the same language, and so A is not minimal.

For (b), observe that the quotienting operation can be defined as for DFAs; if two distinct states recognize the same kernel then the quotient with respect to the language partition has fewer states than A, and so A is not minimal.

($1 \Leftarrow$): Let K be the language recognized by A. We prove that any zDFA A' recognizing K and satisfying (a) and (b) is isomorphic to A. Wdet&mine proceed by induction on the length n of the words of K. The case $n = 0$ is easy. Assume $n > 0$, and let q_0, q'_0 be the initial states of A and A'. Assume that A has a transition $(q_0, a \cdot \Sigma^k, q)$. By (a), $L(q)$ is a kernel, and so $k = ind(K^a)$ and $L(q) = ker(K^a)$ (no other kernel can be reached by a transition with a label of the form $a \cdot \Sigma^k$). By symmetry, A' also has a transition $(q'_0, a \cdot \Sigma^k, q')$, and $L(q) = ker(K^a)$. By induction hypothesis, the automata A_q and $A_{q'}$ obtained from A and A' by removing all states not reachable from q and q' are isomorphic, and we are done.

(2) A_K recognizes K and it satisfies conditions (a) and (b) of part (1) by definition, and so it is a minimal zDFA. Uniqueness follows immediately from the proof of ($1 \Leftarrow$).

(3) Let A be the minimal DFA recognizing K. Then distinct states of A recognize distinct languages. We show that after exhaustively applying the reduction rule, every state of the resulting zDFA recognizes a kernel, which is then the minimal zDFA by (1).

Assume that after exhaustively applying the reduction rule some state q does not recognize a kernel. Without loss of generality, we can assume that $L(q)$ is a language of minimal length. It follows that the target state of all the outgoing transitions of q is the same state q' recognizing the kernel of $L(q)$. But then the reduction rule can be applied to eliminate q, contradicting the hypothesis. □

6.6.3 A Data Structure

We use multi-zDFAs to represent sets of fixed-length languages of the same length. Multi-zDFAs are zDFAs with multiple initial states. The first idea would be to represent a set $\mathcal{L} = \{L_1, \ldots, L_m\}$

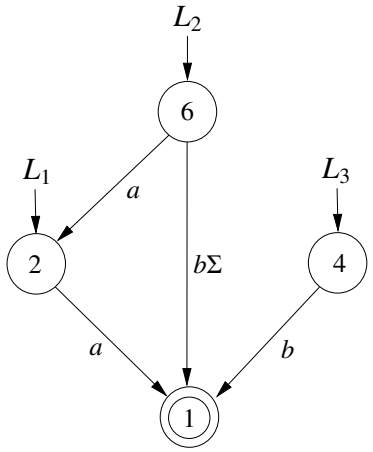


Figure 6.8: The multi-zDFA for $\{L_1, L_2, L_3\}$ with $L_1 = \{aa\}$, $L_2 = \{aa, bb\}$, and $L_3 = \{aa, ab\}$.

of languages by the multi-zDFA whose initial states are the states of the master z-automaton recognizing the kernels $\mathcal{K} = \{\ker(L_1), \ldots, \ker(L_m)\}$. However, this is incorrect: since for every language L and every number $n \geq 0$ we have $\ker(L) = \ker(\Sigma^n L)$, all the sets $\{\Sigma^n L_1, \ldots, \Sigma^n L_m\}$ are mapped to $\mathcal{K}$. This problem is solved by representing $\mathcal{L}$ by the multi-zDFA *and* the common length of $L_1, \ldots, L_m$, which together completely determine $\mathcal{L}$.

Example 6.20 Figure 6.8 shows the multi-zDFA for the set $\{L_1, L_2, L_3\}$ of Example 6.7. Recall that $L_1 = \{aa, ba\}$, $L_2 = \{aa, ba, bb\}$, and $L_3 = \{ab, bb\}$. This multi-zDFA is the result of applying the reduction rule to the multi-DFA of Figure 6.2. We represent the set by this multi-zDFA and the number 2, the length of L_1, L_2, L_3.

Observe that, while L_1, L_2 and L_3 have the same length, their kernels have not. Notice also how the state for L_1 is a descendant of the state for L_2. □

Multi-zDFAs are represented as a table of *kernodes*. A kernode is a triple $\langle q, l, s\rangle$, where q is a *state identifier*, l is a *length*, and $s = (q_1, \ldots, q_m)$ is the *successor tuple* of the kernode. Each kernode of the multi-zDFA corresponds to a state of the master z-automaton. The table for the multi-DFA of Figure 6.8 is:

Ident.	Length	a-succ	b-succ
1	0	0	0
2	1	1	0
4	1	0	1
6	2	2	1

This example explains the role of the new *length* field. If we only now that the a- and b-successors of, say, state 6, are the states 2 and 1, respectively, we still do not know which are the labels of the

transitions leading from 6 to 2 and from 6 to 1: they could be a and $b\Sigma$, or $a\Sigma$ and $b\Sigma^2$, or $a\Sigma^n$ and $b\Sigma^{n+1}$ for any $n \geq 0$. However, once we know that state 6 accepts a language of length 2, we can deduce the correct labels: since states 2 and 1 accept languages of length 1 and 0, respectively, the labels must be a and $b\Sigma$.

The procedure `kmake[T](l, s)`. The algorithms use a procedure `kmake[T](l, s)` with the following specification: `kmake[T](l, s)` returns the state recognizing $\ker\left(\bigcup_{i=1}^{m} a_i\Sigma^{l_i} \cdot K_i\right)$, where $K_1, \ldots, K_m$ are the kernels recognized by the states of the tuple s, and, for every $1 \leq i \leq m$, l_i is the number such that $a_i\Sigma^{l_i} \cdot K_i$ has length l.

If at least two of $K_1, \ldots, K_m$ are different, then `kmake[T](l, s)` can behave like `make[T](s)`: if the table T already contains a kernode $\langle q, l, s\rangle$, then `kmake[T](l, s)` returns its state identifier q; if no such kernode exists, then `kmake[T](l, s)` creates a new kernode $\langle q, l, s\rangle$ with a fresh identifier q, and returns q. It is easy to see that the result is the state

If $K_1 = \cdots = K_m \neq \emptyset$, then `kmake[T](l, s)` does not behave like `make[T](s)`. In this case we have

$$ker\left(\bigcup_{i=1}^{m} a_i\Sigma^{l_i} \cdot K_i\right) = ker(\Sigma \cdot K_1) = K_1$$

and so `kmake[T](l, t)` returns the state recognizing K_1. For instance, if T is the table above, then `kmake[T](3, (2, 2))` returns 3, while `make[T](2, 2)` creates a new node, say 7, having 2 as a-successor and b-successor. This is the feature that allows to obtain a more compact representation: in this situation, `kmake[T](l, t)` "saves" a state.

6.6.4 Operations on Kernels

The algorithms for operations of kernels are simple modifications of the algorithms of the previous section. We show how to modify the intersection algorithm, and the algorithm for simultaneous determinization and minimization.

Intersection.

Let L_1, L_2 be languages of length n, and let q_1, q_2 be the states of the master z-automaton recognizing the kernels $\ker(L_1)$ and $\ker(L_2)$. We derive an algorithm `kinter`$[T](q_1, q_2)$ of Table 6.8 returns the state recognizing $\ker(L_1 \cap L_2)$.

We need an auxiliary operation. Given two kernels K_1, K_2 of lengths l_1 and l_2, we define

$$K_1 \sqcap K_2 = \begin{cases} \ker(\Sigma^{l1-l2}K_1 \cap K_2) & \text{if } l1 \geq l2 \\ \ker(K_1 \cap \Sigma^{l2-l1}K_2) & \text{if } l1 < l2 \end{cases}$$

Intuitively, the operation first creates two languages of length $\max l_1, l_2$, intersects them, and returns the kernel of the intersection. The interest of the operation lies in this lemma:

Lemma 6.21 *For any two languages L_1, L_2 of the same length* $\ker(L_1 \cap L_2) = \ker(L_1) \sqcap \ker(L_2)$.

Proof: Let $L_1 = \Sigma^{n_1} K_1$ and $L_2 = \Sigma^{n_2} K_2$. Assume w.l.o.g. $n_1 \geq n_2$. Then we have $L_1 \cap L_2 = \Sigma^{n_1 - n_2}(\Sigma^{n_2} K_1 \cap K_2)$. So $\ker(L_1 \cap L_2) = \ker(\Sigma^{n_1 - n_2}(\Sigma^{n_2} K_1 \cap K_2)) = \ker(\Sigma^{n_2} K_1 \cap K_2) = K_1 \sqcap K_2$. □

By the lemma, computing the state recognizing $\ker(L_1 \cap L_2)$ amounts to computing the state recognizing $\ker(L_1) \sqcap \ker(L_2)$. Let $K_1 = \ker(L_1)$ and $K_2 = \ker(L_2)$. If K_1 or K_2 are empty, then the task is easy:

(1) if $K_1 = \emptyset$, then $K_1 \sqcap K_2 = \emptyset$;

(2) if $K_2 = \emptyset$, then $K_1 \sqcap K_2 = \emptyset$.

If $K_1 \neq \emptyset \neq K_2$, then we compute the state for $K_1 \sqcap K_2$ recursively, by computing the successor states for each letter $a \in \Sigma$, and then applying `kmake`. Recall that in the master automaton, the a-successor of the state for L is the state for L^a. In the master z-automaton, the a-successor of the state for a kernel K is not the state for K^a, which may not even be a kernel, but the state for $\ker(K^a)$. So for every letter $a \in \Sigma$, we need to compute the state for $\ker((K_1 \sqcap K_2)^a)$. For this, we prove the following equation, where l_1, l_2 are the lengths of K_1, K_2:

(3) if $K_1 \neq \emptyset \neq K_2$, then

$$\ker((K_1 \sqcap K_2)^a) = \begin{cases} K_1 \sqcap \ker(K_2^a) & \text{if } l_1 > l_2 \\ \ker(K_1^a) \sqcap K_2 & \text{if } l_1 < l_2 \\ \ker(K_1^a) \sqcap \ker(K_2^a) & \text{if } l_1 = l_2 \end{cases}$$

To show that (3) holds, assume first $l_1 > l_2$. We have

$$\begin{array}{rcll} \ker((K_1 \sqcap K_2)^a) & = & \ker((\Sigma^{l1-l2} K_1 \cap K_2)^a) & \text{(def. of } \sqcap) \\ & = & \ker(\Sigma^{l1-l2-1} K_1 \cap K_2^a) & ((L \cap M)^a = L^a \cap M^a) \\ & = & \ker(\Sigma^{l1-l2-1} K_1) \sqcap \ker(K_2^a)) & \text{(Lemma 6.21)} \\ & = & K_1 \sqcap \ker(K_2^a) & (K_1 \text{ is a kernel}) \end{array}$$

The case $l_1 > l_2$ is symmetric. If $l_1 = l_2$, let Let $K_1^a = \Sigma^{l'_1} \ker(K_1^a)$, $K_2^a = \Sigma^{l'_2} \ker(K_2^a)$, and let $l' = \min\{l'_1, l'_2\}$

$$\begin{array}{rcll} \ker((K_1 \sqcap K_2)^a) & = & \ker((K_1 \cap K_2)^a) & \text{(def. of } \sqcap) \\ & = & \ker(K_1^a \cap K_2^a) & \\ & = & \ker(K_1^a) \sqcap \ker(K_2^a) & \text{(Lemma 6.21)} \end{array}$$

Equations (1)-(3) lead to the algorithm *kinter*$[T](q_1, q_2)$ shown in Table 6.8.

Example 6.22 Example 6.8 showed a run of *inter*() on the two languages represented by the multi-DFA at the top of Figure 6.4. The multi-zDFA for the same languages is shown at the top of Figure 6.9, and the rest of the figure describes the run of *kinter*() on it. Recall that pink nodes correspond to calls whose result has already been memoized, and need not be executed. The meaning of the green nodes is explained below. □

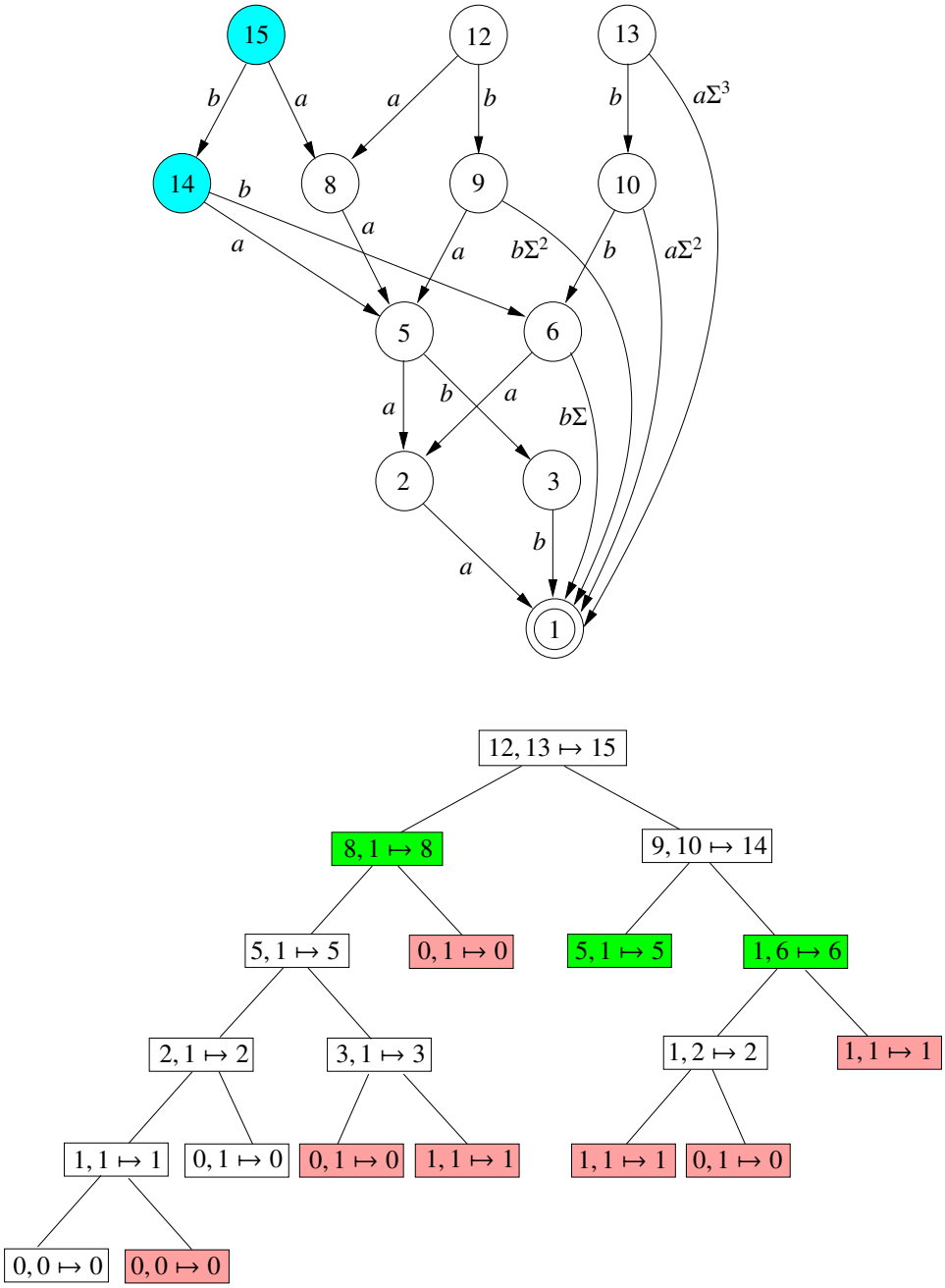


Figure 6.9: An execution of *kinter*().

kinter$[T](q_1, q_2)$
Input: table T, states q_1, q_2 of T for languages L_1, L_2
Output: state recognizing $\ker(L_1 \cap L_1)$

```
1  if G(q1,q2) is not empty  then return G(q1,q2)
2  if q1 = q∅ ∨ q2 = q∅ then return q∅
3  if q1 ≠ q∅ ∧ q2 ≠ q∅ then
4     if l1 < l2 then
5        for all i = 1,...,m do ri ← inter[T](q1, q2^ai)
6        G(q1,q2) ← kmake[T](l2, r1,...,rm)
7     else if l1   l2 then
8        for all i = 1,...,m do ri ← inter[T](q1^ai, q2)
9        G(q1,q2) ← kmake[T](l1, r1,...,rm)
10    else /* l1 = l2 */
11       for all i = 1,...,m do ri ← inter[T](q1^ai, q2^ai)
12       G(q1,q2) ← kmake[T](l1, r1,...,rm)
13 return G(q1,q2)
```

Table 6.8: *kinter*$[T](q_1, q_2)$

The efficiency of the algorithm can be improved by observing that two further equations hold:

(6') if $ker(L_1) = \{\varepsilon\}$ then $(ker(L_1 \cap L_2))^a = (ker(L_2))^a$;

(7') if $ker(L_2) = \{\varepsilon\}$ then $(ker(L_1 \cap L_2))^a = (ker(L_1))^a$;

These equations show that $kinter[T](q_\varepsilon, q) = q = kinter[T](q, q_\varepsilon)$ for every state q. So we can improve *kinter*() by explicitly checking if one of the arguments is q_ε. The green nodes in Figure 6.9 correspond to calls whose result is immediately returned with the help of this check. Observe how this improvement has a substantial effect, reducing the number of calls from 19 to only 5.

Determinization and Minimization.

The algorithm *kdet&min*() that converts an NFA recognizing a fixed-language L into the minimal zDFA recognizing $\ker(L)$ differs from *det&min*() essentially in one letter: it uses *kmake*() instead of *make*(). It is shown in Table 6.9.

Example 6.23 Figure 6.10 shows again the NFA of Figure 6.5, and the minimal zDFA for the kernel of its language. The run of *kdet&min*(A) is shown at the bottom of the figure. For the difference with *det&min*(A), consider the call *kstate*$(\{\delta, \epsilon, \zeta\})$. Since the two recursive calls *kstate*$(\{\eta\})$ and *kstate*$(\{\eta, \theta\})$ return both state 1 with length 1, *kmake*$(1, 1)$ does not create a new state, as *make*$(1, 1)$ would do it returns state 1. The same occurs at the top call *kstate*$(\{\alpha\})$. □

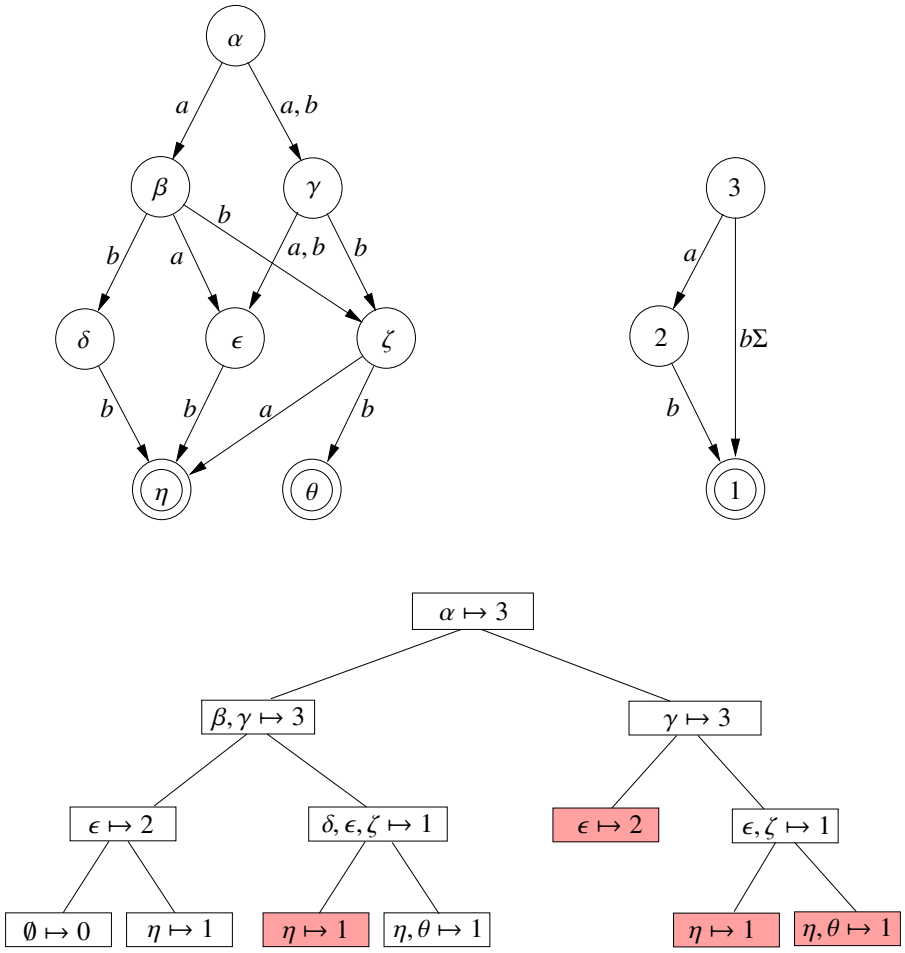

α
a
a, b
β
γ
b
b
a
a, b
b
δ
ϵ
ζ
b
b
a
b
η
θ
3
a
2
bΣ
b
1
α ↦ 3
β, γ ↦ 3
γ ↦ 3
ϵ ↦ 2
δ, ϵ, ζ ↦ 1
ϵ ↦ 2
ϵ, ζ ↦ 1
∅ ↦ 0
η ↦ 1
η ↦ 1
η, θ ↦ 1
η ↦ 1
η, θ ↦ 1

Figure 6.10:

kstate$[A](S, l)$
Input: NFA $A = (Q, \Sigma, \delta, q_0, F)$, set $S \subseteq Q$ of length l
Output: state of the multi-zDFA recognizing $L(R)$
1 **if** $G(S)$ is not empty **then return** $G(S)$
2 **else if** $S = \emptyset$ **then return** $q_\emptyset$
3 **else if** $S \cap F \neq \emptyset$ **then return** q_ϵ
4 **else** $/* S \neq \emptyset$ and $S \cap F = \emptyset * /$
5 **for all** $i = 1, \ldots, m$ **do** $S_i \leftarrow \delta(S, a_i)$
6 $G(S) \leftarrow kmake(l, state[A](S_1), \ldots, state[A](S_m))$;
7 **return** $G(S)$

kdet&*min*(A)
Input: NFA $A = (Q, \Sigma, \delta, q_0, F)$
Output: state of a multi-DFA recognizing $L(A)$
1 **return** $state[A](\{q_0\})$

Table 6.9: The algorithm *kdet*&*min*(A).

Exercises

Exercise 58 Prove that the minimal DFAs for the languages of length 3 have at most 8 states.

Exercise 59 Give an *efficient* algorithm that receives as input the minimal DFA of a fixed-length language and returns the number of words it contains.

Exercise 60 Let $\Sigma = \{0, 1\}$. Given $a, b \in \Sigma$, let $a \cdot b$ be the usual multiplication (an analog of boolean *and*) and let $a \oplus b$ be 0 if $a = b = 0$ and 1 otherwise (an analog of boolean *or*).

Consider the boolean function $f\colon \Sigma^6 \to \Sigma$ defined by

$$f(x_1, x_2, x_3, x_4, x_5, x_6) = (x_1 \cdot x_2) \oplus (x_3 \cdot x_4) \oplus (x_5 \cdot x_6)$$

1. Construct the minimal DFA recognizing $\{x_1x_2x_3x_4x_5x_6 \mid f(x_1, x_2, x_3, x_4, x_5, x_6) = 1\}$.
(For instance, the DFA accepts 111000 because $f(1, 1, 1, 0, 0, 0) = 1$, but not 101010, because $f(1, 0, 1, 0, 1, 0) = 0$.)

2. Construct the minimal DFA recognizing $\{x_1x_3x_5x_2x_4x_6 \mid f(x_1, x_2, x_3, x_4, x_5, x_6) = 1\}$.
(Notice the different order! Now the DFA accepts neither 111000, because $f(1, 0, 1, 0, 1, 0) = 0$, nor 101010, because $f(1, 0, 0, 1, 1, 0) = 0$.)

3. More generally, consider the function

$$f(x_1, \ldots, x_{2n}) = \bigoplus_{1 \leq k \leq n} (x_{2k-1} \cdot x_{2k})$$

and the languages $\{x_1x_2\dots x_{2n-1}x_{2n} \mid f(x_1,\dots,x_{2n}) = 1\}$ and $\{x_1x_3\dots x_{2n-1}x_2x_4\dots x_{2n} \mid f(x_1,\dots,x_{2n}) = 1\}$.
Show that the size of the minimal DFA grows linearly in n for the first language, and exponentially in n for the second language.

Exercise 61 Given a language L of length n and a permutation π of $\{1,\dots,n\}$, we define $\pi(L) = \{a_{\pi(1)}\dots a_{\pi(n)} \mid a_1a_2\dots a_n \in L\}$. Prove that the following problem is NP-complete:

Given: A minimal DFA A recognizing a fixed-length language, a number $k \geq 1$.
Decide: Is there a permutation π of $\{1,\dots,n\}$, where n is the length of $L(A)$, such that the minimal DFA for $L(\pi(A))$ has at most k states?

Exercise 62 Given $X \subset \{0,1,\dots,2^k-1\}$, let A_X be the minimal DFA recognizing the *lsbf* encodings of length k of the elements of X.

- Define $X+1$ by $X+1 = \{x+1 \mod 2^k \mid x \in X\}$. Give an algorithm that on input A_X produces A_{X+1} as output.
- Let $A_X = (Q,\{0,1\},\delta,q_0,F)$. Which is the set of numbers recognized by the automaton $(Q,\{0,1\},\delta',q_0,F)$, where $\delta'(q,b) = \delta(q,1-b)$?

Exercise 63 Recall the definition of DFAs with negative transitions (DFA-nt's) introduced in Exercise 25, and consider the alphabet $\{0,1\}$. Show that if only transitions labelled by 1 can be negative, then every regular language over $\{0,1\}$ has a *unique* minimal DFA-nt.

Chapter 7

Applications I: Pattern matching

As a first example of a practical application of automata, we consider the *pattern matching* problem. Given $w, w' \in \Sigma^*$, we say that w' *is a factor of* w if there are words $w_1, w_2 \in \Sigma^*$ such that $w = w_1 w' w_2$. If w_1 and $w_1 w'$ have lengths k and k', respectively, we say that w' is the $[k, k']$-*factor* of w. The *pattern matching problem* is defined as follows: Given a word $t \in \Sigma^+$ (called the *text*), and a regular expression p over Σ (called the *pattern*), determine the smallest $k \geq 0$ such that some $[k', k]$-factor of t belongs to $L(p)$. We call k the *first occurrence of* p *in* t.

Example 7.1 Let $p = a(ab^*a)b$. Since $ab, aabab \in L(p)$, the $[1, 3]$-, $[3, 5]$-, and $[0, 5]$-factors of *aabab* belong to $L(p)$. So the first occurrence of p in *aabab* is 3. □

Usually one is interested not only in finding the ending position k of the $[k', k]$-factor, but also in the starting position k'. Adapting the algorithms to also provide this information is left as an exercise.

7.1 The general case

We present two different solutions to the pattern matching problem, using nondeterministic and deterministic automata, respectively.

Solution 1. Clearly, some word of $L(p)$ occurs in t if and only if some prefix of t belongs to $L(\Sigma^* p)$. So we construct an NFA A for the regular expression $\Sigma^* p$ using the rules of Figure 2.4 (i.e., the algorithm *RegtoNFA*), and then simulate A on t as in *MemNFA*$[A](q_0,t)$ on page 56. Recall that the simulation algorithm reads the text letter by letter, maintaining the set of states reachable from the initial state by the prefix read so far. So the simulation reaches a set of states S containing a final state if and only if the prefix read so far belongs to $L(\Sigma^* p)$. Here is the pseudocode for this algorithm:

PatternMatchingNFA(t, p)
Input: text $t = a_1 \dots a_n \in \Sigma^+$, pattern $p \in \Sigma^*$
Output: the first occurrence k of p in t, or $\perp$ if no such occurrence exists.

```
1  A ← RegtoNFA(Σ*p)
2  S ← {q_0}
3  for all k = 0 to n − 1 do
4      if S ∩ F ≠ ∅ then return k
5      S ← δ(S, a_i)
6  return ⊥
```

If we assume that the alphabet Σ has fixed size, then the complexity of *PatternMatchingNFA* for a word of length n and a pattern of length m can be estimated as follows. *RegtoNFA*$(\Sigma^* p)$ takes $\mathcal{O}(m)$ time[1]. The loop is executed at most n times, and, for an automaton with k states, each line of the loop's body takes at most $\mathcal{O}(k^2)$ time. Since *RegtoNFA*(p) takes $\mathcal{O}(m)$ time, we have $k \in \mathcal{O}(m)$, and so the loop runs in $\mathcal{O}(nm^2)$ time. The overall runtime is thus $\mathcal{O}(m + nm^2) = \mathcal{O}(nm^2)$.

Solution 2. We proceed as in the previous case, but instead of constructing a NFA for the regular expression $\Sigma^* p$, we construct a DFA instead:

PatternMatchingDFA(t, p)
Input: text $t = a_1 \dots a_n \in \Sigma^+$, pattern p
Output: the first occurrence k of p in t, or $\perp$ if no such occurrence exists.

```
1  A ← NFAtoDFA(RegtoNFA(Σ*p))
2  q ← q_0
3  for all i = 0 to n − 1 do
4      if q ∈ F then return k
5      q ← δ(q, a_i)
6  return ⊥
```

Notice that there is trade-off: while the conversion to a DFA can take (much) longer than the conversion to a NFA, the membership check for a DFA is faster. The complexity of *PatternMatchingDFA* for a word of length n and a pattern of length m can be easily estimated: **RegtoNFA**(p) takes $\mathcal{O}(m)$ time, and so the call to *NFAtoDFA* (see Table 2.3.1) takes $2^{\mathcal{O}(m)}$ time and space. Since the loop is executed at most n times, and each line of the body takes constant time, the overall runtime is $\mathcal{O}(n) + 2^{\mathcal{O}(m)}$.

[1] If Σ does not have fixed size, then constructing the NFA for $\Sigma^* p$ takes $\mathcal{O}(m + |\Sigma|)$ time.

7.2 The word case

We study the special but very common special case of the pattern-matching problem in which we wish to know if a given word appears in a text. In this case the pattern p is the word itself. For the rest of the section we consider an arbitrary but fixed text $t = a_1 \dots a_n$ and an arbitrary but fixed word pattern $p = b_1 \dots b_m$.

It is easy to find a faster algorithm for this special case, without any use of automata theory: just move a "window" of length m over the text t, one letter at a time, and check after each move whether the content of the window is p. The number of moves is $n - m + 1$, and a check requires $\mathcal{O}(m)$ letter comparisons, giving a runtime of $\mathcal{O}(nm)$. In the rest of the section we present an even faster algorithm with time complexity $\mathcal{O}(m + n)$. Notice that in some applications both n and m can be very large, and the difference between $\mathcal{O}(nm)$ and $\mathcal{O}(m + n)$ very significant.

We start by examining Solution 2, and in particular the time required to construct the minimal DFA for $\Sigma^* p$. It is easy to see that the number of states of the minimal DFA must be at least as large as the number of prefixes of p. (For a proof, let p_1, p_2 be two arbitrary but distinct prefixes of p, and let $p_1 s_1 = p = p_2 s_2$. Then $p_1 s_1$ is accepted by the DFA, but $p_2 s_1$ is not. So the residuals of $\Sigma^* p$ with respect to p_1 and p_2 differ. Since the number of states of the minimal DFA is equal to the number of distinct residuals, we are done.) We now construct a DFA in which the states are the prefixes of the pattern p. Since this DFA has a minimal number of states, and the minimal DFA is unique, it follows that this DFA is the minimal DFA. Before giving the formal definition of the construction, we consider an example. Figure 7.1 shows the minimal DFA for $p = nano$, with states ε, n, na, nan, and $nano$. Intuitively, the DFA keeps track of *how close it is* to finding $nano$. For instance:

- if the automaton is in state n and it reads an a, it moves to state na;
- if the automaton is in state na and it reads an a, it moves to state ε;
- if the automaton is in state nan and it reads an a, it moves to state na. This is the crucial case: since the next letter is a instead of o, the DFA has not found $nano$ yet. However, it does not move to state ε. That would be a mistake, because if the next two letters are n and o, then the DFA should accept! So the DFA moves to na.

The general rule is:

> If the DFA is in state $v \in \Sigma^*$ and it reads a letter α, it moves to the largest suffix of $v\alpha$ that is also a prefix of p.

For a reason that will become clear later, we call this automaton the *eager DFA* for p. Let us define it formally.

Definition 7.2 *We denote by $ol(w)$ the longest suffix of w that is a prefix of p. In other words, $ol(w)$ is the unique longest word of the set*

$$\{u \in \Sigma^* \mid \exists v, v' \in \Sigma^*. w = vu \wedge p = uv'\}$$

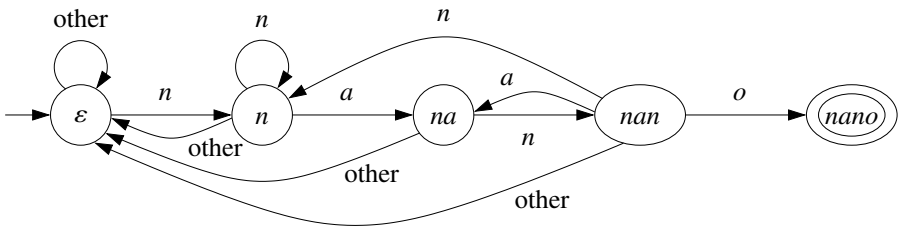


Figure 7.1: DFA for $p = nano$: *eagerDFA*(*nano*)

For example, if $p = nano$, then $overl(nan\mathbf{a}) = na$ and $overl(nan\mathbf{n}) = n$.

Definition 7.3 *The* eager DFA of the pattern p *is the tuple* $\mathbf{eagerDFA}(p) = (Q_e, \Sigma, \delta_e, q_{0e}, F_e)$, *where :*

- Q_e *is the set of prefixes of* p *(including* ε*);*
- *for every* $u \in Q_e$, *for every* $\alpha \in \Sigma$: $\delta_e(u, \alpha) = ol(u\alpha)$;
- $q_{0e} = \varepsilon$; *and*
- $F_e = \{p\}$

We can now obtain an algorithm for the pattern-matching problem in the word case by replacing line 1 in *Pattern-Matching-DFA*(t, p) with

$$A \leftarrow eagerDFA(p)$$

The algorithm stops in state p if and only if the pattern p has been read.

In order to estimate the runtime of this algorithm, observe that *eagerDFA*(p) has $m + 1$ states and $m|\Sigma|$ transitions, where Σ is the size of the alphabet. At this point it makes sense to consider two different scenarios:

- The alphabet Σ is fixed and known in advance. Then we can consider $|\Sigma|$ as a constant, and "hide" it in the $\mathcal{O}$-symbol. We then get a DFA of size $\mathcal{O}(m)$.
- The alphabet Σ is not known in advance, it is implicitely defined as the set of letters that appear in the text and the pattern. If we assume that the text is at least as long as the pattern, then the alphabet has size $\mathcal{O}(n)$, and we get a DFA of size $\mathcal{O}(nm)$.

In the second scenario, since constructing a DFA of size $\mathcal{O}(nm)$ requires $\Omega(m^2)$ time, no algorithm based on the explicit, direct construction of the eager DFA for p can lead to a $\mathcal{O}(n + m)$ algorithm. Moreover, constructing *eagerDFA*(p) requires to examine the text in order to determine the letters that appear in it.

A first idea to reduce the memory requirements is to store the DFA more compactly. Observe that for every state u and for every letter $\alpha \in \Sigma$, if α does not appear in p, then $\delta_e(u, \alpha) = \varepsilon$. So the transitions for letters appearing in the text but not in the pattern can be "summarized" into one single transition. After this optimization each state has at most $m + 1$ outgoing transitions, and *eagerDFA*(p) has size $\mathcal{O}(m^2)$. However, this still leads to a $\mathcal{O}(n + m^2)$ algorithm, and it does not solve the problem of having to inspect the text to construct *eagerDFA*(p).

These problems can be solved by introducing a new data structure for the language $L(\Sigma^* p)$: the *lazy DFA for* p. We show that the lazy DFA has size $\mathcal{O}(m)$ and can be constructed in $\mathcal{O}(m)$ time, even when Σ is not known in advance. The result will be the well-known Knuth-Morris-Pratt algorithm, presented from an automata-theoretic point of view.

7.2.1 Lazy DFAs

A DFA can be seen as the control unit of a machine that reads the input from a tape divided into cells by means of a reading head. Each cell contains a letter of the input and the tape extends infinitely long to the right of the input with all cells empty. Initially, the content of the tape is w, where $w \in \Sigma^*$ is the word to be processed by the automaton, and the head is reading the first cell of the tape. This cell contains the first letter of w if $w \neq \varepsilon$ and it is empty otherwise. At each step, the machine reads the content of the current cell, moves the head one cell to the right, and updates the current control state according to the transition function of the DFA. It accepts an input if the control state is a final state at the moment the head reaches an empty cell for the first time.

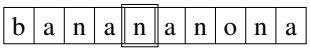


Figure 7.2: Tape with reading head.

Notice that the direction in which the head moves is always the same. We now consider machines in which the direction is also determined by the control unit. For our purposes we only need a very modest extension: the head may either move to the right or stay put. We call this model a *lazy DFA*. Formally, a lazy DFA only differs from an eager DFA in the transition function, which has the form $\delta\colon Q \times \Sigma \to Q \times \{R, N\}$, where R stands for *move Right* and N stands for *No move*. A transition of a lazy DFA is therefore a fourtuple (q, a, q', d), where $d \in \{R, N\}$ is a direction. Like an eager DFA, a lazy DFA accepts an input if if the control state is a final state at the moment the head reaches an empty cell for the first time. (Notice that a lazy DFA may stay put on the same cell forever, and in this case the machine does not accept the input. However, this will never be the case for our lazy DFAs.)

The lazy DFA for p again has the prefixes of p as states – with the same intuitive meaning of how close one is to p. However, it has *fewer transitions*. A lazy DFA at state u only distinguishes whether the current letter is a *hit* (the letter following u in the pattern) or a *miss* (any other letter). Figure 7.3 shows the lazy DFA for the pattern *nano*: for instance, at state *nan* the automaton only

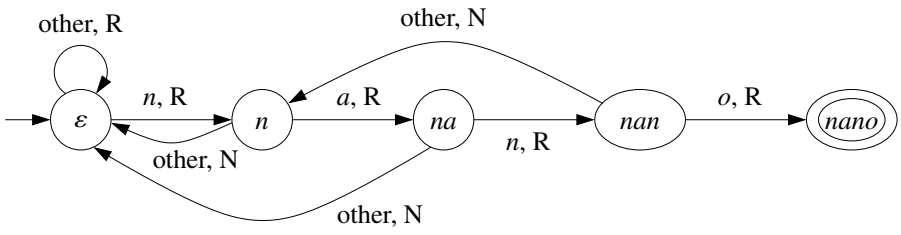


Figure 7.3: Lazy DFA for $p = nano$: $lazyDFA(nano)$

distinguishes between o and "other". In the case of a hit the lazy DFA "moves forward", just like the eager DFA. In the case of a miss, if the current state is ε, then the head moves right and control stays on ε[2]. But if the current state is not ε, then the head *does not move*, and the eager DFA moves to a new state *which depends only on the current state, not on the current letter*. So we can summarize all miss-transitions into one (all have the same destination), and the summarized description of the lazy DFA only has two transitions per state.

If the lazy DFA is at state $u \neq \varepsilon$, and it reads a miss, what should be the new state? The state is chosen to guarantee that the lazy DFA "simulates" the eager DFA: a step $u \xrightarrow{\alpha} v$ of the eager DFA is simulated by a sequence of moves

$$u \xrightarrow{(\alpha,N)} u_1 \xrightarrow{(\alpha,N)} v_2 \cdots u_k \xrightarrow{\alpha,R} v$$

of the lazy DFA. For instance, in our example the move $nan \xrightarrow{n} n$ of the eager DFA is simulated in the lazy DFA by the sequence

$$nan \xrightarrow{(n,N)} n \xrightarrow{(n,N)} \varepsilon \xrightarrow{(n,R)} n\ .$$

But how should we choose the new state to guarantee the simulation property? In the rest of the chapter we show that after a miss from state $u \neq \varepsilon$ we should move to the *largest proper suffix of u which is a prefix of p* (the longest "proper overlap" of u with p). We start by formally defining the lazy DFA for the language $\Sigma^* p$.

Definition 7.4 *Let w be a proper prefix of p.*

- *We denote by h_w the unique letter such that $w\,h_w$ is a prefix of p. We call h_w a* hit *(from state w). Notice that $h_\varepsilon = a_1$.*
- *For $w \neq \varepsilon$ we define $pol(w)$ (short for proper overlap) as the longest* proper *suffix of w that is a prefix of p, that is, $pol(w)$ is the unique longest word of the set*

$$\{u \in \Sigma^* \mid \textit{there exists } v \in \Sigma^+, v' \in \Sigma^* \textit{ such that } w = vu \textit{ and } p = uv'\}$$

[2] Actually, this is still the same behavior as the eager DFA.

Notice the difference in the definitions of *ol* and *pol*: $ol(w)$ is a longest suffix of w, while $pol(w)$ is a longest proper suffix, and so in particular always *strictly shorter* than w. For example, for $p = nano$ we have $overl(nano) = nano$, while $overl(nano) = \varepsilon$.

Definition 7.5 *The lazy DFA for p is the tuple* $\mathbf{lazyDFA}(p) = (Q_l, \Sigma, \delta_l, q_{0l}, F_l)$*, where:*

- Q_l *is the set of prefixes of p;*
- *for every* $u \in Q_l, \alpha \in \Sigma$*:*

$$\delta_l(u,\alpha) = \begin{cases} (u\alpha, R) & \text{if } \alpha = h_u & \text{(hit)} \\ (\varepsilon, R) & \text{if } \alpha \neq h_u \text{ and } u = \varepsilon & \text{(miss from } \varepsilon\text{)} \\ (pol(u), N) & \text{if } \alpha \neq h_u \text{ and } u \neq \varepsilon & \text{(miss from other states)} \end{cases}$$

- $q_{0l} = \varepsilon$*; and*
- $F_l = \{p\}$

In order to prove the simulation property we introduce some notation:

Definition 7.6 *Let* $lazyDFA(p) = (Q_l, \Sigma, \delta_l, q_{0l}, F_l)$*, let* $u \in Q_l$*, and let* $\alpha \in \Sigma$*. We denote by* $\widehat{\delta_l}(u,\alpha)$ *the unique state v such that*

$$u = u_0 \xrightarrow{(\alpha,N)} u_1 \xrightarrow{(\alpha,N)} u_2 \cdots u_k \xrightarrow{(\alpha,R)} v$$

for some $u_1, \ldots, u_k \in Q_l$*,* $k \geq 0$*.*

Notice that v always exists. For a proof, observe that, by definition of $\mathbf{lazyDFA}(p)$, if $u_i \xrightarrow{(\alpha,N)} u_{i+1}$ then u_{i+1} is a proper prefix of u_i, and so the lazy DFA cannot perform an arbitrarily large of steps in which the head does not move.

Proposition 7.7 *Let* $lazyDFA(p) = (Q_l, \Sigma, \delta_l, q_{0l}, F_l)$ *and* $eagerDFA(p) = (Q_e, \Sigma, \delta_e, q_{0e}, F_e)$*. Then* $\widehat{\delta_l}(v,\alpha) = \delta_e(v,\alpha)$ *for every prefix v of p and every* $\alpha \in \Sigma$*.*

Proof: If α is a hit, i.e., if $\alpha = h_v$, then we have $\delta_e(v,\alpha) = v\alpha = \widehat{\delta}(v,\alpha)$. If α is a miss, we proceed by induction on $|v|$. If $|v| = 0$, then $v = \varepsilon$ and by the definitions of δ_e and δ_l we have $\delta_e(v,\alpha) = \delta_l(v,\alpha) = \widehat{\delta_l}(v,\alpha)$. If $|v| > 0$, then by the definition of δ_l we have $\delta_l(v,\alpha) = (pol(v), N)$, and so:

$$\begin{array}{lll} & \widehat{\delta_l}(v,\alpha) & \\ = & \widehat{\delta_l}(pol(v),\alpha) & (\delta_l(v,\alpha) = (pol(v),N) \text{ and definition of } \widehat{\delta_l}) \\ = & \delta_e(pol(v),\alpha) & (\ |pol(v)| < |v| \quad \text{and induction hypothesis}) \end{array}$$

To complete the proof we show $\delta_e(pol(v),\alpha) = \delta_e(v,\alpha)$. By the definition of δ_e we have $\delta_e(pol(v),\alpha) = ol(pol(v)\alpha)$ and $\delta_e(v,\alpha) = ol(v\alpha)$. So we have to prove $ol(pol(v)\alpha) = ol(v\alpha)$ when α is a miss, i.e., when $\alpha \neq h_v$.

Since $pol(v)$ is a suffix of v by definition, every suffix of $pol(v)\alpha$ is a suffix of $v\alpha$. So, by the maximality of $ol(v\alpha)$, there is a (possibly empty) word u such that $ol(v\alpha) = u\, ol(pol(v)\,\alpha)$. We prove that $u = \varepsilon$. If $u \neq \varepsilon$, then $u\, overlap(v)$ is not a proper suffix of v (because $overlap(v)$ is the maximal proper suffix), and so, since $ol(v\,\alpha) = u\, overlap(v)\,\alpha$, we get that $ol(v\,\alpha)$ is not a proper suffix of $v\,\alpha$ either. Since $ol(v\,\alpha)$ is a suffix of $v\,\alpha$, we have $ol(v\,\alpha) = v\,\alpha$, and, since it is also a prefix of p, we get that $v\,\alpha$ is a prefix of p. It follows $\alpha = h_v$, contradicting the assumption that α is a miss. □

7.2.2 Constructing the lazy DFA in $\mathcal{O}(m)$ time

The lazy DFA has size $\mathcal{O}(m)$, but in order to construct it we must compute the function $pol(v)$ for every proper prefix v of p. This can be easily done in $\mathcal{O}(m^2)$ time, but we show that it can even be done in $\mathcal{O}(m)$ time.

Recall that $pol(w)$ is the longest proper suffix of w that is a prefix of p (Definition 7.4). The following equation holds for every proper prefix v of p:

$$pol(v\,h_v) = \begin{cases} \varepsilon & \text{if } v = \varepsilon \\ pol(v)\,h_v & \text{if } v \neq \varepsilon \text{ and } h_{pol(v)} = h_v \\ pol(pol(v)\,h_v) & \text{if } v \neq \varepsilon \text{ and } h_{pol(v)} \neq h_v \end{cases} \tag{7.1}$$

Only the third equation needs an argument. Since both $pol(v\,h_v)$ and $pol(pol(v)\,h_v)$ are prefixes of p, it suffices to show that they have the same length. Clearly, every proper suffix of $pol(v)\,h_v$ is a proper suffix of $v\,h_v$, which, by maximality of pol, implies $|pol(pol(v)\,h_v)| \leq |pol(v\,h_v)|$. Now we prove $|pol(pol(v)\,h_v)| \geq |pol(v\,h_v)|$. If $|pol(v\,h_v)| = 0$, we are done. If $|pol(v\,h_v)| > 0$, then $pol(v\,h_v) = w\,h_v$, where w is a proper suffix of v and a prefix of p. By maximality of pol, we get that w is also a suffix of $pol(v)$, and so that $w\,h_v$ is a suffix of $pol(v)\,h_v$ and a prefix of p. But $pol(v)\,h_v$ is not a prefix of p, because $h_{pol}(v) \neq h_v$, and so $w\,h_v$ is a proper suffix of $pol(v)\,h_v$, which implies $pol(v)\,h_v \neq w\,h_v$. So $w\,h_v$ is a *proper* suffix of $pol(v)\,h_v$ and a prefix of p, which, by the maximality of pol, implies $|pol(pol(v)\,h_v)| \geq |w\,h_v| = |pol(v\,h_v)|$.

Equation 7.1 yields the recursive algorithm on the left of Table 7.1. If we let $p = p[1]\ldots p[m]$ and identify the prefix of p of length k with the number k, we can rewrite the algorithm as shown on the right side of the table.

Table 7.2 shows an iterative algorithm for pol obtained by applying dynamic programming to the recursive algorithm on the right of Table 7.1. The array $pol[]$ memoizes the results of the function calls of the recursive versions. In other words, $pol[i] = pol(p[1]\ldots p[i])$. Notice that at line 6 we have $u < j - 1$ because of line 4, which implies $u + 1 < j$. Therefore, when executing the assignment we have already computed $pol[u + 1]$.

To estimate the complexity of our final algorithm we make the observation that lines 1, 2 are executed once, and that lines 4, 5, and 6 are executed $m - 2$ times. So the algorithm runs in $\mathcal{O}(m)$ time.

Pol(*w*)
Input: a prefix w of p.
Output: $pol(w)$

```
1  if |w| ≤ 1 then return ε
2  if w = vα and v ≠ ε then
3      u ← pol(v)
4      if α = h_u then return uα
5      else return pol(uα)
```

Pol(k)
Input: a number $0 \leq k \leq m$.
Output: the length of $pol(p[1] \dots p[k])$.

```
1  if k ≤ 1 then return 0
2  if k ≥ 2 then
3      u ← pol(k − 1)
4      if p[k] = p[u + 1] then return u + 1
5      else return pol(u + 1)
```

Table 7.1: Two recursive algorithms for computing *pol*

PolIt(m)
Input: a number $1 \leq m$.
Output: the array $pol[1..m]$ with
$pol[i]$ = length of $pol(p[1] \dots p[i])$ for every $1 \leq i \leq m$.

```
1  pol[1] ← 0
2  for all j = 2 to m do
3      u ← pol[j − 1]
4      if p[j] = p[u + 1] then pol[j] = u + 1
5      else pol[j] ← pol[u + 1]
```

Table 7.2: An iterative algorithm for computing *pol*

Exercises

Exercise 64 Design an algorithm that solves the following problem for an alphabet Σ. Discuss the complexity of your solution.

- *Given*: $w \in \Sigma^*$ and a regular expression r over Σ.
- *Find*: A shortest prefix $w_1 \in \Sigma^*$ of w such that there exists a prefix $w_1 w_2$ of w and $w_2 \in \mathcal{L}(r)$.

Exercise 65 Construct the eager and lazy DFAs for the patterns *mammamia* and *abracadabra*.

Exercise 66 We have shown that lazy DFAs are more concise than eager DFAs for languages of the form $\Sigma^* p$. However, this improvement does not entirely come for free. There is a space vs. running-time trade-off, because, due to the steps in which the head does not move, a lazy DFA may need more than n steps to read a word of length n. Find a word w and a pattern p such that the run of **eagerDFA**(p) on w takes at most n steps and the run of **lazyDFA**(p) takes at least $2n - 1$ steps. *Hint:* a simple pattern of the form a^k for some $k \geq 0$ is sufficient.

Exercise 67 Two-way finite automata are an extension of lazy automata in which the reading head may not only move right or stay put, but also move left. The tape extends infinitely long to both the left and to the right of the input with all cells empty. A word is accepted if the control state is a final state at the moment the head reaches an empty cell to the right of the input for the first time.

- Give a two-way DFA with $O(n)$ states for the language $(0 + 1)^*1(0 + 1)^n$.
- Give algorithms for membership and emptiness of two-way automata.
- (Difficult!) Prove that the languages recognized by two-way DFA are regular.

Chapter 8

Applications II: Verification

One of the main applications of automata theory is the automatic verification or falsification of correctness properties of hardware or software systems. Given a system (like a hardware circuit, a program, or a communication protocol), and a property (like"after termination the values of the variables x and y are equal" or "every sent message is eventually received"), we wish to *automatically* determine whether the system satisfies the property or not.

8.1 The Automata-Theoretic Approach to Verification

We consider discrete systems for which a notion of *configuration* can be defined[1]. The system is always at a certain configuration, with instantaneous moves from one configuration to the next determined by the system dynamics. If the semantics allows a move from a configuration c to another one c', then we say that c' is a *legal successor* of c. A configuration may have several successors, in which case the system is nondeterministic. There is a distinguished set of *initial* configurations. An *execution* is a sequence of configurations (finite or infinite) starting at some initial configuration, and in which every other configuration is a legal successor of its predecessor in the sequence. A *full* execution is either an infinite execution, or an execution whose last configuration has no successors.

In this chapter we are only interested in finite executions. The set of executions can then be seen as a language $E \subseteq C^*$, where the alphabet C is the set of possible configurations of the system. We call C^* the *potential executions* of the system.

Example 8.1 As an example of a system, consider the following program with two boolean variables x, y:

[1]We speak of the configurations of a system, and not of its states, in order to avoid confusion with the states of automata.

```
1  while x = 1 do
2    if y = 1 then
3      x ← 0
4    y ← 1 − x
5  end
```

A configuration of the program is a triple $[\ell, n_x, n_y]$, where $\ell \in \{1,2,3,4,5\}$ is the current value of the program counter, and $n_x, n_y \in \{0,1\}$ are the current values of x and y. So the set C of configurations contains in this case $5 \times 2 \times 2 = 20$ elements. The initial configurations are $[1,0,0], [1,0,1], [1,1,0], [1,1,1]$, i.e., all configurations in which control is at line 1. The sequence

$$[1,1,1]\ [2,1,1]\ [3,1,1]\ [4,0,1]\ [1,0,1]\ [5,0,1]$$

is a full execution, while

$$[1,1,0]\ [2,1,0]\ [4,1,0]\ [1,1,0]$$

is also an execution, but not a full one. In fact, all the words of

$$(\ [1,1,0]\ [2,1,0]\ [4,1,0]\ )^*$$

are executions, and so the language E of all executions is infinite. □

Assume we wish to determine whether the system has an execution satisfying some property of interest. If both the language $E \subseteq C^*$ of executions and the language $P \subseteq C^*$ of potential executions that satisfy the property are regular, and we can construct automata recognizing them, then we can solve the problem by checking whether the language $E \cap P$ is empty, which can be decided using the algorithms of Chapter 4. This is the main insight behind the automata-theoretic approach to verification.

The requirement that the language E of executions is regular is satisfied by all systems with finitely many reachable configurations (i.e., finitely many configurations c such that some execution leads from some initial configuration to c). A *system NFA* recognizing the executions of the system can be easily obtained from the *configuration graph*: the graph having the reachable configurations as nodes, and arcs from each configuration to its successors. The construction is very simple: The states of the system NFA are the reachable configurations of the program plus a new state, which is also the initial state. For every transition $c \to c'$ of the graph there is a transition $c \xrightarrow{c'} c'$ in the NFA. All states are final.

Example 8.2 Figure 8.1 shows the configuration graph of the program of Example 8.1, and below it its system NFA. Notice that the labels of the transitions of the NFA carry no information, because they are just the name of the target state.

We wish to automatically determine if the system has a full execution such that initially $y = 1$, finally $y = 0$, and y never increases. Let $[\ell, x, 0], [\ell, x, 1]$ stand for the sets of configurations where $y = 0$ and $y = 1$, respectively, but the values of ℓ and x are arbitrary. Similarly, let $[5, x, 0]$ stand

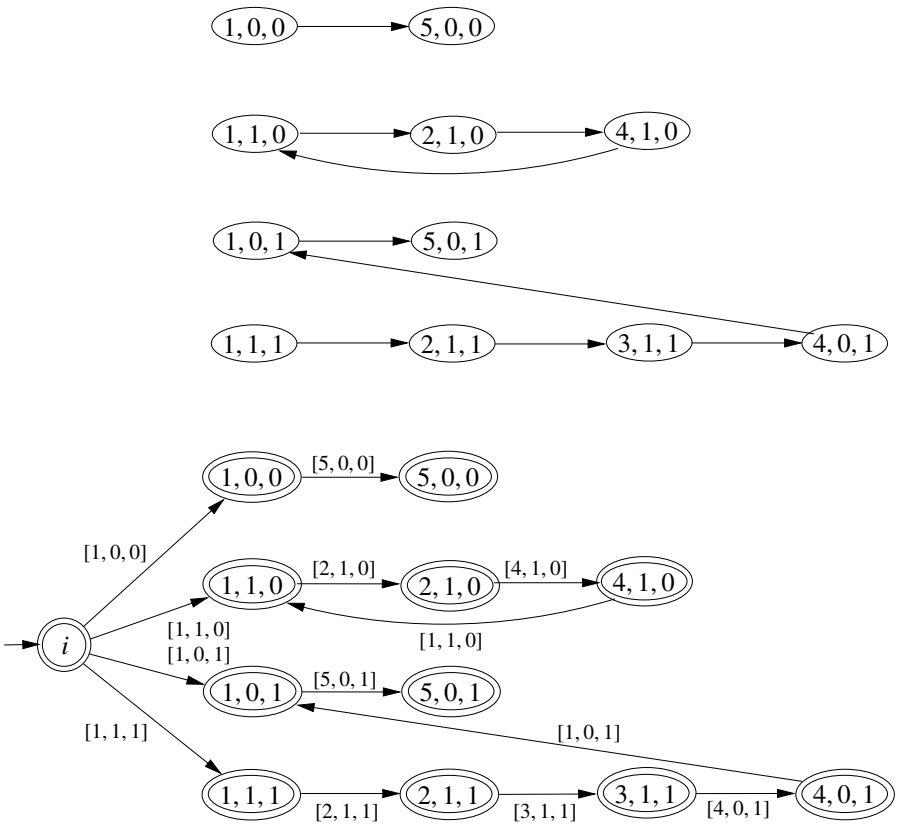


Figure 8.1: Configuration graph and system NFA of the program of Example 8.1

for the set of configurations with $\ell = 5$ and $y = 0$, but x arbitrary. The set of potential executions satisfying the property is given by the regular expression

$$[\ell, x, 1]\ [\ell, x, 1]^*\ [\ell, x, 0]^*\ [5, x, 0]$$

which is recognized by the *property NFA* at the top of Figure 8.2. Its intersection with the system NFA of Figure 8.1 is shown at the bottom of Figure 8.2. A light pink state of the pairing labeled by $[\ell, x, y]$ is the result of pairing the light pink state of the property NFA and the state $[\ell, x, y]$ of the system NFA. Since labels of the transitions of the pairing are always equal to the target state, they are omitted for the sake of readability.

Since no state of the intersection has a dark pink color, the intersection is empty, and so the program has no execution satisfying the property. □

Example 8.3 We wish now to automatically determine whether the assignment $y \leftarrow 1 - x$ in line 4 of the program of Example 8.1 is redundant and can be safely removed. This is the case if

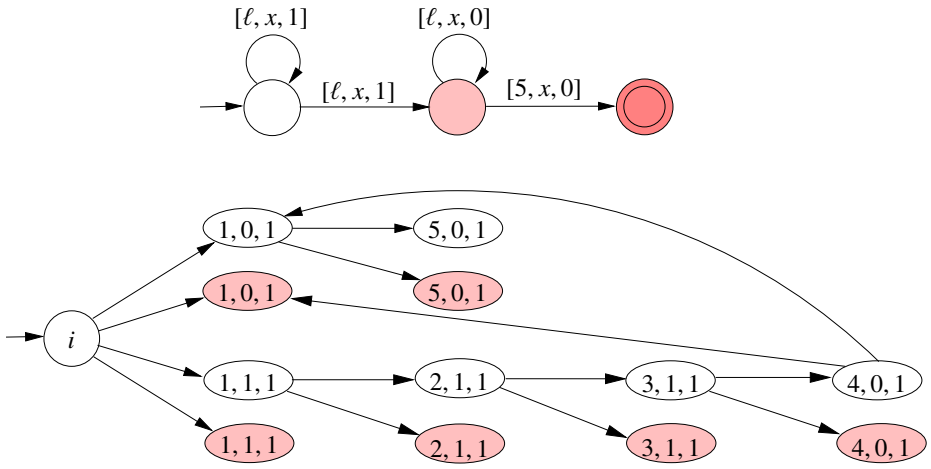


Figure 8.2: Property NFA and product NFA

the assignment never changes the value of y. The potential executions of the program in which the assignment changes the value of y at some point correspond to the regular expression

$$[\ell, x, y]^* \ ([4, x, 0]\, [1, x, 1] \ + \ [4, x, 1]\, [1, x, 0])\ [\ell, x, y]^* \ .$$

A property NFA for this expression can be easily constructed, and its intersection with the system NFA is again empty. So the property holds, and the assignment is indeed redundant. □

8.2 Networks of Automata.

Concurrent systems are particularly difficult to design correctly, which makes them a suitable target for automatic verification techniques. These systems usually consist of a number of communicating sequential components. In this section we exploit this structure to define configuration graphs in a more systematic way. We assign an NFA to each sequential component, yielding a *network of automata* and then define the graph by means of an automata theoretic construction.

A *network of automata* is a tuple $\mathcal{A} = \langle A_1, \ldots, A_n \rangle$ of NFAs with pairwise disjoint sets of states. Each NFA has its own alphabet Σ_i (the alphabets $\Sigma_1, \ldots, \Sigma_n$ are not necessarily pairwise disjoint). Alphabet letters are called *actions*. Given an action a, we say that the i-th NFA *participates in* a if $a \in \Sigma_i$.

A *configuration* of a network is a tuple $\langle q_1, \ldots, q_n \rangle$ of states, where $q_i \in Q_i$ for every $i \in \{1, \ldots, n\}$. The *initial configuration* is the configuration $\langle q_{01}, \ldots, q_{0n} \rangle$, where q_{0i} is the initial state of A_i. An action a is *enabled* at a configuration $\langle q_1, \ldots, q_n \rangle$ if for every $i \in \{1, \ldots, n\}$ *such that* A_i *participates in* a there is a transition $(q_i, a, q_i') \in \delta_i$. If a is enabled, then it can *occur*, and its occurrence makes *all participating NFAs* A_i move to the state q_i', while the non-participating

NFAs *do not change their state*. The configuration reached by the occurrence of a is a *successor* of $\langle q_1, \ldots, q_n \rangle$.

Example 8.4 The upper part of Figure 8.3 shows a network of three NFAs modeling a 3-bit counter.

We call the NFAs A_0, A_1, A_2 instead of A_1, A_2, A_3 to better reflect their meaning: A_i stands for the i-th bit. Each NFA but the last one has three states, two of which are marked with 0 and 1. The alphabets are

$$\Sigma_0 = \{inc, inc_1, 0, \ldots, 7\} \qquad \Sigma_1 = \{inc_1, inc_2, 0, \ldots, 7\} \qquad \Sigma_2 = \{inc_2, 0, \ldots, 7\}$$

Intuitively, the system interacts with its environment by means of the actions $inc, 0, 1, \ldots, 7$. More precisely, *inc* models a request of the environment to increase the counter by 1, and $i \in \{0, 1, \ldots, 7\}$ models a query of the environment asking if i is the current value of the counter. A configuration of the form $[b_2, b_1, b_0]$, where $b_2, b_1, b_0 \in \{0, 1\}$, indicates that the current value of the counter is $4b_2 + 2b_1 + b_0$ (configurations are represented as triples of states of A_2, A_1, A_0, in that order). □

Given a network of automata we define its *asynchronous product* as the output of algorithm *Async* in Table 8.1. Starting at the initial configuration, the algorithm repeatedly picks a configuration from the worklist and constructs its successors; adding nw successors to the worklist.

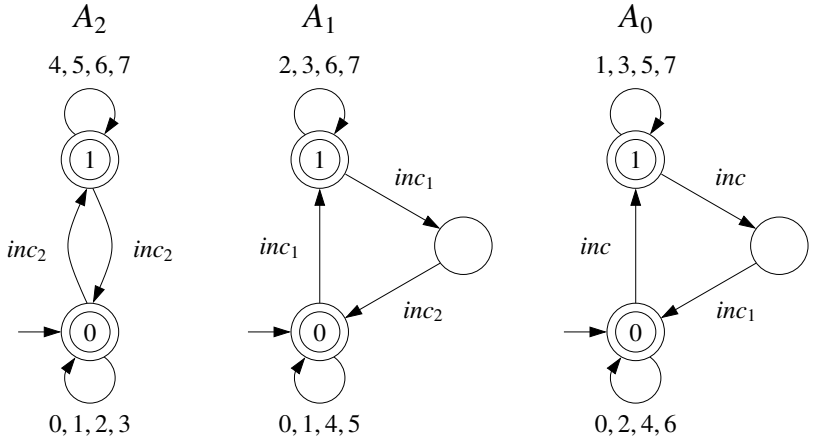
Example 8.5 The bottom part of Figure 8.3 shows the asynchronous product of the network modeling the 3-bit counter, shown at the top of the figure. (Actually, all states are final, but have been drawn as simple instead of double ellipses for simplicity.) Observe that at the configurations $[1, aux, 0]$ and $[0, aux, 0]$ the actions *inc* and inc_2 are *concurrent*: they are both enabled, and the sets of automata participating in them are disjoint. This means that they can occur independently of each other. □

This proposition, whose proof is an easy consequence of the definitions, characterizes the executions of the network in terms of the asynchronous product:

Proposition 8.6 *A sequence $c_0\, c_1 \ldots\, c_n$ of configurations is an execution of the network $A_1 \otimes \cdots \otimes A_n$ if and only if there is a word $a_1 \ldots a_n \in \Sigma^*$ such that $c_0 \xrightarrow{a_1} c_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} c_n$ is an accepting run of $A_1 \otimes \cdots \otimes A_n$.*

Notice that $A_1 \otimes \cdots \otimes A_n$ can be easily transformed into the system NFA accepting the executions of the network by means of a transformation similar to the one described in Figure 8.1.

The next example shows how to model a (possibly concurrent) program with finite-domain variables (like booleans or finite ranges) as a network of automata. The key idea is to have a network component for each process *and for each variable*. A process-component has a state for each control point of the process, and a variable component has one state for each possible value of the variable. Reading or writing of a variable by a process is modeled as a *joint action* with the process-component and the variable-component as participants.



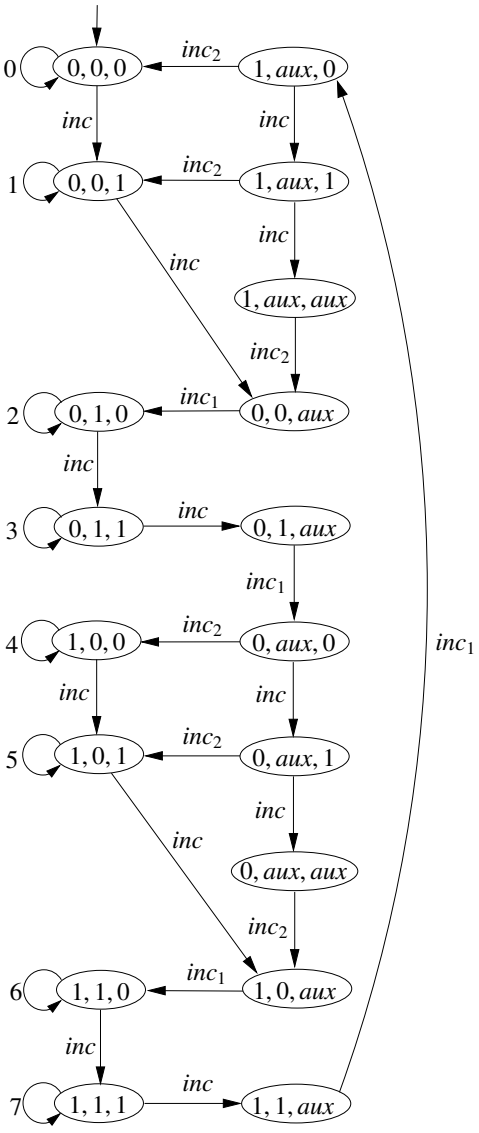


Figure 8.3: A network modeling a 3-bit counter and its asynchronous product.

AsyncProduct$(A_1, \ldots, A_n)$
Input: a network of automata $\mathcal{A} = A_1, \ldots A_n$, where
$A_1 = (Q_1, \Sigma_1, \delta_1, q_{01}, Q_1), \ldots, A_n = (Q_n, \Sigma_n, \delta_n, q_{0n}, Q_n)$
Output: the asynchronous product $A_1 \otimes \cdots \otimes A_n = (Q, \Sigma, \delta, q_0, F)$

```
1  Q, δ, F ← ∅
2  q_0 ← [q_01, ..., q_0n]
3  W ← {[q_01, ..., q_0n]}
4  while W ≠ ∅ do
5     pick [q_1, ..., q_n] from W
6     add [q_1, ..., q_n] to Q
7     add [q_1, ..., q_n] to F
8     for all a ∈ Σ_1 ∪ ... ∪ Σ_n do
9        for all i ∈ [1..n] do
10          if a ∈ Σ_i then Q'_i ← δ_i(q_i, a) else Q'_i = {q_i}
11       for all [q'_1, ..., q'_n] ∈ Q'_1 × ... × Q'_n do
12          if [q'_1, ..., q'_n] ∉ Q then add [q'_1, ..., q'_n] to W
13          add ([q_1, ..., q_n], a, [q'_1, ..., q'_n]) to δ
14 return (Q, Σ, δ, q_0, F)
```

Table 8.1: Asynchronous product of a tuple of automata.

Example 8.7 The network of Figure 8.4 models a version of Lamport and Burns' 1-bit mutual exclusion algorithm for two processes[2]. In the algorithm, process 0 and process 1 communicate through two shared boolean variables, b_0 and b_1, which initially have the value 0. Process i reads and writes variable b_i and reads variable $b_{(1-i)}$. The algorithm should guarantee that the processes 0 and 1 never are simultaneously in their *critical sections*. Other properties the algorithm should satisfy are discussed later.

The NFAs for the processes are shown in Figure 8.4. Initially, process 0 is in its non-critical section (local state nc_0); it can also be trying to enter its critical section (t_0), or be already in its critical section (c_0). It can move from nc_0 to t_0 at any time by setting b_0 to 1; it can move from t_0 to c_0 if the current value of b_1 is 0; finally, it can move from c_0 to nc_0 at any time by setting b_0 to 0.

Process 1 is a bit more complicated. While nc_1, t_1, and c_1 play the same rôle as in process 0, the local states q_1 and q'_1 model a "polite" behavior: Intuitively, if process 1 sees that process 0 is either trying to enter or in the critical section, it moves to an "after you" local state q_1, and then sets b_1 to 0 to signal that it is no longer trying to enter its critical section (local state q'_1). It can then return to the non-critical section if the value of b_0 is 0.

In principle, in the non-critical sections the processes can also do some unspecified work, which

[2]L. Lamport: The mutual exclusion problem: part II-statements and solutions. *JACM* 1986

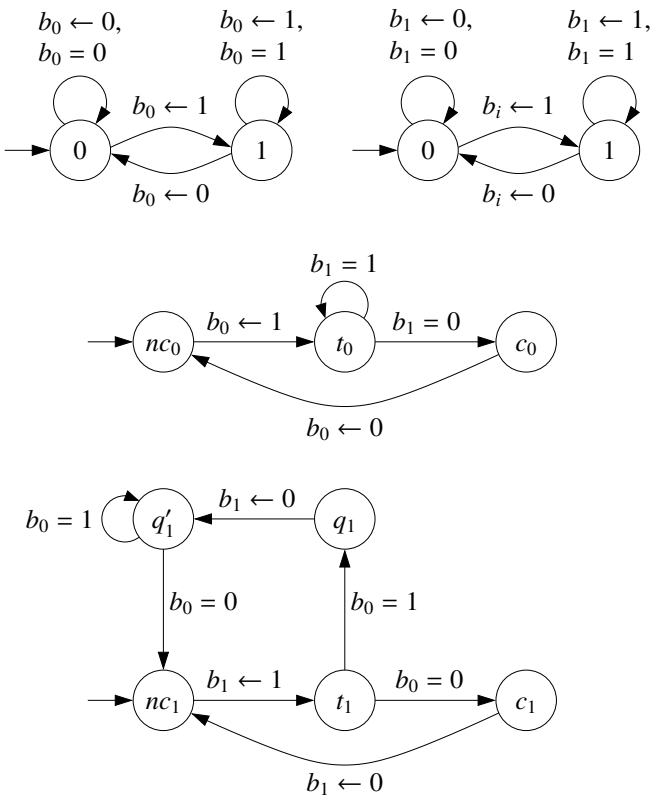


Figure 8.4: A network of four automata modeling a mutex algorithm.

should be represented by transition from $nc_0 and nc_1$ to themselves. Since for the purposes of this chapter these transition are not important, we have omitted them.

A configuration of this network is a four tuple. Transitions of the asynchronous product correspond to the execution of a read or a write statement by one of the processes. The asynchronous product is shown in Figure 8.5, where $\langle v_0, v_1, s_0, s_1 \rangle$ represents the configuration in which the NFA modelling processes 0 and 1 are in states s_0, s_1, and in which $b_0 = v_0$ and $b_1 = v_1$.

□

8.2.1 Checking Properties

We use Lamport's algorithm to present some more examples of properties and how to check them automatically.

The mutual exclusion property can be easily formalized: it holds if the asynchronous product does not contain any configuration of the form $[v_0, v_1, c_0, c_1]$, where $v_0, v_1 \in \{0, 1\}$. The property

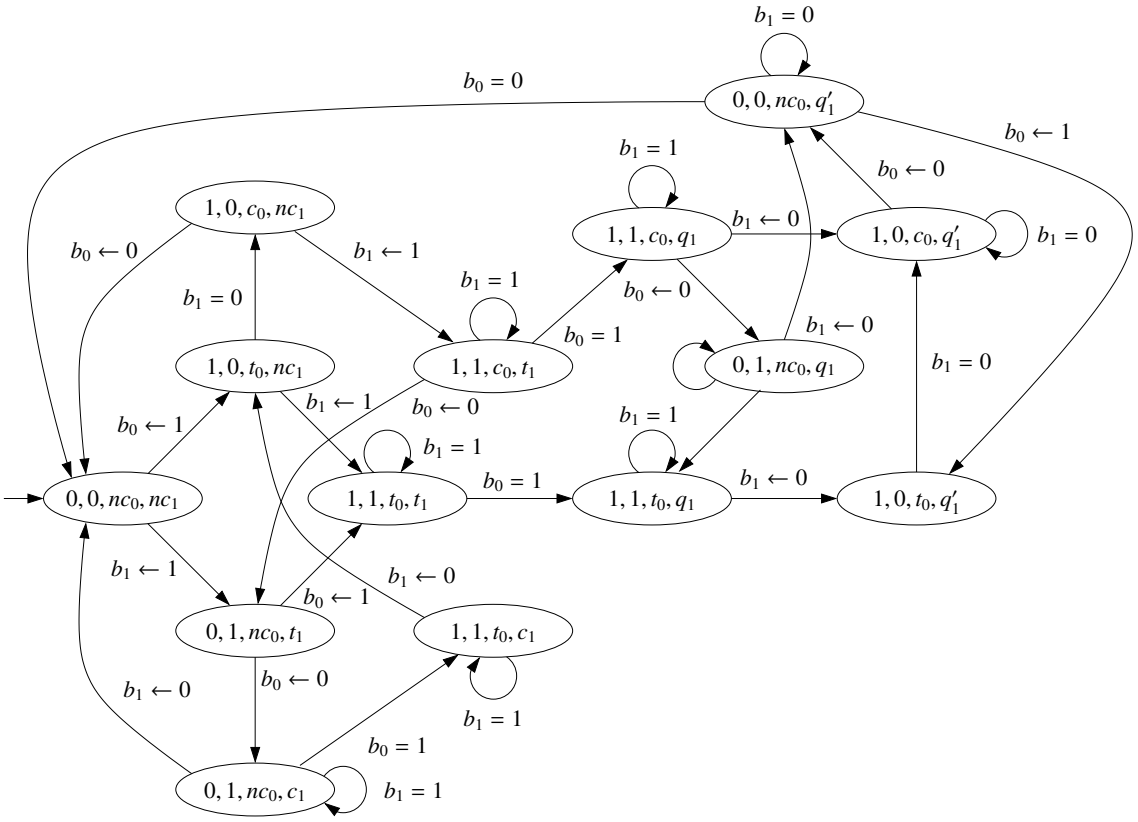


Figure 8.5: Asynchronous product of the network of Figure 8.4.

can be easily checked on-the-fly while constructing the asynchronous product, and a quick inspection of Figure 8.5 shows that it holds. Notice that in this case we do not need to construct the NFA for the executions of the program. This is always the case if we only wish to check the reachability of a configuration or set of configurations. Other properties of interest for the algorithm are:

- Deadlock freedom. The algorithm is deadlock-free if every configuration of the asynchronous product has at least one successor. Again, the property can be checked on the fly, and it holds for Lamport's algorithm.

- Bounded overtaking. After process 0 signals its interest in accessing the critical section (by moving to state t_0), process 1 can enter the critical section at most once before process 0 enters the critical section.
  This property can be checked using the NFA E recognizing the executions of the network, obtained as explained above by renaming the labels of the transitions of the asynchronous product. Let NC_i, T_i, C_i be the sets of configurations in which process i is in its non-critical

section, is trying to access its critical section, or is in its critical section, respectively. Let Σ stand for the set of all configurations. The regular expression

$$r = \Sigma^* \, T_0 \, (\Sigma \setminus C_0)^* \, C_1 \, (\Sigma \setminus C_0)^* \, NC_1 \, (\Sigma \setminus C_0)^* \, C_1 \, \Sigma^*$$

represents all the possible executions that violate the property. Now, if E is the NFA recognizing the language of executions, the property holds if and only if $L(E) \cap L(r) = \emptyset$, which can be checked using the algorithms of Chapter 4.

The straightforward method to check if $L(E) \cap L(r) = \emptyset$ has four steps: (1) construct the NFA E for the network $A_1, \ldots, A_n$ using *AsyncProduct*(); (2) transform r into an NFA V (V for "violations") using the algorithm of Section 2.4.1; (3) construct the NFA $E \cap V$ using *intersNFA*(); (4) check emptiness of $E \cap V$ using *empty*().

The key problem of this approach is that the number of states of E can be as high as the product of the number of states of the the components $A_1, \ldots, A_n$, which can easily exceed the available memory. This is called the *state-explosion* problem, and the literature contains a wealth of proposals to deal with it. We conclude the section with a first easy step towards palliating this problem. A more in depth discussion can be found in the next section.

Observe that the NFA E may have *more* states than $E \cap V$: if a state of E is not reachable by any word on which V has a run, then it does not appear in $E \cap V$. Since the transitions of V are often labeled with only a small fraction of the letters in the alphabet of E, the difference in size between E and $E \cap V$ can be considerable, and so it is better to *directly* construct $E \cap V$, bypassing the construction of E. This is easily achieved by observing that the intersection $A_1 \cap A_2$ of two NFAs A_1, A_2 corresponds to the particular case of the asynchronous product in which $\Sigma_1 \subseteq \Sigma_2$ (or vice versa): if $\Sigma_1 \subseteq \Sigma_2$, then A_2 participates in every action, and the NFAs $A_1 \otimes A_2$ and $A_1 \cap A_2$ coincide. More generally, if $\Sigma_1 \cup \cdots \cup \Sigma_n \subseteq \Sigma_{n+1}$, then $A_1 \otimes \cdots \otimes A_n \otimes A_{n+1} = (A_1 \otimes \cdots \otimes A_n) \cap A_{n+1}$. Since the alphabet of V is the union of the alphabets of $A_1, \ldots, A_n$, we can then check emptiness of $A_1 \otimes \cdots \otimes A_n \otimes V$ by means of the algorithm of Table 8.2.

8.3 The State-Explosion Problem

The automata-theoretic approach constructs an NFA V recognizing the potential executions of the system that violate the property one is interested in, and checks whether the automaton $E \cap V$ is empty, where E is an NFA recognizing the executions of the system. This is done by constructing the set of states of $E \cap V$, while simultaneously checking if any of them is final.

The number of states of E can be very high. If we model E as a network of automata, the number can be as high as the product of the number of states of all the components of the network. So the approach has exponential worst-case complexity. The following result shows that this cannot be avoided unless P=PSPACE.

Theorem 8.8 *The following problem is PSPACE-complete.*
Given: A network of automata $A_1, \ldots, A_n$ over alphabets $\Sigma_1, \ldots, \Sigma_n$, a NFA V over $\Sigma_1 \cup \ldots \cup \Sigma_n$.
Decide: if $L(A_1 \otimes \cdots \otimes A_n \otimes V) \neq \emptyset$.

CheckViol$(A_1, \ldots, A_n, V)$
Input: a network $\langle A_1, \ldots A_n \rangle$, where $A_i = (Q_i, \Sigma_i, \delta_i, q_{0i}, Q_i)$;
an NFA $V = (Q_V, \Sigma_1 \cup \ldots \cup \Sigma_n, \delta_V, q_{0v}, F_v)$.
Output: **true** if $A_1 \otimes \cdots \otimes A_n \otimes V$ is nonempty, **false** otherwise.

```
1  Q ← ∅; q_0 ← [q_01, ..., q_0n, q_0v]
2  W ← {q_0}
3  while W ≠ ∅ do
4     pick [q_1, ..., q_n, q] from W
5     add [q_1, ..., q_n, q] to Q
6     for all a ∈ Σ_1 ∪ ... ∪ Σ_n do
7        for all i ∈ [1..n] do
8           if a ∈ Σ_i then Q'_i ← δ_i(q_i, a) else Q'_i = {q_i}
9        Q' ← δ_V(q, a)
10       for all [q'_1, ..., q'_n, q'] ∈ Q'_1 × ... × Q'_n × Q' do
11          if ⋀_{i=1}^{n} q'_i ∈ F_i and q ∈ F_v then return true
12          if [q'_1, ..., q'_n, q'] ∉ Q then add [q'_1, ..., q'_n, q'] to
W
13 return false
```

Table 8.2: Algorithm to check violation of a property.

Proof: We only give a high-level sketch of the proof. To prove that the problem is in PSPACE, we show that it belongs to NPSPACE and apply Savitch's theorem. The polynomial-space nondeterministic algorithm just guesses an execution of the product, one configuration at a time, leading to a final configuration. Notice that storing a configuration requires linear space.

PSPACE-hardness is proven by reduction from the acceptance problem for linearly bounded automata. A linearly bounded automaton (LBA) is a deterministic Turing machine that always halts and only uses the part of the tape containing the input. Given an LBA A, we construct in linear time a network of automata that "simulates" A. The network has one component modeling the control of A (notice that the control is essentially a DFA), and one component for each tape cell used by the input. The states of the control component are pairs (q, k), where q is a control state of A, and k is a head position. The states of a cell-component are the possible tape symbols. The transitions correspond to the possible moves of A according to its transition table. Acceptance of A corresponds to reachability of certain configurations in the network, which can be easily encoded as an emptiness problem. □

8.3.1 Symbolic State-space Exploration

Figure 8.6 shows again the program of Example 8.1, and its flowgraph. An edge of the flowgraph

```
1  while x = 1 do
2     if y = 1 then
3        x ← 0
4     y ← 1 − x
5  end
```

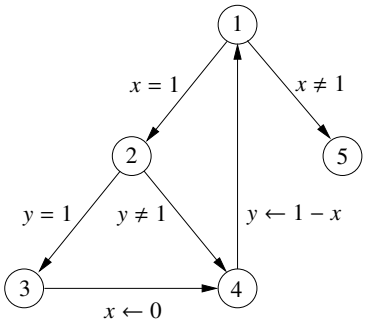


Figure 8.6: Flowgraph of the program of Example 8.1

leading from node ℓ to node ℓ' can be associated a *step relation* $S_{\ell,\ell'}$ containing all pairs of configurations $([\ell, x_0, y_0], [\ell', x_0', y_0'])$ such that if at control point ℓ the current values of the variables are x_0, y_0, then the program can take a step after which the new control point is ℓ', and the new values are x_0', y_0'. For instance, for the edge leading from node 4 to node 1 we have

$$S_{4,1} = \left\{ \left([4, x_0, y_0], [1, x_0', y_0'] \right) \mid x_0' = x_0, y_0' = 1 - x_0 \right\}$$

and for the edge leading from 1 to 2

$$S_{1,2} = \left\{ \left([1, x_0, y_0], [2, x_0', y_0'] \right) \mid x_0 = 1 = x_0', y_0' = y_0 \right\}$$

It will be convenient to assign a relation to every pair of nodes of the control graph, even to those not connected by any edge. If no edge leads from a to b, then we define $R_{a,b} = \emptyset$. The complete program is then described by the global step relation

$$S = \bigcup_{a,b \in C} S_{a,b}$$

where C is the set of control points.

Given a set I of initial configurations, the set of configurations reachable from I can be computed by the following algorithm, which repeatedly applies the **Post** operation:

Reach(I, R)
Input: set I of initial configurations; relation R
Output: set of configurations reachable form I

```
1  OldP ← ∅; P ← I
2  while P ≠ OldP do
3     OldP ← P
4     P ← Union(P, Post(P, S))
5  return P
```

The algorithm can be implemented using different data structures. The verification community distinguishes between *explicit* and *symbolic* data structures. Explicit data structures store separately each of the configurations of P, and the pairs of configurations of S; typical examples are lists and hash tables. Their distinctive feature is that the memory needed to store a set is proportional to the number of its elements. Symbolic data structures, on the contrary, do not store a set by storing each of its elements; they store a representation of the set itself. A prominent example of a symbolic data structure are finite automata and transducers: given an encoding of configurations as words over some alphabet Σ, the set P and the step relation S are represented by an automaton and a transducer, respectively, recognizing the encodings of its elements. Their sizes can be much smaller than the sizes of P or S. For instance, if P is the set of all possible configurations then its encoding is usually Σ^*, which is encoded by a very small automaton.

Symbolic data structures are only useful if all the operations required by the algorithm can be implemented without having to switch to an explicit data structure. This is the case of automata and transducers: **Union**, **Post**, and the equality check in the condition of the while loop operation are implemented by the algorithms of Chapters 4 and 5, or, if they are fixed-length, by the algorithms of Chapter 6.

Symbolic data structures are interesting when the set of reachable configurations can be very large, or even infinite. When the set is small, the overhead of symbolic data structures usually offsets the advantage of a compact representation. Despite this, and in order to illustrate the method, we apply it to the five-line program of Figure 8.6. The fixed-length transducer for the step relation S is shown in Figure 8.7; a configuration $[\ell, x_0, y_0]$ is encoded by the word $\ell x_0 y_0$ of length 3. Consider for instance the transition labeled by $\begin{bmatrix}4\\1\end{bmatrix}$. Using it the transducer can recognize four pairs, which describe the action of the instruction $y \leftarrow 1 - x$, namely

$$\begin{bmatrix}400\\101\end{bmatrix} \quad \begin{bmatrix}401\\101\end{bmatrix} \quad \begin{bmatrix}410\\111\end{bmatrix} \quad \begin{bmatrix}411\\110\end{bmatrix}.$$

Figure 8.8 shows minimal DFAs for the set I and for the sets obtained after each iteration of the while loop.

Variable orders.

We have defined a configuration of the program of Example 8.1 as a triple $[\ell, n_x, n_y]$, and we have encoded it as the word $\ell n_x n_y$. We could have also encoded it as the word $n_x \ell n_y$, $n_y \ell n_x$, or as any other permutation, since in all cases the information content is the same. Of course, when encoding a set of configurations all the configurations must be encoded using the same *variable order*.

While the information content is independent of the variable order, the *size* of the automaton encoding a set is not. An extreme case is given by the following example.

Example 8.9 Consider the set of tuples $X = \{[x_1, x_2, \ldots, x_{2k}] \mid x_1, \ldots, x_{2k} \in \{0, 1\}\}$, and the subset $Y \subseteq X$ of tuples satisfying $x_1 = x_k, x_2 = x_{k+1}, \ldots, x_k = x_{2k}$. Consider two possible encodings of a

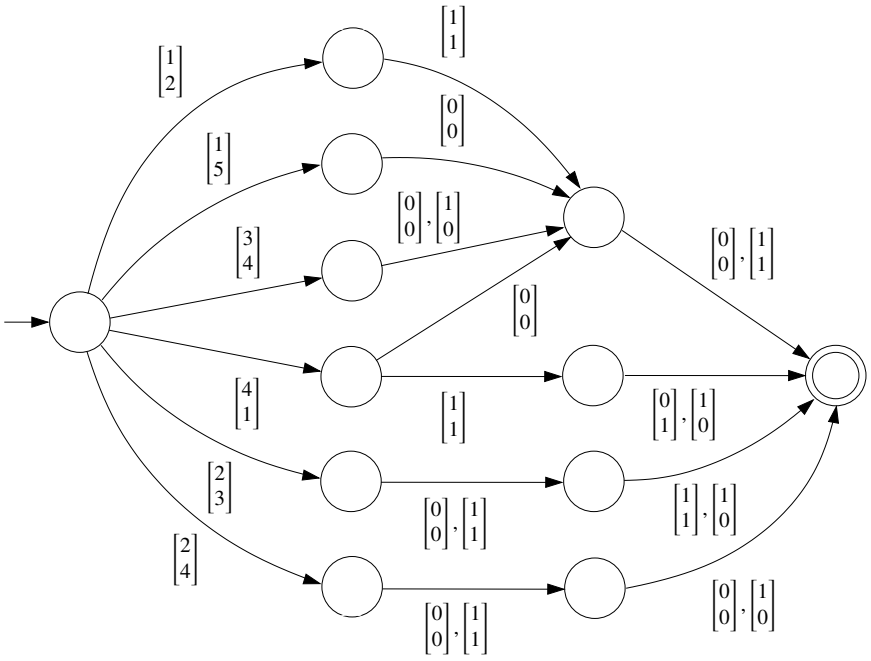


Figure 8.7: Transducer for the program of Figure 8.6

tuple $[x_1, x_2, \ldots, x_{2k}]$: by the word $x_1 x_2 \ldots x_{2k}$, and by the word $x_1 x_{k+1} x_2 x_{k+2} \ldots x_k x_{2k}$. In the first case, the encoding of Y for $k = 3$ is the language

$$L_1 = \{000000, 001001, 010010, 011011, 100100, 101101, 110110, 111111\}$$

and in the second the language

$$L_2 = \{000000, 000011, 001100, 001111, 110000, 110011, 111100, 111111\}$$

Figure 8.9 shows the minimal DFAs for the languages L_1 and L_2. It is easy to see that the minimal DFA for L_1 has at least 2^k states: since for every word $w \in \{0,1\}^k$ the residual L_1^w is equal to $\{w\}$, the language L_1 has a different residual for each word of length k, and so the minimal DFA has at least 2^k states (the exact number is $2^{k+1} + 2^k - 2$). On the other hand, it is easy to see that the minimal DFA for L_2 has only $3k + 1$ states. So a good variable order can lead to a exponentially more compact representation. □

We can also appreciate the effect of the variable order in Lamport's algorithm. The set of reachable configurations, sorted according to the state of the first process and then to the state of

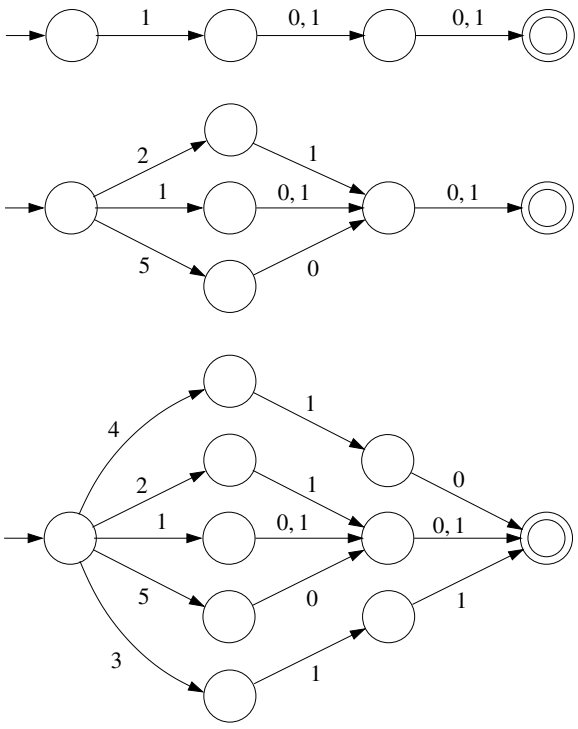


Figure 8.8: Minimal DFAs for the reachable configurations of the program of Figure 8.6

the second process, is

$$\begin{array}{lll}
\langle nc_0, nc_1, 0, 0\rangle & \langle t_0, nc_1, 1, 0\rangle & \langle c_0, nc_1, 1, 0\rangle \\
\langle nc_0, t_1, 0, 1\rangle & \langle t_0, t_1, 1, 1\rangle & \langle c_0, t_1, 1, 1\rangle \\
\langle nc_0, c_1, 0, 1\rangle & \langle t_0, c_1, 1, 1\rangle & \\
\langle nc_0, q_1, 0, 1\rangle & \langle t_0, q_1, 1, 1\rangle & \langle c_0, q_1, 1, 1\rangle \\
\langle nc_0, q_1', 0, 0\rangle & \langle t_0, q_1', 1, 0\rangle & \langle c_0, q_1', 1, 0\rangle
\end{array}$$

If we encode a tuple $\langle s_0, s_1, v_0, v_1\rangle$ by the word $v_0 s_0 s_1 v_1$, the set of reachable configurations is recognized by the minimal DFA on the left of Figure 8.10. However, if we encode by the word $v_1 s_1 s_0 v_0$ we get the minimal DFA on the right. The same example can be used to visualize how by adding configurations to a set the size of its minimal DFA can decrease. If we add the "missing" configuration $\langle c_0, c_1, 1, 1\rangle$ to the set of reachable configurations (filling the "hole" in the list above), two states of the DFAs of Figure 8.10 can be merged, yielding the minimal DFAs of Figure 8.11. Observe also that the set of all configurations, reachable or not, contains 120 elements, but is recognized by a five-state DFA.

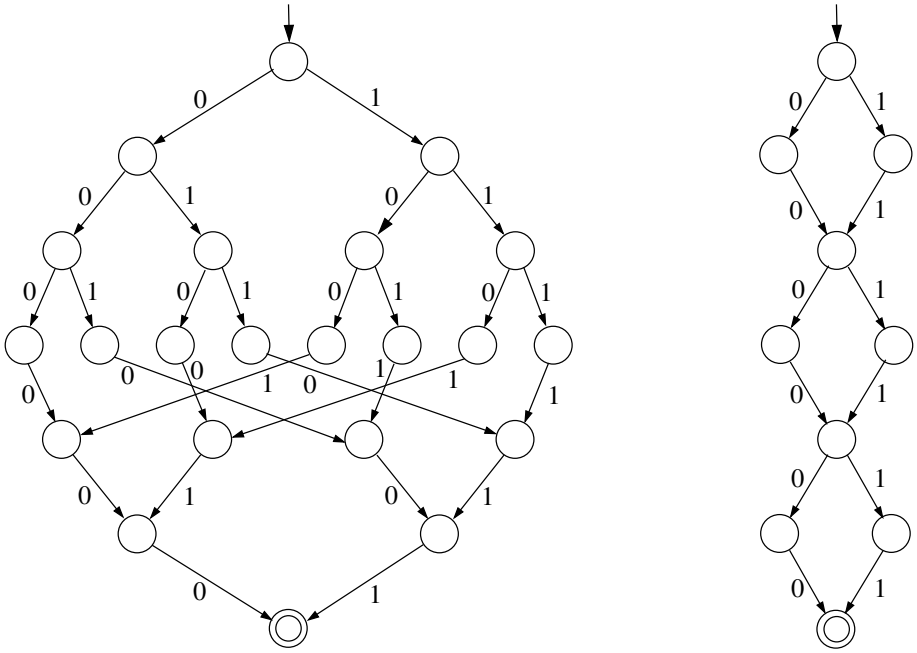


Figure 8.9: Minimal DFAs for the languages L_1 and L_2

8.4 Safety and Liveness Properties

Apart from the state-explosion problem, the automata-theoretic approach to automatic verification described in this chapter has a second limitation: it assumes that the violations of the property can be described by a set of finite executions. In other words, and loosely speaking, it assumes that if the property is violated, then it is already violated after a finite amount of time, independently of the rest of the execution. Not all properties are of this form. A typical example is the property "if a process requests access to the critical section, it eventually enters the critical section" (without specifying how long it may take). After finite time we can only tell that the process has not entered the critical section *yet*, but it might enter it in the future. A violation of the property can only be witnessed by an *infinite* execution, in which we observe that the process requests access, but the access is never granted.

Properties which are violated by finite executions are called *safety properties*. Intuitively, they correspond to properties of the form "nothing bad ever happens". Typical examples are "the system never deadlocks", or, more generally, "the system never enters a set of bad states". Clearly, every interesting system must also satisfy properties of the form "something good eventually happens", because otherwise the system that does nothing would already satisfy all properties. Properties of this kind are called *liveness properties*, and can only be witnessed by *infinite* executions. Fortunately, the automata-theoretic approach can be extended to liveness properties. This requires to

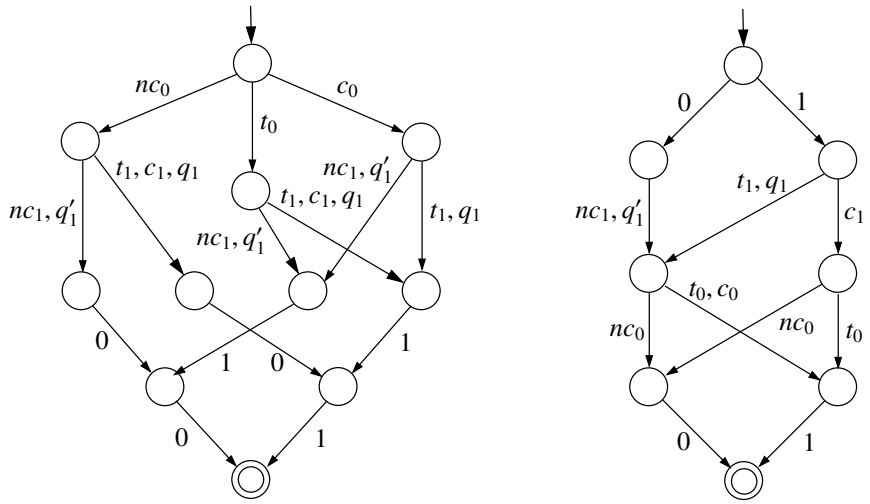


Figure 8.10: Minimal DFAs for the reachable configurations of Lamport's algorithm. On the left a configuration $\langle s_0, s_1, v_0, v_1, q\rangle$ is encoded by the word $s_0 s_1 v_0 v_1 q$, on the right by $v_1 s_1 s_0 v_0$.

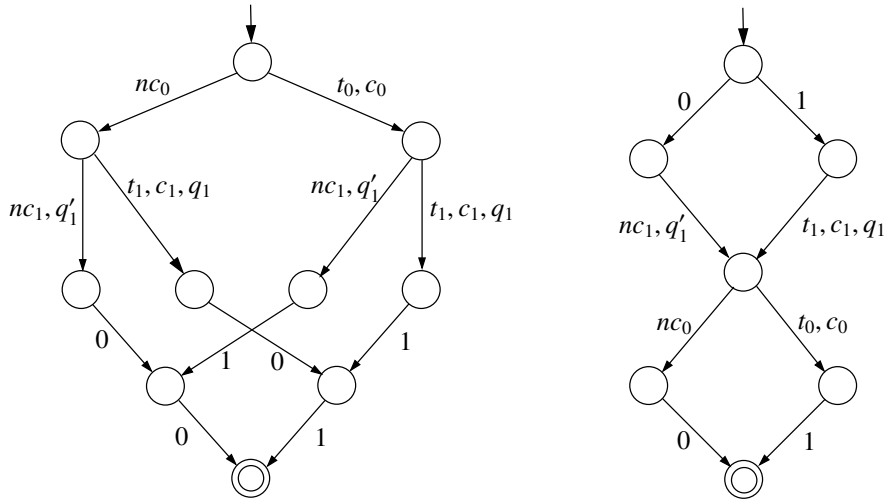


Figure 8.11: Minimal DFAs for the reachable configurations of Lamport's algorithm plus $\langle c_0, c_1, 1, 1\rangle$.

develop a theory of automata on infinite words, which is the subject of the second part of this book. The application of this theory to the verification of liveness properties is presented in Chapter 14.

Exercises

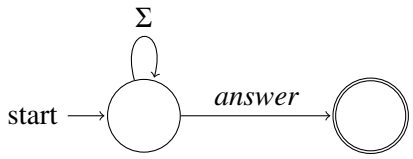
Exercise 68 Let $\Sigma = \{request, answer, working, idle\}$.

1. Build an automaton recognizing all words with the property P_1: for every occurrence of *request* there is a later occurrence of *answer*.

2. P_1 does not imply that every occurrence of *request* has "its own" *answer*: for instance, the sequence *request request answer* satisfies P_1, but both *request*s must necessarily be mapped to the same *answer*. But, if words were infinite and there were infinitely many *request*s, would P_1 guarantee that every *request* has its own *answer*?
   More precisely, let $w = w_1 w_2 \cdots$ satisfying P_1 and containing infinitely many occurrences of *request*, and define $f : \mathbb{N} \to \mathbb{N}$ such that $w_{f(i)}$ is the ith *request* in w. Is there always an injective function $g : \mathbb{N} \to \mathbb{N}$ satisfying $w_{g(i)} = answer$ and $f(i) < g(i)$ for all $i \in \{1, \ldots, k\}$?

3. Build an automaton recognizing all words with the property P_2: there is an occurrence of *answer*, and before that only *working* and *request* occur.

4. Using the intersection construction, prove that all accepting runs of the automaton below satisfy P_1. Find all accepting runs violating P_2.



Exercise 69 This exercise focuses on modelling and verification of mutual exclusion (mutex) algorithms. Consider two processes running the following mutex algorithm, where *id* is an identifier, a local variable having value 0 for one of the processes and value 1 for the other.

while true do
  loop-arbitrarily-many-times
    non-critical-command
  enter(*id*)
    critical-command
  leave(*id*)

The procedures *enter*() and *leave*() are specified below. They use a global variable *turn*, initially set to 0.

proc *enter*(i)
  while $turn = 1 - i$ **do skip**

proc *leave*(i)
  $turn \leftarrow 1 - i$

Design an asynchronous network of automata capturing this algorithm. Furthermore, build an automaton recognizing all runs reaching a configuration with both agents in the critical section. Using the intersection algorithm, prove that there are no such runs of this system, i.e. it is a *mutex* algorithm. Do all infinite runs satisfy that if a process wants to enter the critical section then it eventually enters it?

Consider now a different definition of *enter* and *leave*. Now the processes use two booelan variables *flag*[0] and *flag*[1] initially set to *false*.

proc *enter*(i)
  $flag[i] \leftarrow$ **true**
  while $flag[1 - i]$ **do skip**

proc *leave*(i)
  $flag[i] \leftarrow$ **false**

Design an asynchronous network of automata capturing this behaviour. Can a deadlock occur?

Finally, Peterson's algorithm combines both approache. The processes use variables *turn*, initially set to 0, and *flag*[0], *flag*[1], initially set to **false**. The procedures *enter* and *leave* are defined as follows:

proc *enter*(i)
  $turn \leftarrow 1 - i$
  $flag[i] \leftarrow$ **true**
  while $flag[1 - i]$ **and** $turn = 1 - i$ **do skip**

proc *leave*(i)
  $flag[i] \leftarrow$ **false**

Can a deadlock occur? What kind of starvation can occur?

Exercise 70 Consider a circular railway divided into 8 tracks: $0 \to 1 \to \ldots \to 7 \to 0$. In the railway circulate three trains, modeled by three automata T_1, T_2, and T_3. Each automaton T_i has states $\{q_{i,0}, \ldots, q_{i,7}\}$, alphabet $\{enter[i, j] \mid 0 \leq j \leq 7\}$ (where $enter[i, j]$ models that train i enters track j), transition relation $\{(q_{i,j}, enter[i, j \oplus 1], q_{i,j\oplus 1}) \mid 0 \leq j \leq 7\}$, and initial state $q_{i,2i}$, where $\oplus$ denotes addition modulo 8.

Define automata $C_0, \ldots, C_7$ (the *local controllers*) to make sure that two trains can never be on the same or adjacent tracks (i.e., there must always be at least one empty track between two trains). Each controler C_j can only have knowledge of the state of the tracks $j \ominus 1$, j, and $j \oplus 1$, . There must be no deadlocks, and every train must eventually visit every track. More formally, the asynchronous product $R = C_0 \otimes \ldots \otimes C_7 \otimes T_1 \otimes T_2 \otimes T_3$ must satisfy the following specification:

- For $j = 0, \ldots, 7$: C_j has alphabet $\{enter[i, j \ominus 1], enter[i, j], enter[i, j \oplus 1], \mid 1 \leq i \leq 3\}$. ($C_j$ only knows the state of tracks $j \ominus 1$, j, and $j \oplus 1$.)

- For $i = 1, 2, 3$: $L(R) \mid_{\Sigma_i} = (\, enter[i, 2i]\ enter[i, 2i \oplus 1] \ldots\ enter[i, 2i \oplus 7]\,)^*$.
  (No deadlocks, and every train eventually visits every segment.)

- For every word $w \in L(R)$: if $w = w_1\, enter[i, j]\, enter[i', j']\, w_2$ and $i' \neq i$, then $|j - j'| \notin \{0, 1, 7\}$.
  (No two trains on the same or adjacent tracks.)

Chapter 9

Automata and Logic

A regular expression can be seen as a set of instructions (a 'recipe') for generating the words of a language. For instance, the expression $aa(a + b)^*b$ can be read as "write two a's, repeatedly write a or b an arbitrary number of times, and then write a b". We say that regular expressions are an *operational* description language.

Languages can also be described *intensionally*, as the set of words that satisfy a property. For instance, "the words over $\{a, b\}$ containing an even number of a's and an even number of b's" is an intensional description. A given language may have a simple intensional description and a complicated operational description as a regular expression. For instance, $(aa + bb + (ab + ba)(aa + bb)^*(ba + ab))^*$ is a natural operational description of the language above, and it is arguably less intuitive than the intensional one. This becomes even more clear if we consider the language of the words over $\{a, b, c\}$ containing an even number of a's, of b's, and of c's.

In this chapter we present a logical formalism for the intensional description of regular languages. We use logical formulas to describe properties of words, and logical operators to construct complex properties out of simpler ones. We then show how to automatically translate a formula describing a property of words into an automaton recognizing the words satisfying the property. As a consequence, we obtain an algorithm to convert operational into intensional descriptions, and vice versa.

9.1 First-Order Logic on Words

A language is intensionally defined by its *membership predicate*, i.e., the property that words must satisfy in order to belong to it. For instance, we intensionally define the language $\{a, aa, aba, abba, abbba, \ldots\}$ as the language of all words that begin and end with an a, and otherwise contain only bs.

Predicate logic is the standard language to express membership predicates. Starting from some natural, "atomic" predicates, more complex ones can be constructed through boolean combinations and quantification. We introduce atomic predicates $Q_a(x)$, where a is a letter, and x ranges over the positions of the word. The intended meaning is "the letter at position x is an a." For instance, the

property "all letters are *a*s" is formalized by the formula $\forall x\, Q_a(x)$.

In order to express relations between positions we add to the syntax the predicate $x < y$, with intended meaning "position x is smaller than position y". For example, the property "if the letter at a position is an a, then all subsequent letters are also *a*s" is formalized by the formula

$$\forall x \forall y\, ((Q_a(x) \wedge x < y) \rightarrow Q_a(y))\ .$$

Definition 9.1 *Let $V = \{x, y, z, \ldots\}$ be an infinite sets of* variables*, and let $\Sigma = \{a, b, c, \ldots\}$ be a finite alphabet. The set $FO(\Sigma)$ of* first-order formulas *over Σ is the set of expressions generated by the grammar:*

$$\varphi := Q_a(x) \mid x < y \mid \neg\varphi \mid (\varphi \vee \varphi) \mid \exists x\, \varphi$$

where $a \in \Sigma$.

As usual, variables within the scope of an existential quantifier are *bounded*, and otherwise *free*. A formula without free variables is a *sentence*. Sentences of $FO(\Sigma)$ are interpreted on words over Σ. For instance, $\forall x\, Q_a(x)$ is true for the word *aa*, but false for word *ab*. Formulas with free variables cannot be interpreted on words alone: it does not make sense to ask whether $Q_a(x)$ holds for the word *ab* or not. A formula with free variables is interpreted over a pair $(w, \mathcal{I})$, where $\mathcal{I}$ assigns to each free variable (and perhaps to others) a position in the word. For instance, $Q_a(x)$ is true for the pair $(ab, x \mapsto 1)$, because the letter at position 1 of *ab* is *a*, but false for $(ab, x \mapsto b)$.

Definition 9.2 *An* interpretation *of a formula φ of $FO(\Sigma)$ is a pair $(w, \mathcal{I})$ where $w \in \Sigma^*$ and $\mathcal{I}$ is a mapping that assigns to every free variable x a* position $\mathcal{I}(x) \in \{1, \ldots, |w|\}$ *(the mapping may also assign positions to other variables).*

Notice that if φ is a sentence then a pair $(w, \mathcal{E})$, where $\mathcal{E}$ is the empty mapping that does not assign any position to any variable, is an interpretation of φ. Instead of $(w, \mathcal{E})$ we write simply w.

We now formally define when an interpretation satisfies a formula. Given a word w and a number k, let $w[k]$ denote the letter of w at position k.

Definition 9.3 *The satisfaction relation $(w, \mathcal{I}) \models \varphi$ between a formula φ of $FO(\Sigma)$ and an interpretation $(w, \mathcal{I})$ of φ is defined by:*

$$\begin{array}{lcll} (w, \mathcal{I}) & \models & Q_a(x) & \textit{iff} \quad w[\mathcal{I}(x)] = a \\ (w, \mathcal{I}) & \models & x < y & \textit{iff} \quad \mathcal{I}(x) < \mathcal{I}(y) \\ (w, \mathcal{I}) & \models & \exists x\, \varphi & \textit{iff} \quad |w| \geq 1 \textit{ and some } i \in \{1, \ldots, |w|\} \textit{ satisfies } (w, \mathcal{I}[i/x]) \models \varphi \end{array}$$

where $w[i]$ is the letter of w at position i, and $\mathcal{I}[i/x]$ is the interpretation that assigns i to x and otherwise coincides with $\mathcal{I}$. (Notice that $\mathcal{I}$ may not assign any value to x.) If $(w, \mathcal{I}) \models \varphi$ we say that $(w, \mathcal{I})$ is a model *of φ. Two formulas are* equivalent *if they have the same models.*

It follows easily from this definition that if two interpretations $(w, \mathfrak{I}_1)$ and $(w, \mathfrak{I}_2)$ of φ differ only in the positions assigned by $\mathfrak{I}_1$ and $\mathfrak{I}_2$ to bounded variables, then either both interpretations are models of φ, or none of them is. In particular, whether an interpretation $(w, \mathfrak{I})$ of a sentence is a model or not depends only on w, not on $\mathfrak{I}$.

Notice that according to the definition of the satisfaction relation the empty word ϵ satisfies no formulas of the form $\exists x\, \varphi$, and all formulas of the form $\forall x\, \varphi$. While this causes no problems for our purposes, it is worth noticing that in other contexts it may lead to complications. For instance, the formulas $\exists x\, Q_a(x)$ and $\forall y \exists x\, Q_a(x)$ do not hold for exactly the same words, because the empty word satisfies the second, but not the first.

We use some standard abbreviations:

$$\forall x\, \varphi := \neg \exists\, x \neg \varphi \qquad \varphi_1 \wedge \varphi_2 := \neg\, (\neg \varphi_1 \vee \neg \varphi_2) \qquad \varphi_1 \to \varphi_2 := \neg \varphi_1 \vee \varphi_2$$

Further useful abbreviations are:

$\text{first}(x)$	:=	$\neg \exists y\, y < x$	"x is the first position"
$\text{last}(x)$	:=	$\neg \exists y\, x < y$	"x is the last position"
$y = x + 1$	:=	$x < y \wedge \neg \exists\, z (x < z \wedge z < y)$	"y is the successor position of x"
$y = x + 2$	:=	$\exists\, z (z = x + 1 \wedge y = z + 1)$	
$y = x + (k + 1)$	:=	$\exists\, z (z = x + k \wedge y = z + 1)$**??**	

Example 9.4 Some examples of properties expressible in the logic:

- "The last letter is a b and before it there are only a's."

$$\exists x\, Q_b(x) \wedge \forall x\, (\text{last}(x) \to Q_b(x) \ \wedge\ \neg \text{last}(x) \to Q_a(x))$$

- "Every a is immediately followed by a b."

$$\forall x\, (Q_a(x) \to \exists y\, (y = x + 1 \wedge Q_b(y)))$$

- "Every a is immediately followed by a b, unless it is the last letter."

$$\forall x\, (Q_a(x) \to \forall y\, (y = x + 1 \to Q_b(y)))$$

- "Between every a and every later b there is a c."

$$\forall x \forall y\, (Q_a(x) \wedge Q_b(y) \wedge x < y \to \exists z\, (x < z \wedge z < y \wedge Q_c(z)))$$

□

9.1.1 Expressive power of $FO(\Sigma)$

Once we have defined which words satisfy a sentence, we can associate to a sentence the set of words satisfying it.

Definition 9.5 *The language $L(\varphi)$ of a sentence $\varphi \in FO(\Sigma)$ is the set $L(\varphi) = \{w \in \Sigma^* \mid w \models \phi\}$. We also say that φ* expresses *$L(\varphi)$. A language $L \subseteq \Sigma^*$ is* FO-definable *if $L = L(\varphi)$ for some formula φ of $FO(\Sigma)$.*

The languages of the properties in the example are FO-definable by definition. To get an idea of the expressive power of $FO(\Sigma)$, we prove a theorem characterizing the FO-definable languages in the case of a 1-letter alphabet $\Sigma = \{a\}$. In this simple case we only have one predicate $Q_a(x)$, which is always true in every interpretation. So every formula is equivalent to a formula without any occurrence of $Q_a(x)$. For example, the formula $\exists y\,(Q_a(y) \wedge y < x)$ is equivalent to $\exists y\; y < x$.

We prove that a language over a one-letter alphabet is FO-definable if and only if it is finite or *co-finite*, where a language is co-finite if its complement is finite. So, for instance, even a simple language like $\{a^n \mid n \text{ is even }\}$ is not FO-definable. The plan of the proof is as follows. First, we define the *quantifier-free fragment* of $FO(\{a\})$, denoted by QF; then we show that 1-letter languages are QF-definable iff they are finite or co-finite; finally, we prove that 1-letter languages are FO-definable iff they are QF-definable.

For the definition of QF we need some more macros whose intended meaning should be easy to guess:

$$\begin{array}{rcl} x + k < y & := & \exists z\,(z = x + k \wedge z < y) \\ x < y + k & := & \exists z\,(z = y + k \wedge x < z) \\ k < last & := & \forall x\,(\text{last}(x) \rightarrow x > k) \end{array}$$

Macros like $k > x$ or $x + k > y$ are defined similarly.

Definition 9.6 *The logic QF (for quantifier-free) is the fragment of $FO(\{a\})$ with syntax*

$$f := x \approx k \mid x \approx y + k \mid k \approx last \mid f_1 \vee f_2 \mid f_1 \wedge f_2$$

where $\approx \in \{<, >\}$ and $k \in \mathbb{N}$.

Proposition 9.7 *A language over a 1-letter alphabet is QF-definable iff it is finite or co-finite.*

Proof: ($\Rightarrow$): Let f be a sentence of QF. Since QF does not have quantifiers, f does not contain any occurrence of a variable, and so it is a positive (i.e., negation-free) boolean combination of formulas of the form $k < last$ or $k > last$. We proceed by induction on the structure of f. If $f = k < last$, then $L(\varphi)$ is co-finite, and if $f = k > last$, then $L(\varphi)$ is finite. If $f = f_1 \vee f_2$, then by induction hypothesis $L(f_1)$ and $L(f_2)$ are finite or co-finite; if $L(f_1)$ and $L(f_2)$ are finite, then so is $L(f)$, and otherwise $L(f)$ is co-finite. The case $f = f_1 \wedge f_2$ is similar.

($\Leftarrow$): A finite language $\{a^{k_1}, \ldots, a^{k_n}\}$ is expressed by the formula $(last > k_1 - 1 \wedge last < k_1 + 1) \vee \ldots \vee (last \leq k_n \wedge last \geq k_n)$. To express a co-finite language, it suffices to show that for

every formula f of QF expressing a language L, there is another formula $\overline{f}$ expressing the language $\overline{L}$. This is easily proved by induction on the structure of the formula. □

Theorem 9.8 *Every formula φ of $FO(\{a\})$ is equivalent to a formula f of QF.*

Proof: *Sketch.* By induction on the structure of φ. If $\varphi(x, y) = x < y$, then $\varphi \equiv y < x + 0$. If $\varphi = \neg\psi$, the result follows from the induction hypothesis and the fact that negations can be removed using De Morgan's rules and equivalences like $\neg(x < y + k) \equiv x \geq y + k$. If $\varphi = \varphi_1 \vee \varphi_2$, the result follows directly from the induction hypothesis. Consider now the case $\varphi = \exists x\ \psi$. By induction hypothesis, ψ is equivalent to a formula f of QF, and we can assume that f is in disjunctive normal form, say $f = D_1 \vee \ldots \vee D_n$. Then $\varphi \equiv \exists x\ D_1 \vee \exists x\ D_2 \vee \ldots \vee \exists x\ D_n$, and so it suffices to find a formula f_i of QF equivalent to $\exists x\ D_i$.

The formula f_i is a conjunction of formulas containing all conjuncts of D_i with no occurrence of x, plus other conjuncts obtained as follows. For every *lower bound* $x < t_1$ of D_i, where $t_1 = k_1$ or $t_1 = x_1 + k_1$, and every *upper bound* of the form $x > t_2$, where $t_2 = k_1$ or $t_2 = x_1 + k_1$ we add to f_i a conjunct equivalent to $t_2 + 1 < t_1$. For instance, $y + 7 < x$ and $x < z + 3$ we add $y + 5 < z$. It is easy to see that $f_i \equiv \exists x\ D_i$. □

Corollary 9.9 *The language Even $= \{a^{2n} \mid n \geq 0\}$ is not first-order expressible.*

These results show that first-order logic cannot express all regular languages, not even over a 1-letter alphabet. For this reason we now introduce monadic second-order logic.

9.2 Monadic Second-Order Logic on Words

Monadic second-order logic extends first-order logic with variables $X, Y, Z, \ldots$ ranging over *sets* of positions, and with predicates $x \in X$, meaning "position x belongs to the set X. [1] It is allowed to quantify over both kinds of variables. Before giving a formal definition, let us informally see how this extension allows to describe the language *Even*. The formula states that the last position belongs to the set of even positions. A position belongs to this set iff it is the second position, or the second successor of another position in the set.

The following formula states that X is the set of even positions:

$$\begin{aligned} \text{second}(x) &:= \exists y\ (\text{first}(y) \wedge x = y + 1) \\ \mathit{Even}(X) &:= \forall x\ (x \in X \leftrightarrow (\text{second}(x) \vee \exists y\ (x = y + 2 \wedge y \in X))) \end{aligned}$$

For the complete formula, we observe that the word has even length if its last position is even:

$$\mathit{EvenLength} := \exists X\ (\mathit{Even}(X) \wedge \forall x\ (\text{last}(x) \rightarrow x \in X)\ )$$

We now define the formal syntax and semantics of the logic.

[1] More generally, second-order logic allows for variables ranging over relations of arbitrary arity. The monadic fragment only allows arity 1, which corresponds to sets.

Definition 9.10 *Let* $X_1 = \{x, y, z, \ldots\}$ *and* $X_2 = \{X, Y, Z, \ldots\}$ *be two infinite sets of* first-order *and* second-order variables. *Let* $\Sigma = \{a, b, c, \ldots\}$ *be a finite alphabet. The set* $MSO(\Sigma)$ *of* monadic second-order formulas *over* Σ *is the set of expressions generated by the grammar:*

$$\varphi := Q_a(x) \mid x < y \mid x \in X \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x\, \varphi \mid \exists X\, \varphi$$

An interpretation *of a formula* φ *is a pair* $(w, \mathcal{I})$ *where* $w \in \Sigma^*$*, and* $\mathcal{I}$ *is a mapping that assigns every free first-order variable* x *a* position $\mathcal{I}(x) \in \{1, \ldots, |w|\}$ *and every free second-order variable* X *a set of positions* $\mathcal{I}(X) \subseteq \{1, \ldots, |w|\}$*. (The mapping may also assign positions to other variables.)*

The satisfaction relation $(w, \mathcal{I}) \models \varphi$ *between a formula* φ *of* $MSO(\Sigma)$ *and an interpretation* $(w, \mathcal{I})$ *of* φ *is defined as for* $FO(\Sigma)$*, with the following additions:*

$$\begin{array}{lcll} (w, \mathcal{I}) & \models & x \in X & \textit{iff} \quad \mathcal{I}(x) \in \mathcal{I}(X) \\ (w, \mathcal{I}) & \models & \exists X\, \varphi & \textit{iff} \quad |w| > 0 \textit{ and some } S \subseteq \{0, \ldots, |w| - 1\} \\ & & & \quad \textit{satisfies } (w, \mathcal{I}[S/X]) \models \varphi \end{array}$$

where $\mathcal{I}[S/X]$ *is the interpretation that assigns* S *to* X *and otherwise coincides with* $\mathcal{I}$ *— whether* $\mathcal{I}$ *is defined for* X *or not. If* $(w, \mathcal{I}) \models \varphi$ *we say that* $(w, \mathcal{I})$ *is a* model *of* φ*. Two formulas are* equivalent *if they have the same models. The language* $L(\varphi)$ *of a sentence* $\varphi \in MSO(\Sigma)$ *is the set* $L(\varphi) = \{w \in \Sigma^* \mid w \models \phi\}$*. A language* $L \subseteq \Sigma^*$ *is* MSO-definable *if* $L = L(\varphi)$ *for some formula* $\varphi \in MSO(\Sigma)$*.*

Notice that in this definition the set S may be empty. So, for instance, ay interpretation that assigns the empty set to X is a model of the formula $\exists X\, \forall x\, \neg(x \in X)$.

We use the standard abbreviations

$$\forall x \in X\, \varphi \;:=\; \forall x\, (x \in X \to \varphi) \qquad \exists x \in X\, \varphi \;:=\; \exists x\, (x \in X \wedge \varphi)$$

9.2.1 Expressibility of $MSO(\Sigma)$

We show that the languages expressible in monadic second-order logic are exactly the regular languages. We start with an example.

Example 9.11 Let $\Sigma = \{a, b, c, d\}$. We construct a formula of $MSO(\Sigma)$ expressing the regular language $c^*(ab)^*d^*$. The membership predicate of the language can be informally formulated as follows:

> There is a block of consecutive positions X such that: before X there are only c's; after X there are only d's; in X b's and a's alternate; the first letter in X is an a and the last letter is a b.

The predicate is a conjunction of predicates. We give formulas for each of them.

- "X is a block of consecutive positions."

$$\text{Cons}(X) := \forall x \in X\ \forall y \in X\ \ (x < y \rightarrow (\forall z\ (x < z \wedge z < y) \rightarrow z \in X))$$

- "x lies before/after X."

$$\text{Before}(x, X) := \forall y \in X\ x < y \qquad \text{After}(x, X) := \forall y \in X\ y < x$$

- "Before X there are only c's."

$$\text{Before\_only\_c}(X) := \forall x\ \text{Before}(x, X) \rightarrow Q_c(x)$$

- "After X there are only d's."

$$\text{After\_only\_d}(X) := \forall x\ \text{After}(x, X) \rightarrow Q_d(x)$$

- "a's and b's alternate in X."

$$\begin{aligned}\text{Alternate}(X) := \ \forall x \in X\ \ (\ & Q_a(x) \rightarrow \forall y \in X\ (y = x + 1 \rightarrow Q_b(y)\ ) \\ & \wedge \\ & Q_b(x) \rightarrow \forall y \in X\ (y = x + 1 \rightarrow Q_a(y)\ )\ )\end{aligned}$$

- "The first letter in X is an a and the last is a b."

$$\begin{aligned}\text{First\_a}(X) &:= \forall x \in X\ \forall y\ (y < x \rightarrow \neg y \in X) \rightarrow Q_a(x) \\ \text{Last\_b}(X) &:= \forall x \in X\ \forall y\ (y > x \rightarrow \neg y \in X) \rightarrow Q_a(x)\end{aligned}$$

Putting everything together, we get the formula

$$\begin{aligned}\exists X(\ & \text{Cons}(X) \wedge \text{Before\_only\_c}(X) \wedge \text{After\_only\_d}(X) \wedge \\ & \text{Alternate}(X) \wedge \text{First\_a}(X) \wedge \text{Last\_b}(X)\ )\end{aligned}$$

Notice that the empty word is a model of the formula. because the empty set of positions satisfies all the conjuncts. □

Let us now directly prove one direction of the result.

Proposition 9.12 *If $L \subseteq \Sigma^*$ is regular, then L is expressible in MSO(Σ).*

Proof: Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA with $Q = \{q_0, \ldots, q_n\}$ and $L(A) = L$. We construct a formula φ_A such that for every $w \neq \epsilon$, $w \models \varphi_A$ iff $w \in L(A)$. If $\epsilon \in L(A)$, then we can extend the formula to $\varphi_A \vee \varphi'_A$, where φ'_A is only satisfied by the empty word (e.g. $\varphi'_A = \forall x\ x < x$).

We start with some notations. Let $w = a_1 \ldots a_m$ be a word over Σ, and let

$$P_q = \left\{ i \in \{1, \ldots, m\} \mid \hat{\delta}(q_0, a_0 \ldots a_i) = q \right\} .$$

In words, $i \in P_q$ iff A is in state q immediately *after* reading the letter a_i. Then A accepts w iff $m \in \bigcup_{q \in F} P_q$.

Assume we were able to construct formula $\text{Visits}(X_0, \ldots X_n)$ with free variables $X_0, \ldots X_n$ such that $\mathcal{I}(X_i) = P_{q_i}$ holds for *every* model $(w, \mathcal{I})$ and for every $0 \leq i \leq n$. In words, $\text{Visits}(X_0, \ldots X_n)$ is only true when X_i takes the value P_{q_i} for every $0 \leq i \leq n$. Then $(w, \mathcal{I})$ would be a model of

$$\psi_A := \exists X_0 \ldots \exists X_n\ \text{Visits}(X_0, \ldots X_n) \wedge \exists x \left(\text{last}(x) \wedge \bigvee_{q_i \in F} x \in X_i \right)$$

iff w has a last letter, and $w \in L$. So we could take

$$\varphi_A := \begin{cases} \psi_A & \text{if } q_0 \notin F \\ \psi_A \vee \forall x\ x < x & \text{if } q_0 \in F \end{cases}$$

Let us now construct the formula $\text{Visits}(X_0, \ldots X_n)$. The sets P_q are the unique sets satisfying the following properties:

(a) $1 \in P_{\delta(q_0, a_1)}$, i.e., after reading the letter at position 1 the DFA is in state $\delta(q_0, a_1)$;

(b) every position i belongs to exactly one P_q, i.e., the P_q's build a partition of the set positions; and

(c) if $i \in P_q$ and $\delta(q, a_{i+1}) = q'$ then $i + 1 \in P_{q'}$, i.e., the P_q's "respect" the transition function δ.

We express these properties through formulas. For every $a \in \Sigma$, let $q_{i_a} = \delta(q_0, a)$. The formula for (a) is:

$$\text{Init}(X_0, \ldots, X_n) = \exists x \left(\text{first}(x) \wedge \left(\bigvee_{a \in \Sigma} (Q_a(x) \wedge x \in X_{i_a}) \right) \right)$$

(in words: if the letter at position 1 is a, then the position belongs to X_{i_a}).
Formula for (b):

$$\text{Partition}(X_0, \ldots, X_n) = \forall x \left(\bigvee_{i=0}^{n} x \in X_i \ \wedge \bigwedge_{\substack{i,j=0 \\ i \neq j}}^{n} (x \in X_i \rightarrow x \notin X_j) \right)$$

Formula for (c):

$$\text{Respect}(X_0,\ldots,X_n) = \forall x \forall y \left(y = x+1 \rightarrow \bigvee_{\substack{a \in \Sigma \\ i,j \in \{0,\ldots,n\} \\ \delta(q_i,a) = q_j}} (x \in X_i \wedge Q_a(x) \wedge y \in X_j) \right)$$

Altogether we get

$$\text{Visits}(X_0,\ldots X_n) := \text{Init}(X_0,\ldots,X_n) \wedge \text{Partition}(X_0,\ldots,X_n) \wedge \text{Respect}(X_0,\ldots,X_n)$$

□

It remains to prove that MSO-definable languages are regular. Given a sentence $\varphi \in MSO(\Sigma)$ show that $L(\varphi)$ is regular by induction on the structure of φ. However, since the subformulas of a sentence are not necessarily sentences, the language defined by the subformulas of φ is not defined. We correct this. Recall that the interpretations of a formula are pairs $(w, \mathcal{I})$ where $\mathcal{I}$ assigns positions to the free first-order variables and sets of positions to the free second-order variables. For example, if $\Sigma = \{a, b\}$ and if the free first-order and second-order variables of the formula are x, y and X, Y, respectively, then two possible interpretations are

$$\left(aab\ , \begin{array}{l} x \mapsto 1 \\ y \mapsto 3 \\ X \mapsto \{2,3\} \\ Y \mapsto \{1,2\} \end{array} \right) \qquad \left(ba\ , \begin{array}{l} x \mapsto 2 \\ y \mapsto 1 \\ X \mapsto \emptyset \\ Y \mapsto \{1\} \end{array} \right)$$

Given an interpretation $(w, \mathcal{I})$, we can encode each assignment $x \mapsto k$ or $X \mapsto \{k_1, \ldots, k_l\}$ as a bitstring of the same length as w: the string for $x \mapsto k$ contains exactly a 1 at position k, and 0's everywhere else; the string for $X \mapsto \{k_1, \ldots, k_l\}$ contains 1's at positions $k_1, \ldots, k_l$, and 0's everywhere else. After fixing an order on the variables, an interpretation $(w, \mathcal{I})$ can then be encoded as a tuple $(w, w_1, \ldots, w_n)$, where n is the number of variables, $w \in \Sigma^*$, and $w_1, \ldots, w_n \in \{0,1\}^*$. Since all of $w, w_1, \ldots, w_n$ have the same length, we can as in the case of transducers look at $(w, w_1, \ldots, w_n)$ as a word over the alphabet $\Sigma \times \{0,1\}^n$. For the two interpretations above we get the encodings

	a	a	b
x	1	0	0
y	0	0	1
X	0	1	1
Y	1	1	0

and

	b	a
x	0	1
y	1	0
X	0	0
Y	1	0

corresponding to the words

$$\begin{bmatrix} a \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}\begin{bmatrix} a \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}\begin{bmatrix} b \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} b \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}\begin{bmatrix} a \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \text{of } \Sigma \times \{0,1\}^4$$

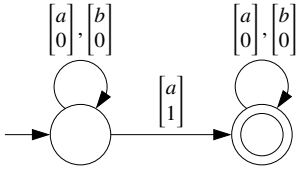
Definition 9.13 *Let φ be a formula with n free variables, and let $(w, \mathfrak{I})$ be an interpretation of φ. We denote by* $\mathrm{enc}(w, \mathfrak{I})$ *the word over the alphabet $\Sigma \times \{0,1\}^n$ described above. The language of φ is $L(\varphi) = \{\mathrm{enc}(w, \mathfrak{I}) \mid (w, \mathfrak{I}) \models \varphi\}$.*

Now that we have associate to every formula φ a language (whose alphabet depends on the free variables), we prove by induction on the structure of φ that $L(\varphi)$ is regular. We do so by exhibiting automata (actually, transducers) accepting $L(\varphi)$. For simplicity we assume $\Sigma = \{a, b\}$, and denote by $\mathit{free}(\varphi)$ the set of free variables of φ.

- $\varphi = Q_a(x)$. Then $\mathit{free}(\varphi) = x$, and the interpretations of φ are encoded as words over $\Sigma \times \{0,1\}$. The language $L(\varphi)$ is given by

$$L(\varphi) = \left\{ \begin{bmatrix} a_1 \\ b_1 \end{bmatrix} \cdots \begin{bmatrix} a_k \\ b_k \end{bmatrix} \;\middle|\; \begin{array}{l} k \geq 0, \\ a_i \in \Sigma \text{ and } b_i \in \{0,1\} \text{ for every } i \in \{1,\ldots,k\}, \text{ and} \\ b_i = 1 \text{ for exactly one index } i \in \{1,\ldots,k\} \text{ such that } a_i = a \end{array} \right\}$$

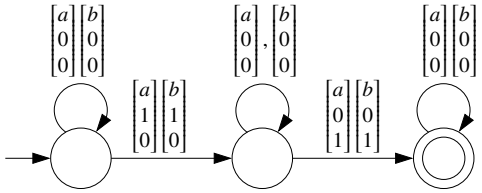
  and is recognized by



- $\varphi = x < y$. Then $\mathit{free}(\varphi) = \{x, y\}$, and the interpretations of ϕ are encoded as words over $\Sigma \times \{0,1\}^2$. The language $L(\varphi)$ is given by

$$L(\varphi) = \left\{ \begin{bmatrix} a_1 \\ b_1 \\ c_1 \end{bmatrix} \cdots \begin{bmatrix} a_k \\ b_k \\ c_k \end{bmatrix} \;\middle|\; \begin{array}{l} k \geq 0, \\ a_i \in \Sigma \text{ and } b_i, c_i \in \{0,1\} \text{ for every } i \in \{1,\ldots,k\}, \\ b_i = 1 \text{ for exactly one index } i \in \{1,\ldots,k\}, \\ c_j = 1 \text{ for exactly one index } j \in \{1,\ldots,k\}, \text{ and} \\ i < j \end{array} \right\}$$

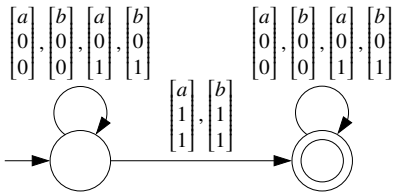
  and is recognized by



- $\varphi = x \in X$. Then $free(\varphi) = \{x, X\}$, and interpretations are encoded as words over $\Sigma \times \{0,1\}^2$. The language $L(\varphi)$ is given by

$$L(\varphi) = \left\{ \begin{bmatrix} a_1 \\ b_1 \\ c_1 \end{bmatrix} \dots \begin{bmatrix} a_k \\ b_k \\ c_k \end{bmatrix} \;\middle|\; \begin{array}{l} k \geq 0, \\ a_i \in \Sigma \text{ and } b_i, c_i \in \{0,1\} \text{ for every } i \in \{1,\dots,k\}, \\ b_i = 1 \text{ for exactly one index } i \in \{1,\dots,k\}, \text{ and} \\ \text{for every } i \in \{1,\dots,k\}, \text{ if } b_i = 1 \text{ then } c_i = 1 \end{array} \right\}$$

  and is recognized by



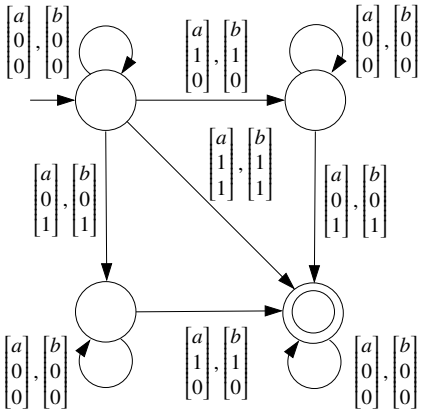
- $\varphi = \neg\psi$. Then $free(\varphi) = free(\psi)$, and by induction hypothesis there exists an automaton A_ψ s.t. $L(A_\psi) = L(\psi)$.

  Observe that $L(\varphi)$ is *not* in general equal to $\overline{L(\psi)}$. To see why, consider for example the case $\psi = Q_a(x)$ and $\varphi = \neg Q_a(x)$. The word

$$\begin{bmatrix} a \\ 1 \end{bmatrix} \begin{bmatrix} a \\ 1 \end{bmatrix} \begin{bmatrix} a \\ 1 \end{bmatrix}$$

  belongs neither to $L(\psi)$ nor $L(\varphi)$, because it is not the encoding of any interpretation: the bitstring for x contains more than one 1. What holds is $L(\varphi) = \overline{L(\psi)} \cap Enc(\psi)$, where $Enc(\psi)$ is the language of the encodings of all the interpretations of ψ (whether they are models of ψ or not). We construct an automaton A_ψ^{enc} recognizing $Enc(\psi)$, and so we can take $A_\varphi = A_\psi \cap A_\psi^{enc}$.

  Assume ψ has k first-order variables. Then a word belongs to $Enc(\psi)$ iff each of its projections onto the 2nd, 3rd, …, $(k+1)$-th component is a bitstring containing exactly one 1. As states of A_ψ^{enc} we take all the strings $\{0,1\}^k$. The intended meaning of a state, say state 101 for the case $k = 3$, is "the automaton has already read the 1's in the bitstrings of the first and third variables, but not yet read the 1 in the second." The initial and final states are 0^k and 1^k, respectively. The transitions are defined according to the intended meaning of the states. For instance, the automaton $A_{x<y}^{enc}$ is

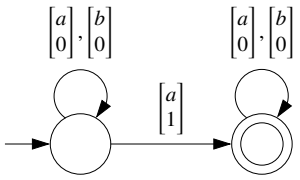


Observe that the number of states of A_ψ^{enc} grows exponentially in the number of free variables. This makes the negation operation expensive, even when the automaton A_ϕ is deterministic.

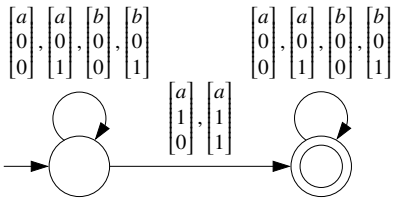
- $\varphi = \varphi_1 \vee \varphi_2$. Then $free(\varphi) = free(\varphi_1) \cup free(\varphi_2)$, and by induction hypothesis there are automata A_{φ_i}, A_{φ_2} such that $\mathcal{L}(A_{\varphi_1}) = \mathcal{L}(\varphi_1)$ and $\mathcal{L}(A_{\varphi_2}) = \mathcal{L}(\varphi_2)$.

  If $free(\varphi_1) = free(\varphi_2)$, then we can take $A_\varphi = A_{\varphi_1} \cup A_{\varphi_2}$. But this need not be the case. If $free(\varphi_1) \neq free(\varphi_2)$, then $\mathcal{L}(\varphi_1)$ and $\mathcal{L}(\varphi_2)$ are languages over different alphabets Σ_1, Σ_2, or over the same alphabet, but with different intended meaning, and we cannot just compute their intersection. For example, if $\varphi_1 = Q_a(x)$ and $\varphi_2 = Q_b(y)$, then both $\mathcal{L}(\varphi_1)$ and $\mathcal{L}(\varphi_2)$ are languages over $\Sigma \times \{0, 1\}$, but the second component indicates in the first case the value of x, in the second the value of y.

  This problem is solved by extending $\mathcal{L}(\varphi_1)$ and $\mathcal{L}(A_{\varphi_2})$ to languages L_1 and L_2 over $\Sigma \times \{0, 1\}^2$. In our example, the language L_1 contains the encodings of all interpretations $(w, \{x \mapsto n_1, y \mapsto n_2\})$ such that the projection $(w, \{x \mapsto n_1\})$ belongs to $\mathcal{L}(Q_a(x))$, while L_2 contains the interpretations such that $(w, \{y \mapsto n_2\})$ belongs to $\mathcal{L}(Q_b(y))$. Now, given the automaton $A_{Q_a(x)}$ recognizing $\mathcal{L}(Q_a(x))$

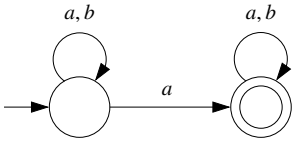


  we transform it into an automaton A_1 recognizing L_1



After constructing A_2 similarly, take $A_\varphi = A_1 \cup A_2$.

- $\varphi = \exists x\ \psi$. Then $free(\varphi) = free(\psi) \setminus \{x\}$, and by induction hypothesis there is an automaton A_ψ s.t. $L(A_\psi) = L(\psi)$. Define $A_{\exists\, x\psi}$ as the result of the projection operation, where we project onto all variables but x. The operation simply corresponds to removing in each letter of each transition of A_σ the component for variable x. For example, the automaton $A_{\exists x\ Q_a(x)}$ is obtained by removing the second components in the automaton for $A_{Q_a(x)}$ shown above, yielding



Observe that the automaton for $\exists x\ \psi$ can be nondeterministic even if the one for ψ is deterministic, since the projection operation may map different letters into the same one.

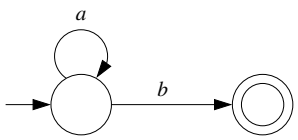
- $\varphi = \exists X\ \varphi$. We proceed as in the previous case.

Size of A_φ. The procedure for constructing A_φ proceeds bottom-up on the syntax tree of φ. We first construct automata for the atomic formulas in the leaves of the tree, and then proceed upwards: given automata for the children of a node in the tree, we construct an automaton for the node itself.

Whenever a node is labeled by a negation, the automaton for it can be exponentially bigger than the automaton for its only child. This yields an upper bound for the size of A_φ equal to a tower of exponentials, where the height of the tower is equal to the largest number of negations in any path from the root of the tree to one of its leaves.

It can be shown that this very large upper bound is essentially tight: there are formulas for which the smallest automaton recognizing the same language as the formula reaches the upper bound. This means that MSO-logic allows to describe some regular languages in an extremely *succinct* form.

Example 9.14 Consider the alphabet $\Sigma = \{a, b\}$ and the language $a^*b \subseteq \Sigma^*$, recognized by the NFA



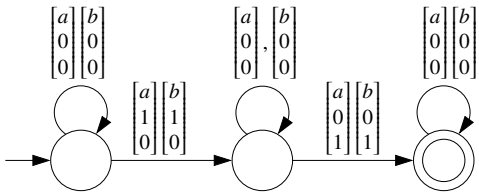
We derive this NFA by giving a formula φ such that $L(\varphi) = a^*b$, and then using the procedure described above. We shall see that the procedure is quite laborious. The formula states that the last letter is b, and all other letters are a's.

$$\varphi = \exists x\ (\text{last}(x) \wedge Q_b(x)) \ \wedge\ \forall x\ (\neg\text{last}(x) \rightarrow Q_a(x))$$

We first bring φ into the equivalent form

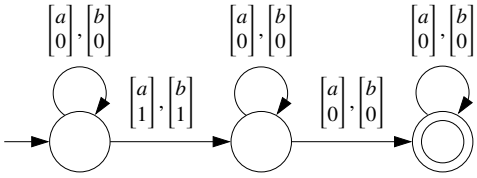
$$\psi = \exists x\ (\text{last}(x) \wedge Q_b(x)) \ \wedge\ \neg\exists x\ (\neg\text{last}(x) \wedge \neg Q_a(x))$$

We transform ψ into an NFA. First, we compute an automaton for $\text{last}(x) = \neg\exists y\ x < y$. Recall that the automaton for $x < y$ is



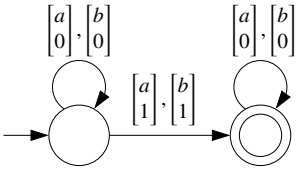
$[x < y]$

Applying the projection operation, we get following automaton for $\exists y\ x < y$

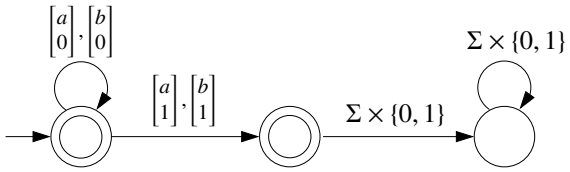


$[\exists y\ x < y]$

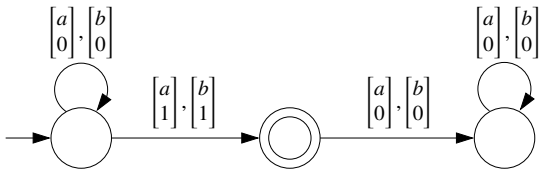
Recall that computing the automaton for the negation of a formula requires more than complementing the automaton. First,we need an automaton recognizing $Enc(\exists y\ x < y)$.



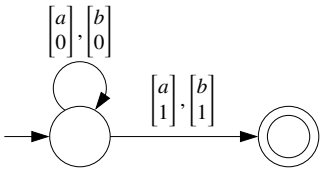
Second, we determinize and complement the automaton for $\exists y\ x < y$:



And finally, we compute the intersection of the last two automata, getting

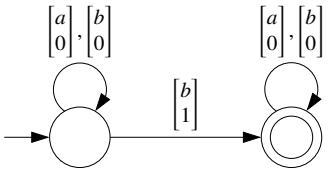


whose last state is useless and can be removed, yielding the following NFA for last(x):



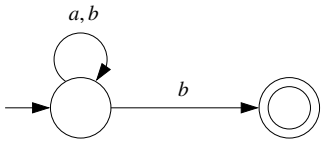
[last(x)]

Next we compute an automaton for $\exists x\ (\text{last}(x) \wedge Q_b(x))$, the first conjunct of ψ. We start with an NFA for $Q_b(x)$



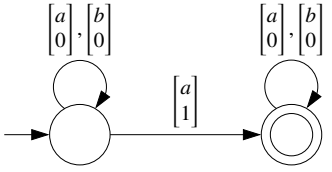
$[Q_b(x)]$

The automaton for $\exists x\ (\text{last}(x) \wedge Q_b(x))$ is the result of intersecting this automaton with the NFA for last(x) and projecting onto the first component. We get



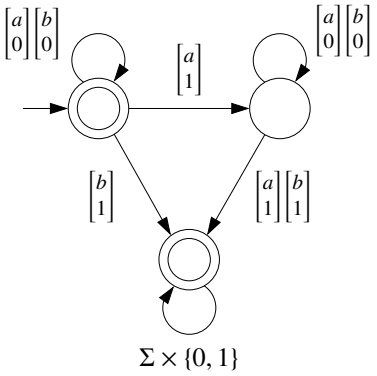
$[\exists x\ (\text{last}(x) \wedge Q_b(x))]$

Now we compute an automaton for $\neg\exists x\ (\neg\text{last}(x) \wedge \neg Q_a(x))$, the second conjunct of ψ. We first obtain an automaton for $\neg Q_a(x)$ by intersecting the complement of the automaton for $Q_a(x)$ and the automaton for $Enc(Q_a(x))$. The automaton for $Q_a(x)$ is

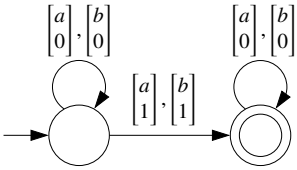


$[Q_a(x)]$

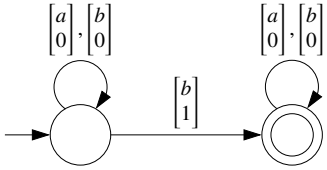
and after determinization and complementation we get



For the automaton recognizing $Enc(Q_a(x))$, notice that $Enc(Q_a(x)) = Enc(\exists y\ x < y)$, because both formulas have the same free variables, and so the same interpretations. But we have already computed an automaton for $Enc(\exists y\ x < y)$, namely



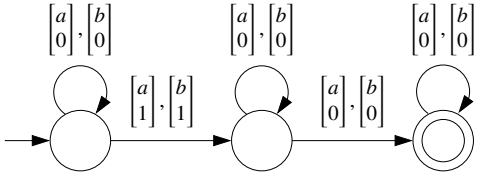
The intersection of the last two automata yields a three-state automaton for $\neg Q_a(x)$, but after eliminating a useless state we get



$[\neg Q_a(x)]$

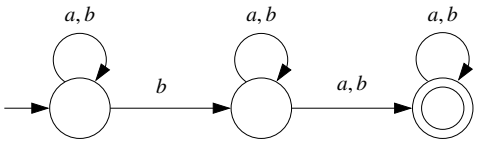
Notice that this is the same automaton we obtained for $Q_b(x)$, which is fine, because over the alphabet $\{a, b\}$ the formulas $Q_b(x)$ and $\neg Q_a(x)$ are equivalent.

To compute an automaton for $\neg last(x)$ we just observe that $\neg last(x)$ is equivalent to $\exists y\ x < y$, for which we have already compute an NFA, namely



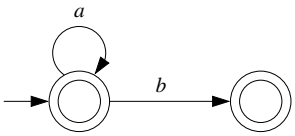
$[\neg last(x)]$

Intersecting the automata for $\neg last(x)$ and $\neg Q_a(x)$, and subsequently projecting onto the first component, we get an automaton for $\exists x\ (\neg last(x) \wedge \neg Q_a(x))$



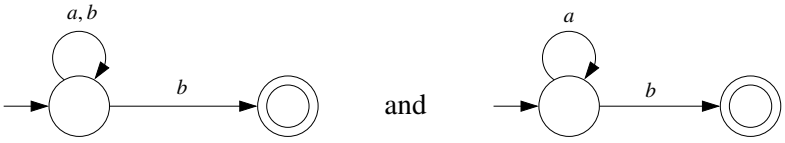
$[\exists x\ (\neg last(x) \wedge \neg Q_a(x))]$

Determinizing, complementing, and removing a useless state yields the following NFA for $\neg\exists x\ (\neg last(x) \wedge \neg Q_a(x))$:

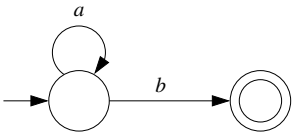


$[\neg\exists x\ (\neg last(x) \wedge \neg Q_a(x))]$

Summarizing, the automata for the two conjuncts of ψ are



whose intersection yields a 3-state automaton, which after removal of a useless state becomes



$[\exists x\ (\text{last}(x) \wedge Q_b(x)) \ \wedge\ \neg\exists x\ (\neg\text{last}(x) \wedge \neg Q_a(x))]$

ending the derivation. □

Exercises

Exercise 71 Characterize the languages described by the following formulas and give a corresponding automaton:

1. $\exists x\, first(x)$
2. $\forall x\, first(x)$
3. $\neg\exists x\exists y\ (x < y \wedge Q_a(x) \wedge Q_b(y)) \ \wedge\ \forall x\ (Q_b(x) \rightarrow \exists y\ x < y \wedge Q_a(y)) \ \wedge\ \exists x\ \neg\exists y\ (x < y \wedge Q_a(x))$

Exercise 72 Give a defining MSO-formula, an automaton, and a regular expression for the following languages over $\{a, b\}$.

- The set of words of even length and containing only a's or only b's.
- The set of words, where between each two b's with no other b in between there is a block of an odd number of letters a.
- The set of words with odd length and an odd number of occurrences of a.

Exercise 73 For every $n \geq 1$, give a FO-formula of polynomial length in n abbreviating $y = x+2^n$. (Notice that the abbreviation $y = x + k$ of page **??** has length $\mathcal{O}(k)$, and so it cannot be directly used.) Use it to give another FO-formula φ_n, also of of polynomial length in n, for the language $L_n = \{ww \in \{a, b\}^* \mid |w| = 2^n\}$.

Remark: Since the minimal DFA for L_n has 2^{2^n} states (Exercise 10), this shows that the number of states of a minimal automaton equivalent to a given FO-formula may be double exponential in the length of the formula.

Exercise 74 *MSO* over a unary alphabet can be used to automatically prove some simple properties of the natural numbers. Consider for instance the following property: every finite set of natural numbers has a minimal element[2]. It is easy to see that this property holds iff the formula

$$\forall Z \exists x \forall y \, (y \in Z \rightarrow (x \leq y \wedge x \in Z))$$

is a *tautology*, i.e., if it is satisfied by every word. Construct an automaton for the formula, and check that it is universal.

Exercise 75 Give formulas $\varphi_1, \ldots, \varphi_4$ for the following abbreviations:

$Sing(X)$	:=	φ_1	"X is a singleton, i.e., X contains one element"
$X \subseteq Y$	:=	φ_2	"X is a subset of Y"
$X \subseteq Q_a$	:=	φ_3	"every position of X contains an a"
$X < Y$	:=	φ_4	"X and Y are singletons $X = \{x\}$ and $Y = \{y\}$ satisfying $x < y$"

Exercise 76 Express addition in $MSO(\{a\})$. More precisely, find a formula $+(X, Y, Z)$ of $MSO(\{a\})$ that is true iff $x + y = z$, where x, y, z are the numbers encoded by the sets X, Y, Z, respectively, in *lsbf* encoding. You may use any abbrevation defined in the chapter.

[2]Of course, this also holds for every infinite set, but we cannot prove it using MSO over finite words.

Exercise 77 The *nesting depth* $d(\varphi)$ of a formula φ of $FO(\{a\})$ is defined inductively as follows:

- $d(Q_a(x)) = d(x < y) = 0$;
- $d(\neg\varphi) = d(\varphi)$, $d(\varphi_1 \vee \varphi_2) = \max\{d(\varphi_1), d(\varphi_2)\}$; and
- $d(\exists x\, \varphi) = 1 + d(\varphi)$.

Prove that every formula φ of $FO(\{a\})$ of nesting depth n is equivalent to a formula f of QF having the same free variables as φ, and such that every constant k appearing in f satisfies $k \geq 2^n$.

Hint: The proof is similar to that of Theorem 9.8. The difficult case is the one where φ has the form $\exists x\, \psi$ and ψ is a conjunction. Define f as the following conjunction. All conjuncts of ψ not containing x are also conjuncts of f; for every conjunct of D of the form $x \geq k$ or $x \geq y + k$, f contains a conjunct $last \geq k$; for every two conjuncts of D containing x, f contains a conjunct obtained by "quantifying x away": for example, if the conjuncts are $x \geq k_1$ and $y \geq x + k_2$, then f has the conjunct $y \geq k_1 + k_2$. Since the constants in the new conjuncts are the sum of two old constants, the new constants are bounded by $2 \cdot 2^d = 2^{d+1}$.

Chapter 10

Applications III: Presburger Arithmetic

Presburger arithmetic is a logical language to formulate properties of numbers that are expressible using addition and comparison. A typical example of such a property is "$x + 2y > z$ and $2x - 3z = 4y$". This property is satisfied by some valuations (n_x, n_y, n_z) of the triple of variables (x, y, z), like $(4, 2, 0)$, but not by others, like $(1, 1, 4)$. We call the former the *solutions* of the formula. We show how to construct for a given formula ϕ of Presburger arithmetic an NFA A_ϕ recognizing the solutions of ϕ.

10.1 Syntax and Semantics

Formulas of Presburger arithmetic are constructed out of an infinite set of *variables* $V = \{x, y, z, \ldots\}$ and the constants 0 and 1. The syntax of formulas is defined in three steps. First, the set of *terms* is inductively defined as follows:

- the symbols 0 and 1 are terms;
- every variable is a term;
- if t and u are terms, then $t + u$ is a term.

An *atomic formula* is an expression $t \leq u$, where t and u are terms. The set of Presburger formulas is inductively defined as follows:

- every atomic formula is a formula;
- if φ_1, φ_2 are formulas, then so are $\neg\varphi_1$, $\varphi_1 \vee \varphi_2$, and $\exists x\, \varphi_1$.

As usual, variables within the scope of an existential quantifier are bounded, and otherwise free. Besides the usual abbreviations like $\forall$, $\wedge$, $\rightarrow$, we also introduce:

$$\begin{array}{rcl@{\qquad}rcl} n & := & \underbrace{1+1+\ldots+1}_{n \text{ times}} & t \geq t' & := & t' \leq t \\ & & & t = t' & := & t \leq t' \wedge t \geq t' \\ nx & := & \underbrace{x+x+\ldots+x}_{n \text{ times}} & t < t' & := & t \leq t' \wedge \neg(t = t') \\ & & & t > t' & := & t' < t \end{array}$$

An *interpretation* is a function $\mathcal{I}\colon V \to \mathbb{N}$. An interpretation $\mathcal{I}$ is extended to terms in the natural way: $\mathcal{I}(0) = 0$, $\mathcal{I}(1) = 1$, and $\mathcal{I}(t+u) = \mathcal{I}(t) + \mathcal{I}(u)$. The satisfaction relation $\mathcal{I} \models \varphi$ is defined as one would expect, where $\mathcal{I}(n/x)$ denotes the interpretation that assigns the number n to the variable x, and the same numbers as $\mathcal{I}$ to all other variables:

$$\begin{array}{lll} \mathcal{I} \models t \leq u & \text{iff} & \mathcal{I}(t) \leq \mathcal{I}(u) \\ \mathcal{I} \models \neg\varphi_1 & \text{iff} & \mathcal{I} \not\models \varphi_1 \\ \mathcal{I} \models \varphi_1 \vee \varphi_2 & \text{iff} & \mathcal{I} \models \varphi_1 \text{ or } \mathcal{I} \models \varphi_2 \\ \mathcal{I} \models \exists x\, \varphi & \text{iff} & \text{there exists } n \geq 0 \text{ such that } I[n/x] \models \varphi \end{array}$$

Clearly, whether an interpretation satisfies a formula depends only on the values assigned by the interpretation assigns to the variables of φ. It is easy to see that, even further, it only depends on the values assigned to the *free* variables of φ (that is, if two interpretations assign the the same values to the free variables, then either both satisfy the formula, or none does). For a formula φ with k free variables, the set of all satisfying interpretations of its free variables constitutes a subset of $\mathbb{N}^k$, or, equivalently, a relation over the universe $\mathbb{N}$ of arity k. (where we assume a fix order on the free variables of the formula). It is also called the *solution space* or the set of *models* of φ, and we denote it by $Sol(\varphi)$.

Example 10.1 The solution space of the formula $x - 2 \geq 0$ is the set $\{2, 3, 4, \ldots\}$. The solution space of $\exists x\,(2x = y \wedge 2y = z)$ is the set of pairs $\{(2n, 4n) \mid n \geq 0\}$ (where we assume that the first and second components of the pairs are the values of y and z, respectively). □

Automata encoding natural numbers. Transducers can be used as data structure for computing and manipulating solution spaces. As in Section 5.1 of Chapter 5, we encode natural numbers as strings over $\{0, 1\}$ using the least-significant-bit-first encoding *lsbf*. If we have free variables $x_1, \ldots, x_k$, the elements of the solution space are encoded as a word over $\{0, 1\}^k$. For instance, the word

$$\begin{array}{cc} x_1 & \begin{bmatrix}1\\0\\0\end{bmatrix}\begin{bmatrix}0\\1\\0\end{bmatrix}\begin{bmatrix}1\\0\\0\end{bmatrix}\begin{bmatrix}0\\1\\0\end{bmatrix} \\ x_2 & \\ x_3 & \end{array}$$

is an encoding of the solution $(3, 10, 0)$. The language of a formula is then defined to be

$$L(\varphi) = \bigcup_{s \in Sol(\varphi)} lsbf(s)$$

where *lsbf*(s) denotes the set of all encodings of s. In other words, $L(\varphi)$ is the encoding of the relation $Sol(\varphi)$.

10.2 Constructing an NFA for the Solution Space.

Given a Presburger formula φ, we construct a transducer A such that $L(A) = L(\varphi)$. For this, use the implementations of operations on relations defined in Chapter 5.

Recall that ifφ has k free variables, then $Sol(\varphi)$ is a relation over $\mathbb{N}$ of arity k. The last section of Chapter 5 discusses how to generalize the implementations of operations for binary relations to relations of arbitrary arity. These operations can be used to compute the solution space of the negation of a formula, the disjunction of two formulas, and the existential quantification of two formulas.

- The solution space of the negation of a formula with k free variables is the complement of its solution space with respect to the universe U^k. In general, when computing the complement of a relation we have to worry about ensuring that the NFAs we obtain only accept words that encode some tuple of elements (i.e., some clean-up maybe necessary to ensure that the automata do not accept ‘ rubbish”, meaning words encoding nothing). In the case of Presburger arithmetic this is not necessary, because the *lsbf* encoding has the property that *every* word is the encoding of some tuple of numbers.

- The solution space of a disjunction $\varphi_1 \vee \varphi_2$ where φ_1 and φ_2 have the same free variables is clearly the union of their solution spaces, and can be computed as **Union**$(Sol(\varphi_1), Sol(\varphi_2))$. If φ_1 and φ_2 have different sets V_1 and V_2 of free variables, then some preprocessing is necessary. Define $Sol_{V_1 \cup V_2}(\varphi_i)$ as the set of valuations of $V_1 \cup V_2$ whose projection onto V_1 belongs to $Sol(\varphi_i)$. Transducers recognizing $Sol_{V_1 \cup V_2}(\varphi_i)$ for $i = 1, 2$ are easy to compute from transducers recognizing $Sol(\varphi_i)$. The solution space of φ is then given by **Union**$(Sol_{V_1 \cup V_2}(\varphi_1), Sol_{V_1 \cup V_2}(\varphi_2))$.

- The solution space of a formula $\exists x\ \varphi$, where x is a free variable of φ, is given by **Projection_I**$(Sol(\varphi))$, where I contains the indices of all variables with the exception of the index of x.

It only remains to show how to construct a DFA recognizing the solution space of atomic formulas. Consider an expression of the form

$$\varphi = \quad a_1 x_1 + \ldots + a_n x_n \leq b$$

where $a_1, \ldots, a_n, b \in \mathbb{Z}$ (not $\mathbb{N}$!). We let $a = (a_1, \ldots, a_n)$, $x = (x_1, \ldots, x_n)$, and denote by $a \cdot x$ the scalar product of a and x. So we have $\varphi = \quad a \cdot x \leq b$. Since we allow negative integers as coefficients, for every atomic formula there is an equivalent expression in this form (i.e., an expression with the same solution space). For example, $x \geq y + 4$ is equivalent to $-x + y \leq -4$.

The DFA must accept the encodings of all the the tuples $c \in \mathbb{N}^n$ satisfying $a \cdot c \leq b$. An encoding of a tuple $c = (c_1, c_2, \ldots, c_n) \in \mathbb{N}^n$ is a word $\zeta_0 \ldots \zeta_m$ over $\{0, 1\}^n$. We denote the j-th bit in the encoding of c_i by ζ_{ij}:

$$\begin{array}{cccccc} & \zeta_0 & \zeta_1 & \cdots & \zeta_m \\ c_1 & \begin{bmatrix}\zeta_{10}\end{bmatrix} & \begin{bmatrix}\zeta_{11}\end{bmatrix} & \cdots & \begin{bmatrix}\zeta_{1m}\end{bmatrix} \\ c_2 & \begin{bmatrix}\zeta_{20}\end{bmatrix} & \begin{bmatrix}\zeta_{21}\end{bmatrix} & \cdots & \begin{bmatrix}\zeta_{2m}\end{bmatrix} \\ \cdots & \begin{bmatrix}\cdots\end{bmatrix} & \begin{bmatrix}\cdots\end{bmatrix} & \cdots & \begin{bmatrix}\cdots\end{bmatrix} \\ c_n & \begin{bmatrix}\zeta_{n0}\end{bmatrix} & \begin{bmatrix}\zeta_{n1}\end{bmatrix} & \cdots & \begin{bmatrix}\zeta_{nm}\end{bmatrix} \end{array}$$

For every $1 \leq j \leq m+1$, let $c^j \in \mathbb{N}^n$ denote the tuple of numbers encoded by the prefix $\zeta_0 \dots \zeta_{j-1}$, and let c^0 denote the tuple encoded by ϵ, i.e., $c^0 = (0,0,0,0)$. For instance, for the encoding $\zeta_0\zeta_1\zeta_2$ of the tuple $(0,4,7,3)$ given by

$$\begin{array}{cccc} & \zeta_0 & \zeta_1 & \zeta_2 \\ 0 & \begin{bmatrix}0\\0\\1\\1\end{bmatrix} & \begin{bmatrix}0\\0\\1\\1\end{bmatrix} & \begin{bmatrix}0\\1\\1\\0\end{bmatrix} \\ 4 & & & \\ 7 & & & \\ 3 & & & \end{array} \qquad \text{we get} \qquad \begin{array}{cccc} & & \zeta_0 & \zeta_1 \\ 0 & & \begin{bmatrix}0\\0\\1\\1\end{bmatrix} & \begin{bmatrix}0\\0\\1\\1\end{bmatrix} \\ 0 & & & \\ 3 & & & \\ 3 & & & \end{array}$$

and so $c^2 = (0,0,3,3)$.

We construct a DFA for the solution space of φ. The idea is that after reading a prefix $\zeta_0 \dots \zeta_{j-1}$ the automaton should be in the state

$$\left\lfloor \frac{1}{2^j}\left(b - a \cdot c^j\right) \right\rfloor$$

So, intuitively, the state of the automaton after $\zeta_0 \dots \zeta_{j-1}$ is the difference between the 'ceiling' b that the tuple being read should not exceed in order to satisfy φ, and the current value $a \cdot c^j$, multiplied by a 'normalization' factor $1/2^j$. The final states are the nonnegative numbers, because they indicate that the tuple does not exceed the ceiling.

Initially we have $c^0 = (0, \dots, 0)$, and so the initial state is the number $\frac{1}{2^0}(b - a \cdot c^0) = b$. For the transitions, assume that before and after reading the letter ζ_j the automaton is in the states q and q', respectively. Then we have

$$q = \left\lfloor \frac{1}{2^j}\left(b - a \cdot c^j\right) \right\rfloor \qquad \text{and} \qquad q' = \left\lfloor \frac{1}{2^{j+1}}\left(b - a \cdot c^{j+1}\right) \right\rfloor$$

Using the fact that, by definition

$$c^j = \left(\sum_{i=0}^{j-1} 2^i \zeta_{1i}\, , \dots ,\ \sum_{i=0}^{j-1} 2^i \zeta_{ni} \right)$$

we get

$$c^{j+1} = c^j + 2^j \zeta_j$$

Inserting this in the expression for q', and comparing with q, we obtain

$$q' = \left\lfloor \frac{1}{2}(q - a \cdot \zeta_j) \right\rfloor$$

and so for every state q and every letter $\zeta \in \{0,1\}^n$ we define

$$\delta(q, \zeta) = \frac{1}{2}(q - a \cdot \zeta) \ .$$

This leads to the algorithm *PAtoDFA*(φ) of Table 10.1, where for clarity the state corresponding to an integer $k \in \mathbb{Z}$ is denoted by s_k.

PAtoDFA(φ)
Input: PA formula $\varphi = a \cdot x \leq b$
Output: DFA $A = (Q, \Sigma, \delta, q_0, F)$ such that $L(A) = L(\varphi)$

```
1  q_0 ← s_b
2  W ← {s_b}
3  while W ≠ ∅ do
4     pick s_k from W
5     add s_k to Q
6     if k ≥ 0 then add s_k to F
7     for all ζ ∈ {0,1}^n do
8        j ← ⌊(1/2)(k − a · ζ)⌋
9        if s_j ∉ Q then add s_j to W
10       add (s_k, ζ, s_j) to δ
```

Table 10.1: Converting an inequality into a DFA accepting the least-significant bit encoding of the solution space.

Example 10.2 Consider the formula $2x - y \leq 2$. The DFA obtained by applying *PAtoDFA*() to it is shown in Figure 10.1. The initial state is 2. Taking the $(0,0)$ transition leads to the state $\lfloor (2 - (2 \cdot 0 - 0))/2 \rfloor = 1$. Taking the $(1,1)$ transition leads to $\lfloor (2 - (2 \cdot 1 - 1))/2 \rfloor = 0$. States 2, 1 , and 0 are final. The DFA accepts, for example, the word

$$\begin{bmatrix}0\\0\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}$$

which corresponds to $x = 12$ and $y = 50$ and, indeed $24 - 50 \leq 2$. If we remove the last letter then the word corresponds to $x = 12$ and $y = 18$, is not accepted, and indeed $24 - 18 \not\leq 2$.

Consider now the formula $x + y \geq 4$. We rewrite it as $-x - y \leq -4$, and apply the algorithm. The resulting DFA is shown in Figure 10.2. The initial state is -4. Taking the transition $(1,1)$ leads

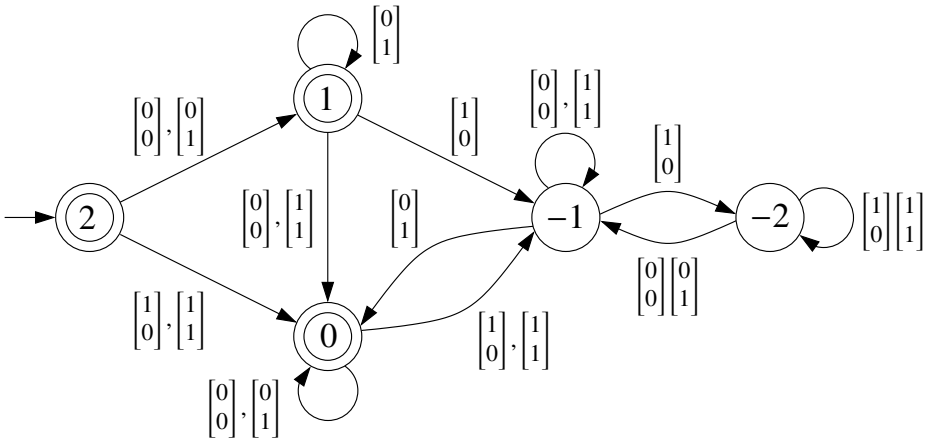


Figure 10.1: DFA for the formula $2x - y \leq 2$.

to the state $\lfloor(-4-(-1-1))/2\rfloor = -1$. Taking the $(0,1)$ transition leads to $\lfloor(-4-(0-1))/2\rfloor = -2$. Notice that the DFA is not minimal. In particular, states 0 and 1 can be merged without changing the language.

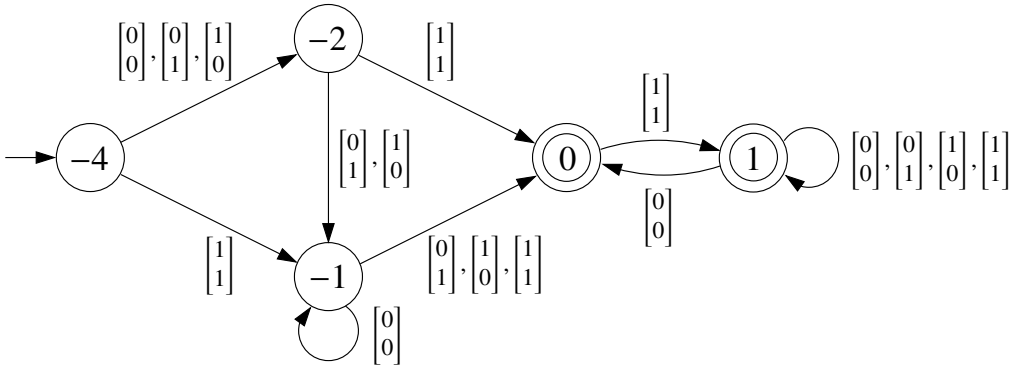


Figure 10.2: DFA for the formula $x + y \geq 4$.

□

The partial correctness of *PAtoDFA* follows from the considerations above. But we have not yet shown that the algorithm always terminates: in principle it could keep generating new states forever. We show that this is not the case.

Lemma 10.3 *Let $\varphi = a \cdot x \leq b$ and $s = \sum_{i=1}^{k} |a_i|$. All states s_j added to the worklist during the execution of PAtoDFA(φ) satisfy*

$$-|b| - s \leq j \leq |b| + s.$$

Proof: The property holds for s_b, the first state added to the worklist. We show that if all the states added to the worklist so far satisfy the property, then so does the next one. Let s_j be this next state. Then there exists a state s_k in the worklist and $\zeta \in \{0,1\}^n$ such that $j = \lfloor \frac{1}{2}(k - a \cdot \zeta) \rfloor$. Since by assumption s_k satisfies the property we have

$$-|b| - s \leq k \leq |b| + s$$

and so

$$\left\lfloor \frac{-|b| - s - a \cdot \zeta}{2} \right\rfloor \leq j \leq \left\lfloor \frac{|b| + s - a \cdot \zeta}{2} \right\rfloor$$

Now we observe

$$\begin{array}{rcccl} -|b| - s & \leq & \dfrac{-|b| - 2s}{2} & \leq & \left\lfloor \dfrac{-|b| - s - a \cdot \zeta}{2} \right\rfloor \\ \left\lfloor \dfrac{|b| + s - a \cdot \zeta}{2} \right\rfloor & \leq & \dfrac{|b| + 2s}{2} & \leq & |b| + s \end{array}$$

which together with 10.2 yields

$$-|b| - s \leq j \leq |b| + s$$

and we are done. □

Example 10.4 We compute all the solutions of the system of linear equations and inequations

$$\begin{array}{rcrcl} 2x & - & y & \leq & 2 \\ x & + & y & \geq & 4 \end{array}$$

such that both x and y are multiples of 4. This corresponds to computing a DFA for the Presburger formula

$$\exists z\, x = 4z \ \wedge\ \exists w\, y = 4w \ \wedge\ 2x - y \leq 2 \ \wedge\ x + y \geq 4$$

The minimal DFA for the first two conjuncts can be computed using the algorithms of the chapter, but the result is also easy to guess: it is the DFA of Figure 10.3 (a trap state has been omitted).

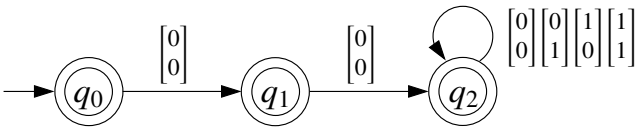


Figure 10.3: DFA for the formula $\exists z x = 4z \ \wedge\ \exists w\, y = 4w$.

The solutions are then represented by the intersection of the DFAs of Figures 10.1, 10.2 (after merging states 0 and 1), and 10.3. The resulting DFA is shown in Figure 10.4. (The intersection operation actually produces some additional states from which no final state can be reached; these states have been omitted.)

□

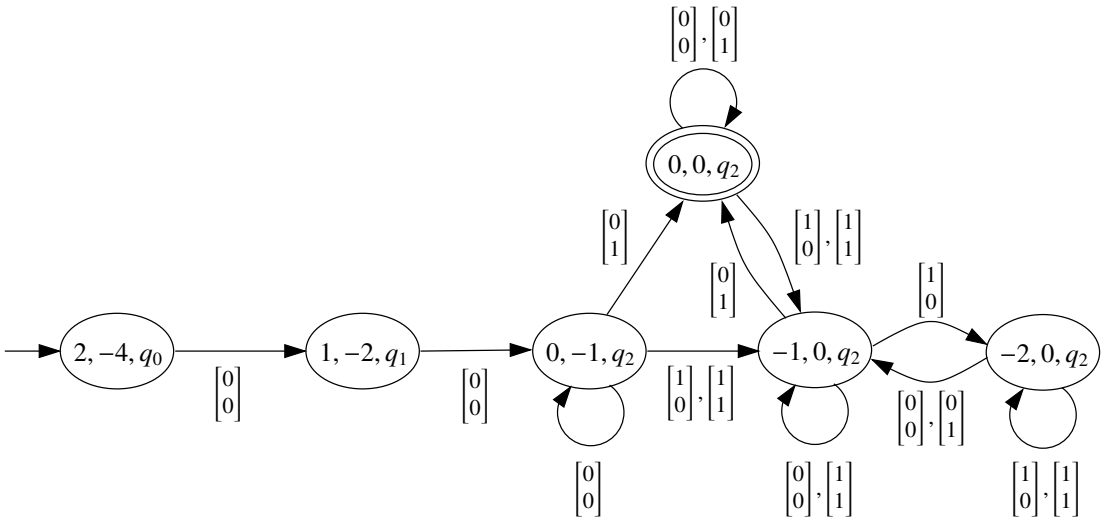


Figure 10.4: Intersection of the DFAs of Figures 10.1, 10.2, and 10.3. States from which no final state is reachable have been omitted.

Exercises

Exercise 78 Let $k_1, k_2 \in \mathbb{N}_0$ be constants. Find a Presburger arithmetic formula, $\varphi(x, y)$, with free variables x and y such that $\mathcal{I} \models \varphi(x, y)$ iff $\mathcal{I}(x) \geq \mathcal{I}(y)$ and $\mathcal{I}(x) - \mathcal{I}(y) \equiv k_1 \pmod{k_2}$. Find a corresponding automaton for the case $k_1 = 0$ and $k_2 = 2$.

Exercise 79 Using the algorithms discussed in the lecture, construct a finite automaton for the Presburger formula $\exists y\ x = 3y$.

Exercise 80 *PAtoDFA* returns a DFA recognizing all solutions of a given linear inequation

$$a_1 x_1 + a_2 x_2 + \ldots + a_k x_k \leq b \text{ with } a_1, a_2, \ldots, a_k, b \in \mathbb{Z} \qquad (*)$$

encoded using the *lsbf* encoding of $\mathbb{N}^k$. We may also use the most-significant-bit-first (*msbf*) encoding, e.g.,

$$\text{msbf}\left(\begin{bmatrix} 2 \\ 3 \end{bmatrix}\right) = \mathcal{L}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}^* \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix}\right)$$

1. Construct a finite automaton for the inequation $2x - y \leq 2$ w.r.t. *msbf* encoding.
2. Adapt *PAtoDFA* to the msbf encoding.
3. Recall that integers can be encoded as binary strings using two's complement: a binary string $s = b_0 b_1 b_2 \ldots b_n$ is interpreted in *msbf* encoding as the integer

$$-b_0 \cdot 2^n + b_1 \cdot 2^{n-1} + b_2 \cdot 2^{n-2} + \ldots + b_n \cdot 2^0.$$

In particular, s and $(b_0)^*s$ represent the same integer. This extends in the standard way to tuples of integers, e.g., the pair $(-3, 5)$ has the following encodings:

$$\left[\begin{array}{c}1\\0\end{array}\right]^*\left[\begin{array}{c}1\\0\end{array}\right]\left[\begin{array}{c}1\\1\end{array}\right]\left[\begin{array}{c}0\\0\end{array}\right]\left[\begin{array}{c}1\\1\end{array}\right]$$

- Construct an automaton accepting all (encodings of) *integer* solutions of the inequation $2x - y \leq 2$.
- Modify the algorithm from Exercise 80 so that it returns a DFA recognizing all two's complement encodings of all integer solutions of $(*)$.

Part II

Automata on Infinite Words

Chapter 11

Classes of ω-Automata and Conversions

In Section 11.1 we introduce ω-regular expressions, a textual notation for defining languages of infinite words. The other sections introduce different classes of automata on infinite words having the same expressive power as ω-regular expressions, and conversion algorithms allowing to move between these classes.

11.1 ω-languages and ω-regular expressions

Let Σ be an alphabet. An *infinite* word, also called an *ω-word*, is an infinite sequence $a_0a_1a_2\ldots$ of letters of Σ. The concatenation of a finite word $w_1 = a_1\ldots a_n$ and an ω-word $w_2 = b_1b_2\ldots$ is the ω-word $w_1w_2 = a_1\ldots a_nb_1b_2\ldots$, sometimes also denoted by $w_1 \cdot w_2$. Notice that $\varepsilon \cdot w = w$. We denote by Σ^ω the set of all ω-words over Σ. A set $L \subseteq \Sigma^\omega$ of ω-words is an *infinitary language* or *ω-language* over Σ.

The *concatenation* of a language L_1 and an ω-language L_2 is $L_1 \cdot L_2 = \{w_1w_2 \in \Sigma^\omega \mid w_1 \in L_1 \text{ and } w_2 \in L_2\}$. The *$\omega$-iteration* of a language $L \subseteq \Sigma^*$ is the ω-language $L^\omega = \{w_1w_2w_3\ldots \mid w_i \in L \setminus \{\epsilon\}\}$. Observe that $\{\epsilon\}^\omega = \emptyset$, in contrast to the case of finite words, where $\{\epsilon\}^* = \{\epsilon\}$. Notice that $\{\epsilon\}^\omega = \{\epsilon\}$ does not make sense, because all the words of L^ω must have infinite length.

We now extend regular expressions to ω-regular expressions, a formalism to define ω-languages.

Definition 11.1 *ω-regular expressions s over an alphabet Σ are defined by the following grammar, where $r \in \mathcal{RE}(\Sigma)$ is a regular expression*

$$s ::= \emptyset \mid r^\omega \mid rs_1 \mid s_1 + s_2$$

Sometimes we write $r \cdot s_1$ instead f rs_1. The set of all ω-regular expressions over Σ is written $\mathcal{RE}_\omega(\Sigma)$. The language *$L_\omega(s) \subseteq \Sigma$ of an ω-regular expression $s \in \mathcal{RE}_\omega(\Sigma)$ is defined inductively as*

- $L_\omega(\emptyset) = \emptyset$;
- $L_\omega(r^\omega) = (L(r))^\omega$;

- $L_\omega(rs_1) = L(r) \cdot L_\omega(s_1)$*; and*
- $L_\omega(s_1 + s_2) = L_\omega(s_1) \cup L_\omega(s_2)$.

A language L is ω-regular *if there is an ω-regular expression s such that $L = L_\omega(s)$.*

11.2 Büchi automata

Büchi automata have the same syntax as NFAs, but a different definition of acceptance. Suppose that an NFA $A = (Q, \Sigma, \delta, q_0, F)$ is given as input an infinite word $w = a_0a_1a_2\ldots$ over Σ. Intuitively, a run of A on w never terminates, and so we cannot define acceptance by looking at the state reached at the end of the run. Instead we consider the *limit* behavior of the run. We say that the run is accepting if some accepting state is visited along the run *infinitely often*. If we picture the automaton as having a light that is switched on precisely whenever the automaton visits an accepting state, then the run is accepting if the light is switched on infinitely many times.

Observe that the name "final state" is no longer appropriate for Büchi automata, because the run never ends. So from now on we speak of "accepting states". However, we still use F to denote the set of accepting states.

Definition 11.2 *A* nondeterministic Büchi automaton *(NBA) is a tuple $A = (Q, \Sigma, \delta, q_0, F)$, where Q, Σ, δ, q_0, and F are defined as for NFAs. A* run *of A on an ω-word $a_0a_1a_2\ldots$ is an infinite sequence $\rho = p_0 \xrightarrow{a_0} p_1 \xrightarrow{a_1} p_2 \ldots$, such that $p_i \in Q$ for $0 \le i \le n$ and $p_{i+1} \in \delta(p_i, a_i)$ for every $0 \le i$. Let $\inf(\rho)$ be the set $\{q \in Q \mid q = p_i$ for infinitely many i's$\}$, i.e., the set of states that occur in ρ infinitely often. The run ρ is* accepting *if there is some accepting state that repeats in ρ infinitely often, i.e., if $\inf(\rho) \cap F \neq \emptyset$. An NBA* accepts *an ω-word $w \in \Sigma^\omega$ if it has an accepting run on w. The* language recognized *by and NBA A is the set $L_\omega(A) = \{w \in \Sigma^\omega \mid w$ is accepted by $A\}$.*

A set F of accepting states induces an acceptance condition on runs: a run ρ is accepting iff the condition $\inf(\rho) \cap F \neq \emptyset$ holds. Abusing language, we often refer to this condition as *the Büchi condition F*.

NBAs are the extension to ω-words of NFAs. Deterministic Büchi automata (DBAs) are defined as for finite words, that is, a Büchi automaton is deterministic if it is deterministic when seen as an automaton on finite words. It is also possible to define NBAs with ϵ-transitions, but we will not need them.[1]

Figure 11.1 shows two Büchi automata. The automaton on the left is deterministic, and recognizes all ω-words over the alphabet $\{a, b\}$ that contain infinitely many occurrences of a. So, for instance, A accepts a^ω, ba^ω, $(ab)^\omega$, or $(ab^{100})^\omega$, but not b^ω or $a^{100}b^\omega$. The automaton on the right is not deterministic, and recognizes the complement of this language, i.e., all ω-words over the alphabet $\{a, b\}$ that contain *finitely many* occurrences of a.

[1] Notice that the definition of NBA-ϵ requires some care, because infinite runs containing only finitely many non-ϵ transitions are never accepting, even if they visit some accepting state infinitely often.

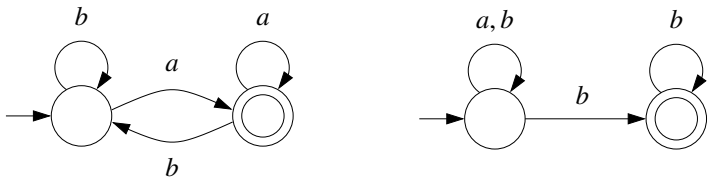


Figure 11.1: Two Büchi automata

Let A be the automaton on the left. To prove that A recognizes the words containing infinitely many occurrences of a, we have to show that every ω-word containing infinitely many as is accepted by A, and that every word accepted by A contains infinitely many as. For the first part, observe that immediately after reading any a, the automaton A always visits its (only) accepting state (because all transitions labeled by a lead to it); therefore, when A reads an ω-word containing infinitely many as it visits its accepting state infinitely often, and so it accepts. For the second part, let w be an ω-word accepted by A. Then there is a run of A on w that visits the accepting state infinitely often. Since all transitions leading to the accepting state are labeled by a, the automaton must read infinitely many as during the run. So w contains infinitely many as, and we are done. The proof that the automaton on the right recognizes the ω-words over the alphabet $\{a, b\}$ containing finitely many occurrences of a is similar.

Figure 11.2 shows three further Büchi automata over the alphabet $\{a, b, c\}$. The top left one recognizes the ω-words in which for every occurrence of a there is a later occurrence of b. So, for instance, the automaton accepts $(ab)^\omega$, c^ω, or $(bc)^\omega$, but not ac^ω or $ab(ac)^\omega$. The top right automaton recognizes the ω-words that contain finitely many occurrences of a, or infinitely many occurrences of a *and* infinitely many occurrences of b. Finally, the automaton at the bottom recognizes the ω-words in which between every occurrence of a and the next occurrence of c there is *at most one* occurrence of b; more precisely, for every two numbers $i < j$, if the letter at position i is an a and the first occurrence of c after i is at position j, then there is at most one number $i < k < j$ such that the letter at position k is a b.

11.2.1 From ω-regular expressions to NBAs and back

We present algorithms for converting an ω-regular expression into a Büchi automaton, and vice versa. This provides a first sanity check for NBAs as data structure: they can represent exactly the ω-regular languages.

From ω-regular expressions to NBAs. Recall that the result of exhaustively applying the preprocessing rules of page 21 to a given regular expression r is another regular expression r' such that $L(r) = L(r')$ and either $r' = \emptyset$ or r' does not contain any occurrence of $\emptyset$. We add the following rules, where r is a regular expression and s an ω-regular expression:

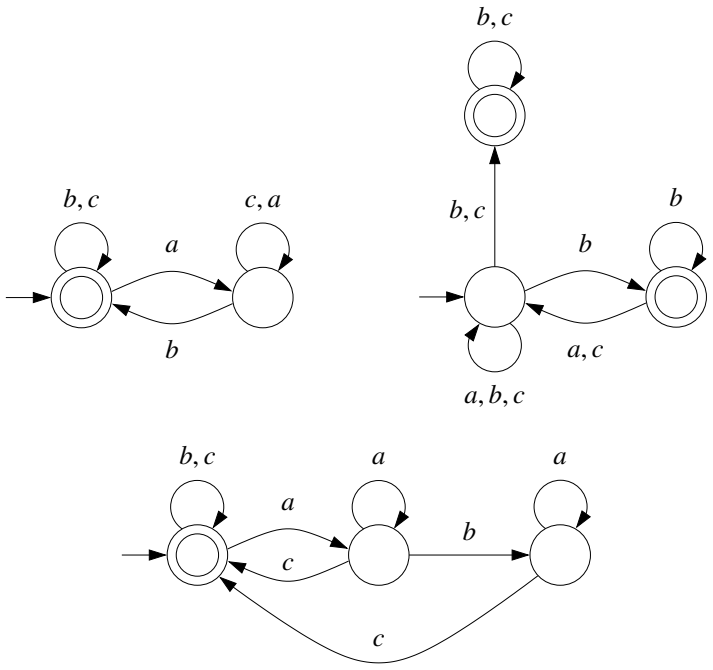
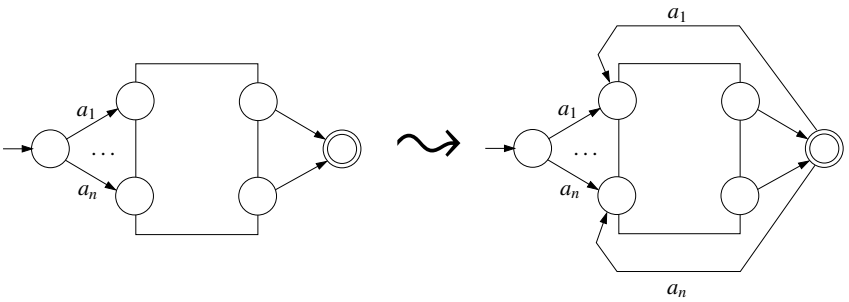


Figure 11.2: Three further Büchi automata

$$\begin{array}{lcl} \emptyset^\omega & \rightsquigarrow & \emptyset \\ \emptyset \cdot s & \rightsquigarrow & \emptyset \\ \emptyset + s & \rightsquigarrow & s \end{array} \qquad \begin{array}{lcl} r \cdot \emptyset & \rightsquigarrow & \emptyset \\ s + \emptyset & \rightsquigarrow & s \end{array}$$

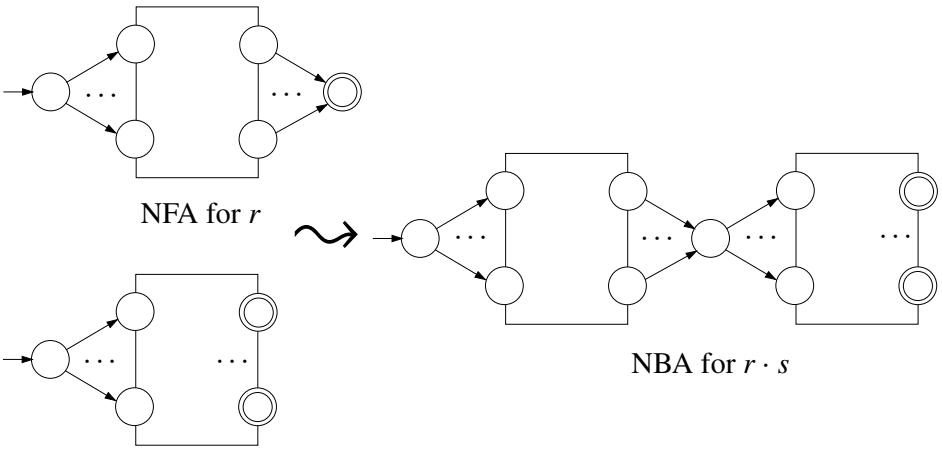
The result of exhaustively applying these rules to a given ω-regular expression s is another ωregular expression s' such that $L_\omega(s) = L_\omega(s')$ and either $s' = \emptyset$ or s' does not contain any occurrence of $\emptyset$. In the first case we can directly obtain a Büchi automaton. In the second case, we inductively construct an NBA. Recall that for every regular expression with no occurrences of $\emptyset$ we can construct an NFA-ϵ with a unique final state, and such that no transition leads to the initial state, and no transition starts from the final state. Moreover, using *NFA-εtoNFA*, this NFA-ϵ is transformed into an NFA satisfying the same property. Now we proceed as follows.

An NBA for r^ω is obtained by adding to the NFA for r transitions leading from the final state to the targets of the transitions leaving the initial state:



NFA for r NBA for r^ω

An NBA for $r \cdot s$ is obtained by merging the final state of the NFA for r and the initial state of the NBA for s as follows:

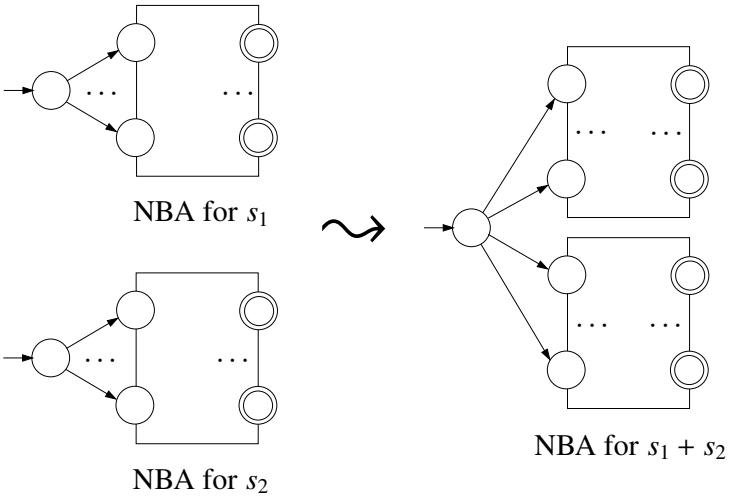


NFA for r

NBA for s

NBA for $r \cdot s$

Finally, an NBA for $s_1 + s_2$ is obtained by merging the initial states of the NBAs for s_1 and s_2:



From NBAs to ω-regular expressions. The fact that NBAs can be converted into ω-regular expressions is an easy consequence of the fact that NFAs can be converted into regular expressions.

Let $A = (Q, \Sigma, \delta, q_0, F)$ be a NBA. For every two states $q, q' \in Q$, let $A_q^{q'} = (Q, \Sigma, \delta, q, \{q'\})$ be the NFA (not the NBA!) obtained from A by changing the initial state to q and the final state to q'. Using algorithm *NFAtoRE* we can construct a regular expression $r_q^{q'}$ such that $L(A_q^{q'}) = L(r_q^{q'})$.

We use these regular expressions to find an ω-regular expression for $L_\omega(A)$. For every accepting state $q \in F$, let $L_q \subseteq L_\omega(A)$ be the set of ω-words w such that some run of A on w visits the state q infinitely often. We have $L_\omega(A) = \bigcup_{q \in F} L_q$.

Every word $w \in L_q$ can be split into an infinite sequence $w_1 w_2 w_3 \ldots$ of finite, nonempty words, where w_1 is the word read by the automaton until it visits q for the first time, and for every $i > 1$ w_i is the word read by the automaton between the i-th and the $(i+1)$-th visits to q. It follows $w_1 \in L(r_{q_0}^q)$, and $w_i \in L(r_q^q)$ for every $i > 1$. So we have $L_q = L_\omega(r_{q_0}^q \left(r_q^q\right)^\omega)$, and so

$$\sum_{q \in F} r_{q_0}^q \left(r_q^q\right)^\omega$$

is the ω-regular expression we are looking for.

Example 11.3 Consider the top right NBA of Figure 11.2, shown again in Figure 11.3. We have to compute $r_0^1 \left(r_1^1\right)^\omega + r_0^2 \left(r_2^2\right)^\omega$. Using *NFAtoRE* and simplifying we get

$$\begin{aligned} r_0^1 &= (a+b+c)^*(b+c)(b+c)^* \\ r_0^2 &= (a+b+c+bb^*(a+c))^* bb^* = (a+b+c)^* bb^* \\ r_1^1 &= (b+c)^* \\ r_2^2 &= (b+(a+c)(a+b+c)^* b)^* \end{aligned}$$

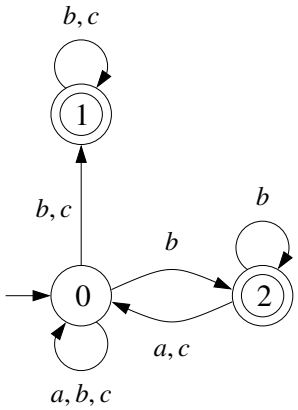


Figure 11.3: An NBA

and (after some further simplifications) we get the ω-regular expression

$$(a+b+c)^*(b+c)^\omega + (a+b+c)^*(b+(a+c)(a+b+c)^*b)^\omega$$

. □

11.2.2 Non-equivalence of NBA and DBA

Unfortunately, DBAs do not recognize all ω-regular languages, and so they do not have the same expressive power as NBAs.

Proposition 11.4 *The language $L = (a+b)^*b^\omega$, (i.e., L consists of all infinite words in which a occurs only finitely many times) is not recognized by any DBA.*

Proof: Assume by way of contradiction that $L = L_\omega(A)$, for some DBA $A = (\{a,b\}, Q, q_0, \delta, F)$. We extend δ to a mapping $Q \times \{a,b\}^* \to Q$ in the usual way: $\hat{\delta}(q,\epsilon) = q$ and $\hat{\delta}(q,wa) = \delta(\hat{\delta}(q,w),a)$.

Consider the infinite word $w_0 = b^\omega$. Clearly, w_0 is accepted by A, and so A has an accepting run on w_0. Thus, w_0 has a finite prefix u_0 such that $\hat{\delta}(q_0,u_0) \in F$. Consider now the infinite word $w_1 = u_0ab^\omega$. Clearly, w_1 is also accepted by A, and so A has an accepting run on w_1. Thus, w_1 has a finite prefix u_0au_1 such that $\hat{\delta}(q_0,u_0au_1) \in F$. In a similar fashion we can continue to find finite words u_i such that $\hat{\delta}(q_0,u_0au_1a\ldots au_i) \in F$. Since Q is finite, there are i, j, where $0 \le i < j$, such that $\delta(q_0,u_0au_1a\ldots au_i) = \delta(q_0,u_0au_1a\ldots au_ia\ldots au_j)$. It follows that A has an accepting run on

$$u_0au_1a\ldots au_i(au_{i+1}\ldots u_{j-1}au_j)^\omega.$$

But the latter word has infinitely many occurrences of a, so it does not belong to L. □

Note that the $\overline{L} = ((a+b)^*a)^\omega$ (the set of infinite words in which a occurs infinitely often) is accepted by the DBA on the left of Figure 11.1.

11.3 Generalized Büchi automata

Generalized Büchi automata are an extension of Büchi automata that is convenient for some constructions (for instance, intersection).

A *generalized Büchi automaton* (NGA) differs from a Büchi automaton in that it has a *collection of sets of accepting states* $\mathcal{F} = \{F_0, \ldots, F_{m-1}\}$, instead of only one set F. A run ρ is accepting if *for every set* $F_i \in \mathcal{F}$ some state of F_i is visited by ρ infinitely often. Formally, ρ is accepting if $\inf(\rho) \cap F_i \neq \emptyset$ for every $i \in \{0, \ldots, m-1\}$. Abusing language, we speak of the *generalized Büchi condition* $\mathcal{F}$. Ordinary Büchi automata correspond to the special case $m = 1$.

A NGA with n states and m sets of accepting states can be translated into an NBA with mn states. The construction is based on the following observation: a run ρ visits some state of F_i infinitely often for every $i \in \{0, \ldots, m-1\}$ if and only if the following two conditions hold:

(1) ρ eventually visits F_0; and

(2) for every $i \in \{0, \ldots, m-1\}$, every visit of ρ to F_i is eventually followed by a later visit to $F_{i \oplus 1}$, where $\oplus$ denotes addition modulo m. (Between the visits to F_i and $F_{i \oplus 1}$ there can be arbitrarily many visits to other sets.)

The proof is obvious: by (1) and (2), ρ visits each F_i infinitely often. Moreover, since each F_i is finite, some state of each F_i is visited infinitely often.

This observation suggests to take for the NBA m "copies" of the NGA. We move from the i-th to the $i \oplus 1$-th copy whenever we visit a state of F_i (i.e., we redirect the transitions of the i-th copy that leave a state of F_i to the next copy). This way, visiting the accepting states of the first copy infinitely often is equivalent to visiting the accepting states of *each* copy infinitely often.

The states of the NBA are pairs $[q, i]$ where q is a state of the NGA and $i \in \{0, \ldots, m-1\}$. Intuitively, $[q, i]$ is the i-th copy of q. The successors of $[q, i]$ are either states of the i-th copy, if $q \notin F_i$, or states of the $(i \oplus 1)$-th copy, if $q \in F_i$.

NGAtoNBA(A)
Input: NGA $A = (Q, \Sigma, q_0, \delta, \mathcal{F})$, where $\mathcal{F} = \{F_1, \ldots, F_m\}$
Output: NBA $A' = (Q', \Sigma, \delta', q_0', F')$

```
1  Q', δ', F' ← ∅
2  q_0' ← [q_0, 0]
3  W ← {[q_0, 0]}
4  while W ≠ ∅ do
5     pick [q, i] from W
6     add [q, i] to Q'
7     if q ∈ F_0 and i = 0 then add [q, i] to F'
8     for all a ∈ Σ do
9        for all q' ∈ δ(q, a) do
10          if q ∉ F_i then
11             if [q', i] ∉ Q' then add [q', i] to W
12             add ([q, i], a, [q', i]) to δ'
13          else /* q ∈ F_i */
14             if [q', i ⊕ 1] ∉ Q' then add [q', i ⊕ 1] to W
15             add ([q, i], a, [q', i ⊕ 1]) to δ'
16 return (Q', Σ, δ', q_0', F')
```

Example 11.5 Figure 11.4 shows a NGA over the alphabet $\{a, b\}$ on the left, and the NBA obtained by applying *NGAtoNBA* to it on the right. The NGA has two sets of accepting states, $F_0 = \{q\}$ and $F_1 = \{r\}$. So the accepting runs are those visiting *both* q and r. It is easy to see that the automaton recognizes the ω-words containing infinitely many occurrences of a *and* infinitely many occurrences of b.

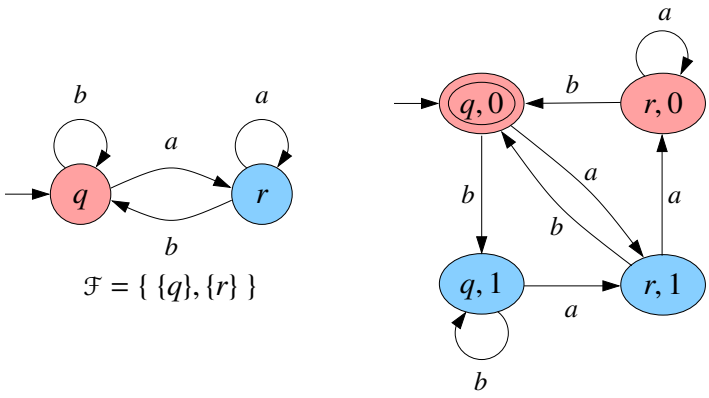


Figure 11.4: A NGA and its corresponding NBA

The NBA on the right consists of two copies of the NGA: the 0-th copy (pink) and the 1-st copy

(blue). The only accepting state is $[q, 0]$. □

11.4 Other classes of ω-automata

We have seen that not every NBA is equivalent to a DBA; i.e., there is determinization procedure for Büchi automata. This raises the question whether there exists other classes of automata for which a determinization procedure exists. As we shall see, the answer is yes, but the first determinizable classes we find will have other problems, and so this section can be seen as a **quest** for automata classes satisfying more and more properties.

11.4.1 Co-Büchi Automata

A *(nondeterministic) co-Büchi automaton* (NCA) has, like a B"uchi automaton, a set F of accepting states. However, a run ρ is now accepting if it does not visit any state of F infinitely often. Formally, ρ is accepting if $\inf(\rho) \cap F = \emptyset$. So a run of a NCA is accepting iff it is not accepting as run of a NBA (this is the reason for the name "co-Büchi").

We show that co-Büchi automata can be determinized. We fix an NCA $A = (Q, \Sigma, \delta, q_0, F)$ with n states, and use Figure 12.5 as running example. We construct a DCA B such that $L_\omega(B)) = L_\omega(A)$

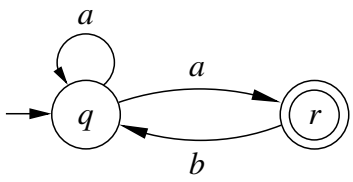


Figure 11.5: Running example for the determinization procedure

in three steps.

1. We define a mapping *dag* which assigns to each word $w \in \Sigma^\omega$ a directed acyclic graph $dag(w)$.

2. We prove that w is rejected by A if and only if $dag(w)$ contains only finitely many *checkpoints*.

3. We construct a DBA B which accepts w if and only if $dag(w)$ has finitely many checkpoints.

Intuitively, $dag(w)$ is the result of "bundling together" all the runs of A on the word w. Figure 11.6 shows the initial parts of $dag(aba^\omega)$ and $dag((ab)^\omega)$. Formally, let $w = \sigma_1\sigma_2\ldots$ be a word of Σ^ω. The directed acyclic graph $dag(w)$ has nodes in $Q \times \mathbb{N}$ and edges labelled by letters of Σ, and is inductively defined as follows:

- $dag(w)$ contains a node $\langle q, 0\rangle$ for every initial state $q \in Q_0$.

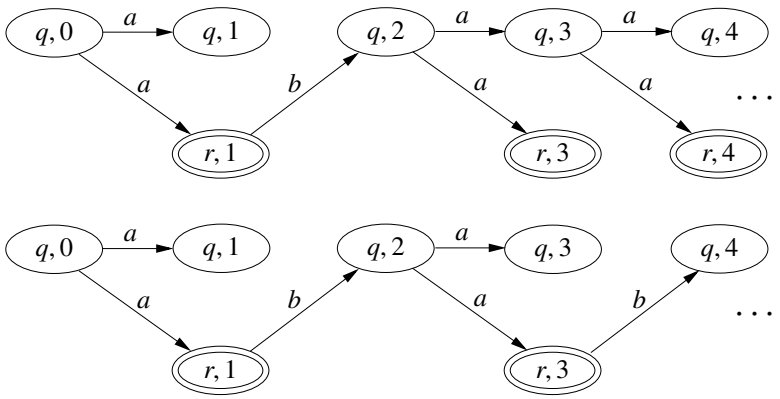


Figure 11.6: The (initial parts of) $dag(aba^\omega)$ and $dag((ab)^\omega)$

- If $dag(w)$ contains a node $\langle q, i\rangle$ and $q' \in \delta(\sigma_{i+1}, q)$, then $dag(w)$ also contains a node $\langle q', i+1\rangle$ and an edge from $\langle q, i\rangle \xrightarrow{\sigma_{i+1}} \langle q', i+1\rangle$.
- $dag(w)$ contains no other nodes or edges.

Clearly, $q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} q_2 \cdots$ is a run of A if and only if $\langle q_0, 0\rangle \xrightarrow{\sigma_1} \langle q_1, 1\rangle \xrightarrow{\sigma_2} \langle q_2, 2\rangle \cdots$ is a path of $dag(w)$. Moreover, A accepts w if and only if *no path* of $dag(w)$ visits accepting states infinitely often.

We partition the nodes of $dag(w)$ into *levels*: the i-th level contains all the nodes of $dag(w)$ of the form $\langle q, i\rangle$. One could be tempted to think that the accepting condition

> no path of $dag(w)$ visits accepting states infinitely often

is equivalent to

> only finitely many levels of $dag(w)$ contain accepting states

but $dag(aba^\omega)$ shows that this is not true: all levels $i \geq 3$ contain accepting states, but no path visits accepting states infinitely often. For this reason we introduce the *breakpoint set* of $dag(w)$. The breakpoint set is the set of levels of $dag(w)$ inductively defined as follows:

- The first level of $dag(w)$ is a breakpoint.
- If level l is a breakpoint, then the next level $l' > l$ such that every path between nodes of l and l' visits an accepting state is also a breakpoint.

We claim that

> no path of $dag(w)$ visits accepting states infinitely often

is equivalent to

the breakpoint set is finite.

If the breakpoint set is infinite, then clearly $dag(w)$ contains at least an infinite path, and all infinite paths visit accepting states infinitely often. If the breakpoint set is finite, let i be the largest breakpoint. If $dag(w)$ is fiite, we are done. If $dag(w)$ is infinite, then for every $j > i$ there is a path π_j from level i to level j that does not visit any accepting state. The paths $\{\pi_j\}_{j>i}$ build an acyclic graph of bounded degree, and so the graph has an infinite path that never visits any accepting state.

Now, if we were able to tell that a level is a checkpoint just by examining it, we would be done: we could take the posible levels as states of the DCA (as in the subset construction for the detrminization of NFAs), and the possible transitions between levels as transitions: and the checkpoints as accepting states. The run of this automaton on w would be nothing but an encoding of $dag(w)$, and it would be accepting iff it contains only finitely many checkpoints. However, it is easy to see that this is not possible. The solution consists of adding information to the states: we take for the states pairs $[P, O]$, where $P \subseteq Q$ and $O \subseteq P$ with the following intended meaning: P is the set of states of a level, and $q \in O$ if q is the endpoint of a path that starts at the last checkpoint and has not yet visited an accepting state. We call O the set of *owing* states (states that "owe" a visit to accepting states. To guarantee that O indeed has this meaning, we define the initial states and transitions as follows.

- The initial state is the pair $[\{q_0\}, \emptyset]$ if $q_0 \in F$, and $[\{q_0\}, \{q_0\}]$ otherwise.

The transition relation $\tilde{\delta}$ is defined as follows. Let $[P, O]$ be a state, and let $a \in \Sigma$. Then, $\tilde{\delta}([P, O], a) = [P', O']$, where $P' = \delta(P, a)$ (the set of states of the next level), and the new set O' of owing states is defined as follows:

- If $O \neq \emptyset$, i.e., if the current set of owing states is nonempty, then $O' = \delta(O, a) \setminus F$.

- If $O = \emptyset$, then the current level is a checkpoint, and then the automaton must start searching for the next one. For this, all non-final states in the next level become owing: $O' = \delta(P, a) \setminus F$.

The accepting states are those at which a checkpoint is reached:

- a state $[P, O]$ is accepting if $O = \emptyset$.

With this definition, a run is accepting if it contains infinitely many checkpoints, which means that every state at each level ranking eventually reaches a successor with odd rank.

NCAtoDCA(A)
Input: NCA $A = (Q, \Sigma, \delta, q_0, F)$
Output: DCA $B = (\tilde{Q}, \Sigma, \tilde{\delta}, \tilde{q}_0, \tilde{F})$ with $L_\omega(A) = \overline{B}$

```
1  Q̃, δ̃, F̃ ← ∅
2  if q0 ∈ F then q̃0 ← [q0, ∅] else q̃0 ← [{q0}, {q0}]
3  W ← { q̃0 }
4  while W ≠ ∅ do
5     pick [P, O] from W; add [P, O] to Q̃
6     if P = ∅ then add [P, O] to F̃
7     for all a ∈ Σ do
8        P' = δ(P, a)
9        if O ≠ ∅ then O' ← δ(O, a) \ F else O' ← δ(P, a) \ F
10       add ([P, O], a, [P', O']) to δ̃
11       if [P', O'] ∉ Q̃ then add [P', Q'] to W
```

Figure 11.7 shows the result of applying the algorithm to our running example. The NCA is shown again at the top. The DCA is below it on the left. On the right we show the DFA obtained by applying the powerset construction. Observe how the powerset construction does not yield the correct result: the resulting NCA accepts for instance the word b^ω, which is not accepted by the NCA. For the complexity, we observe that the number of states of the DCA is bounded by the number of pairs $[P, O]$ such that $O \subseteq P \subseteq Q$. For every state $q \in Q$ there are three mutually exclusive posibilities: $q \in O$, $q \in P \setminus O$, and $q \in Q \setminus P$. Therefore, if A has n states then the number of states of B is at most 3^n.

Unfortunately, co-Büchi automata do not recognize all ω-regular languages. In particular, no NCA recognizes the language L of ω-words over $\{a, b\}$ containing infinitely many a's. To see why, assume the contrary. Then there would also be a DCA for the same language. Now we swap accepting and non-accepting states, and interpret the result as a DBA. It is easy to see that this DBA accepts the complement of L. But the complement of L is $(a + b)^* b^\omega$, which by Proposition 11.4 is not accepted by any DBA, and so we reach a contradiction.

So the next question in our quest whether there is a class of ω-automata that (1) recognizes all ω-regular languages and (2) has a determinization procedure.

11.4.2 Muller automata

A *(nondeterministic) Muller automaton* (NMA) has a collection $\{F_0, \ldots, F_{m-1}\}$ of sets of accepting states. A run ρ is *accepting* if the set of states ρ visits infinitely often is equal to one of the F_i's. Formally, ρ is accepting if $\inf(\rho) = F_i$ for some $i \in \{0, \ldots, m-1\}$. We speak of the *Muller condition* $\{F_0, \ldots, F_{m-1}\}$.

NMAs have the nice feature that any boolean combination of predicates of the form "state q is visited infinitely often" can be formulated as a Muller condition. It suffices to put in the collection all sets of states for which the predicate holds. For instance, the condition $(q \in \inf(\rho)) \wedge$

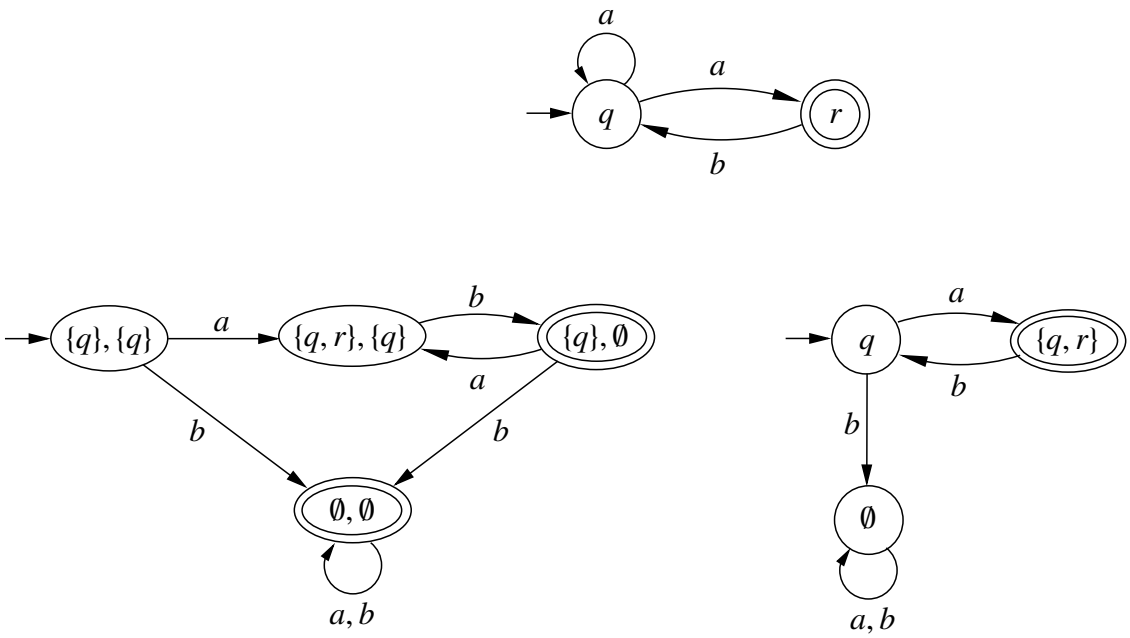


Figure 11.7: NCA of Figure 12.5 (top), DCA (lower left), and DFA (lower right)

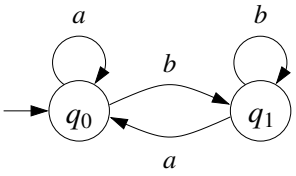
$\neg(q' \in \inf(\rho))$ corresponds to the Muller condition containing all sets of states F such that $q \in F$ and $q' \notin F$. In particular, the Büchi and generalized Büchi conditions are special cases of the Muller condition (as well as the Rabin and Street conditions introduced in the next sections). The obvious disadvantage is that the translation of a Büchi condition into a Muller condition involves an exponential blow-up: a Büchi automaton with states $Q = \{q_0, \ldots, q_n\}$ and Büchi condition $\{q_n\}$ is transformed into an NMA with the same states and transitions, but with a Muller condition $\{F \subseteq Q \mid q_n \in F\}$, a collection containing 2^n sets of states.

Deterministic Muller automata recognize all ω-regular languages. The proof of this result is complicated, and we omit it here.

Theorem 11.6 (Safra) *Any NBA with n states can be effectively transformed into a DMA of size* $n^{O(n)}$.

We can easily give a deterministic Muller automaton for the language $L = (a + b)^* b^\omega$, which, as shown in Proposition 11.4, is not recognized by any DBA. The automaton is
with Muller condition $\{ \{q_1\} \}$. The accepting runs are the runs ρ such that $\inf(\rho) = \{q_1\}$, i.e., the runs visiting state 1 infinitely often and state q_0 finitely often. Those are the runs that initially move between states q_0 and q_1, but eventually jump to q_1 and never visit q_0 again. It is easy to see that these runs accept exactly the words containing finitely many occurrences of a.

We finally show that an NMA can be translated into a NBA, and so that Muller and Büchi automata have the same expressive power. We start with a simple observation. In Chapter 4 we



have presented an algorithm that given NFAs A_1 and A_2 computes an NFA $A_1 \cup A_2$ recognizing $L(A_1) \cup L(A_2)$. The algorithm puts A_1 and A_2 "side by side", and adds a new initial state and transitions. It is graphically represented in Figure 11.8. It is easy to see that exactly the same algorithm works for NBAs.

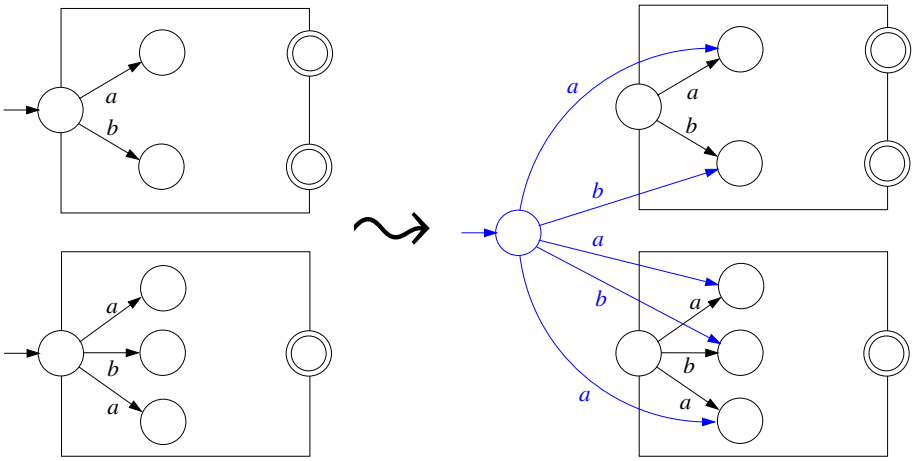


Figure 11.8: Union for NBAs

Equipped with this observation we proceed as follows. Given a Muller automaton $A = (Q, \Sigma, q_0, \delta, \{F_0, \ldots, F_{m-1}\})$, it is easy to see that $L_\omega(A) = \bigcup_{i=0}^{m-1} L_\omega(A_i)$, where $A_i = (Q, \Sigma, q_0, \delta, \{F_i\})$. So we proceed in three steps: first, we convert the NMA A_i into a NGA A_i'; then we convert A_i' into a NBA A_i'' using *NGAtoNBA*() and, finally, we construct $A'' = \bigcup_{i=0}^{m-1} A_i''$ using the algorithm for union of NFAs.

For the first step, we observe that, since an accepting run ρ of A_i satisfies $\inf(\rho) = F_i$, from some point on the run only visits states of F_i. In other words, ρ consists of an initial *finite* part, say ρ_0, that may visit all states, and an infinite part, say ρ_1, that only visits states of F_i. The idea for the construction for A_i' is to take two copies of A_i. The first one is a "full" copy, while the second one only contains copies of the states of F_i. For every transition $[q, 0] \xrightarrow{a} [q', 0]$ of the first copy such that $q' \in F_i$ there is another transition $[q, 0] \xrightarrow{a} [q', 1]$ leading to the "twin brother" $[q', 1]$. Intuitively, A_i' simulates ρ by executing ρ_0 in the first copy, and ρ_1 in the second. The condition that ρ_1 must visit each state of F_i infinitely often is enforced as follows: if $F_i = \{q_1, \ldots, q_k\}$, then we take for A_i the generalized Büchi condition $\{ \{[q_1, 1]\}, \ldots, \{q_k, 1]\} \}$.

Example 11.7 Figure 11.9 shows a NMA $A = (Q, \Sigma, \delta, q_0, \mathcal{F})$ where $\mathcal{F} = \{ \{q\}, \{r\} \}$. While A is

syntactically identical to the NGA of Figure 11.4, we now interpret $\mathcal{F}$ as a Muller condition: a run ρ is accepting if $\inf(\rho) = \{q\}$ or $\inf(\rho) = \{r\}$. In other words, an accepting run ρ eventually moves to q and stays there forever, or eventually moves to r and stays there forever. It follows that A accepts the ω-words that contain finitely many as or finitely many bs. The top-right part of the figure shows the two NGAs A'_0, A'_1 defined above. Since in this particular case $\mathcal{F}'_0$ and $\mathcal{F}'_1$ only contain singleton sets, A'_0 and A'_1 are in fact NBAs, i.e., we have $A''_0 = A'_0$ and $A''_1 = A'_1$. The bottom-right part shows the final NBA $A'' = A''_0 \cup A''_1$. □

Formally, the algorithm carrying out the first step of the construction looks as follows:

NMA1toNGA(A)
Input: NMA $A = (Q, \Sigma, q_0, \delta, \{F\}$
Output: NGA $A = (Q', \Sigma, q'_0, \delta', \mathcal{F}')$

```
1  Q', δ', F' ← ∅
2  q'_0 ← [q_0, 0]
3  W ← {[q_0, 0]}
4  while W ≠ ∅ do
5     pick [q, i] from W; add [q, i] to Q'
6     if q ∈ F and i = 1 then add {[q, 1]} to F'
7     for all a ∈ Σ, q' ∈ δ(q, a) do
8        if i = 0 then
9           add ([q, 0], a, [q', 0]) to δ'
10          if [q', 0] ∉ Q' then add [q', 0] to W
11          if q' ∈ F then
12             add ([q, 0], a, [q', 1]) to δ'
13             if [q', 1] ∉ Q' then add [q', 1] to W
14       else /* i = 1 */
15          if q' ∈ F then
16             add ([q, 1], a, [q', 1]) to δ'
17             if [q', 1] ∉ Q' then add [q', 1] to W
18 return (Q', Σ, q'_0, δ', F')
```

Complexity. Assume Q contains n states and $\mathcal{F}$ contains m accepting sets. Each of the NGAs $A'_0, \ldots, A'_{m-1}$ has at most $2n$ states, and an acceptance condition containing at most m acceptance sets. So each of the NBAs $A'_0, \ldots, A'_{m-1}$ has at most $2n^2$ states, and the final NBA has at most $2n^2m + 1$ states. Observe in particular that while the conversion from NBA to NMA involves a possibly exponential blow-up, the conversion NMA to NBA does not.

It can be shown that the exponential blow-up in the conversion from NBA to NMA cannot be avoided, which leads to the next step in our quest: is there a class of ω-automata that (1) recognizes

all ω-regular languages, (2) has a determinization procedure, and (3) has polynomial conversion algorithms to and from NBA.

11.4.3 Rabin automata

The acceptance condition of a *nondeterministic Rabin automaton* (NRA) is a set of pairs $\mathcal{F} = \{\langle F_0, G_0\rangle, \ldots, \langle F_m, G_m\rangle\}$, where the F_i's and G_i's are sets of states. A run ρ is *accepting* if there is a pair $\langle F_i, G_i\rangle$ such that ρ visits *some* state of F_i *infinitely often* and *all* states of G_i *finitely often*. Formally, ρ is accepting if there is $i \in \{1, \ldots, m\}$ such that $\inf(\rho) \cap F_i \neq \emptyset$ and $\inf(\rho) \cap G_i = \emptyset$.

A Büchi automaton can be easily transformed into a Rabin automaton and vice versa, without any exponential blow-up.

NBA $\to$ NRA. A Büchi condition $\{q_1, \ldots, q_k\}$ corresponds to the Rabin condition $\{(\{q_1\}, \emptyset), \ldots, (\{q_n\}, \emptyset)\}$.

NRA $\to$ NBA. Given a Rabin automaton $A = (Q, \Sigma, q_0, \delta, \{\langle F_0, G_0\rangle, \ldots, \langle F_{m-1}, G_{m-1}\rangle\})$, it is easy to see that, as for Muller automata, we have $L_\omega(A) = \bigcup_{i=0}^{m-1} L_\omega(A_i)$, where $A_i = (Q, \Sigma, q_0, \delta, \{\langle F_i, G_i\rangle\})$. In this case we directly translate each A_i into an NBA. Since an accepting run ρ of A_i satisfies $\inf(\rho) \cap G_i = \emptyset$, from some point on the run only visits states of $Q_i \setminus G_i$. So ρ consists of an initial *finite* part, say ρ_0, that may visit all states, and an infinite part, say ρ_1, that only visits states of $Q \setminus G_i$. Again, we take two copies of A_i. Intuitively, A_i' simulates ρ by executing ρ_0 in the first copy, and ρ_1 in the second. The condition that ρ_1 must visit some state of F_i infinitely often is enforced by taking F_i as Büchi condition.

Example 11.8 Figure 11.9 can be reused to illustrate the conversion of a Rabin into a Büchi automaton. Consider the automaton on the left, but this time with Rabin accepting condition $\{\langle F_0, G_0\rangle, \langle F_1, G_1\rangle\}$, where $F_0 = \{q\} = G_1$, and $G_0 = \{r\} = F_1$. Then the automaton accepts the ω-words that contain finitely many as or finitely many bs. The Büchi automata A_0', A_1' are as shown at the top-right part, but now instead of NGAs they are NBAs with accepting states $[q, 1]$ and $[r, 1]$, respectively. The final NBA is exactly the same one. □

For the complexity, observe that each of the A_i' has at most $2n$ states, and so the final Büchi automaton has at most $2nm + 1$ states.

The proof that NRAs are as expressive as DRAs is complicated. Since NRAs and NBAs are equally expressive by the conversions above, it suffices to show that DRAs are as expressive as NBAs. We only present the result, without proof.

Theorem 11.9 (Safra) *Any NBA with n states can be effectively transformed into a DRA of size $n^{\mathcal{O}(n)}$ and $\mathcal{O}(n)$ accepting pairs.*

The accepting condition of Rabin automata is not "closed under negation". Recall the condition:

there is $i \in \{1, \ldots, m\}$ such that $\inf(\rho) \cap F_i \neq \emptyset$ and $\inf(\rho) \cap G_i = \emptyset$

The negation is of the form

for every $i \in \{1, \ldots, m\}$: $\inf(\rho) \cap F_i = \emptyset$ or $\inf(\rho) \cap G_i \neq \emptyset$

This is called the *Streett condition*. We finish the chapter with a short discussion of Street automata.

Streett automata

The final class of ω-automata we consider are Streett automata. In a *Streett automaton*, the acceptance condition is again a set of pairs $\{\langle F_1, G_1\rangle, \ldots, \langle F_m, G_m\rangle\}$, where F_i, G_i are sets of states. A run ρ is accepting if for *every* pair $\langle F_i, G_i\rangle$, if ρ visits some state of F_i infinitely often, then it also visits some state of G_i infinitely often. Formally, a run ρ is accepting if for every $i \in \{1, \ldots, m\}$ $\inf(\rho) \cap F_i \neq \emptyset$ implies $\inf(\rho) \cap G_i \neq \emptyset$ (equivalently, if $\inf(\rho) \cap F_i = \emptyset$ or $\inf(\rho) \cap G_i \neq \emptyset$ for every $i \in \{1, \ldots, m\}$). The reader has probably observed that this is the dual of Rabin's condition: a run satisfies the Streett condition $\{\langle F_1, G_1\rangle, \ldots, \langle F_m, G_m\rangle\}$ if and only if it does not satisfy the Rabin condition $\{\langle F_1, G_1\rangle, \ldots, \langle F_m, G_m\rangle\}$.

A Büchi automaton can be easily transformed into a Streett automaton and vice versa. However, the conversion from Streett to Büchi is exponential.

NBA $\to$ NSA. A Büchi condition $\{q_1, \ldots, q_k\}$ corresponds to the Streett condition $\{\langle Q, \{q_1, \ldots, q_k\}\rangle\}$.

NSA $\to$ NBA. We can transform into an NBA by following the path NSA $\to$ NMA $\to$ NBA. This yields for a NSA with n states an NBA with $2n^2 2^n$ states. It can be shown that the exponential blow-up is unavoidable; in other words, Streett automata can be exponentially more succinct than Büchi automata.

Example 11.10 Let $\Sigma = \{0, 1, 2\}$. For $n \geq 1$, we represent an infinite sequence $x_1, x_2, \ldots$ of vectors of dimension n with components in Σ by the ω-word $x_1 x_2 \ldots$ over Σ^n. Let L_n be the language in which, for each component $i \in \{1, \ldots, n\}$, $x_j(i) = 1$ for infinitely many j's if and only if $x_k(i) = 2$ for infinitely many k's. It is easy to see that L_n can be accepted by a NSA with $3n$ states and $2n$ accepting pairs, but cannot be accepted by any NBA with less than 2^n states. □

Exercises

Exercise 81 A finite set of finite words is always a regular language, but a finite set of ω-words is not always an ω-regular language: find an ω-word $w \in \{a, b\}^\omega$ such that no Büchi automaton recognizes the language $\{w\}$.

Exercise 82 Construct Büchi automata and ω-regular expressions recognizing the following languages over the alphabet $\{a, b, c\}$.

1. $\{w \in \{a, b, c\}^\omega \mid \{a, b\} \supseteq inf(w)\}$
2. $\{w \in \{a, b, c\}^\omega \mid \{a, b\} = inf(w)\}$
3. $\{w \in \{a, b, c\}^\omega \mid \{a, b\} \subseteq inf(w)\}$
4. $\{w \in \{a, b, c\}^\omega \mid \{a, b, c\} = inf(w)\}$
5. $\{w \in \{a, b, c\}^\omega \mid \text{if } a \in inf(w) \text{ then } \{b, c\} \subseteq inf(w)\}$

Hint: It may be easier to construct a generalized Büchi automaton first and then transform it into a Büchi automaton.

Exercise 83 Find algorithms for the following decision problems:

- Given finite words $u, v, x, y \in \Sigma^*$, decide whether the ω-words $= u\, v^\omega$ and $x\, y^\omega$ are equal.
- Given a Büchi automaton A and finite words u, v, decide whether A accepts the ω-word $u\, v^\omega$.

Exercise 84 Find ω-regular expressions for the following languages:

1. $\{w \in \{a, b\}^\omega \mid k \text{ is even for each substring } ba^k b \text{ of } w\}$
2. $\{w \in \{a, b\}^\omega \mid w \text{ has no occurrence of } bab\}$

Exercise 85 A Büchi automaton is *deterministic in the limit* if all its accepting states and their descendants are deterministic states. Formally, $\mathcal{A} = (\Sigma, Q, Q^0, \delta, \alpha)$ is deterministic in the limit if $|\delta(q, \sigma)| \leq 1$ for every state $q \in Q$ that is reachable from some state of α, and for every $\sigma \in \Sigma$. Prove that every language recognized by nondeterministic Büchi automata is also accepted by Büchi automata deterministic in the limit.

Exercise 86 The *parity acceptance condition* for ω-automata is defined as follows. Every state q of the automaton is assigned a natural number n_q. A run ρ is accepting if the number $\max\{n_s \mid s \in inf(\rho)\}$ is even.

- Find a parity automaton accepting the language $L = \{w \in \{a, b\}^\omega \mid w \text{ has exactly two occurrences of } ab\}$.
- Show that each language accepted by a parity automaton is also accepted by a Rabin automaton and vice versa.

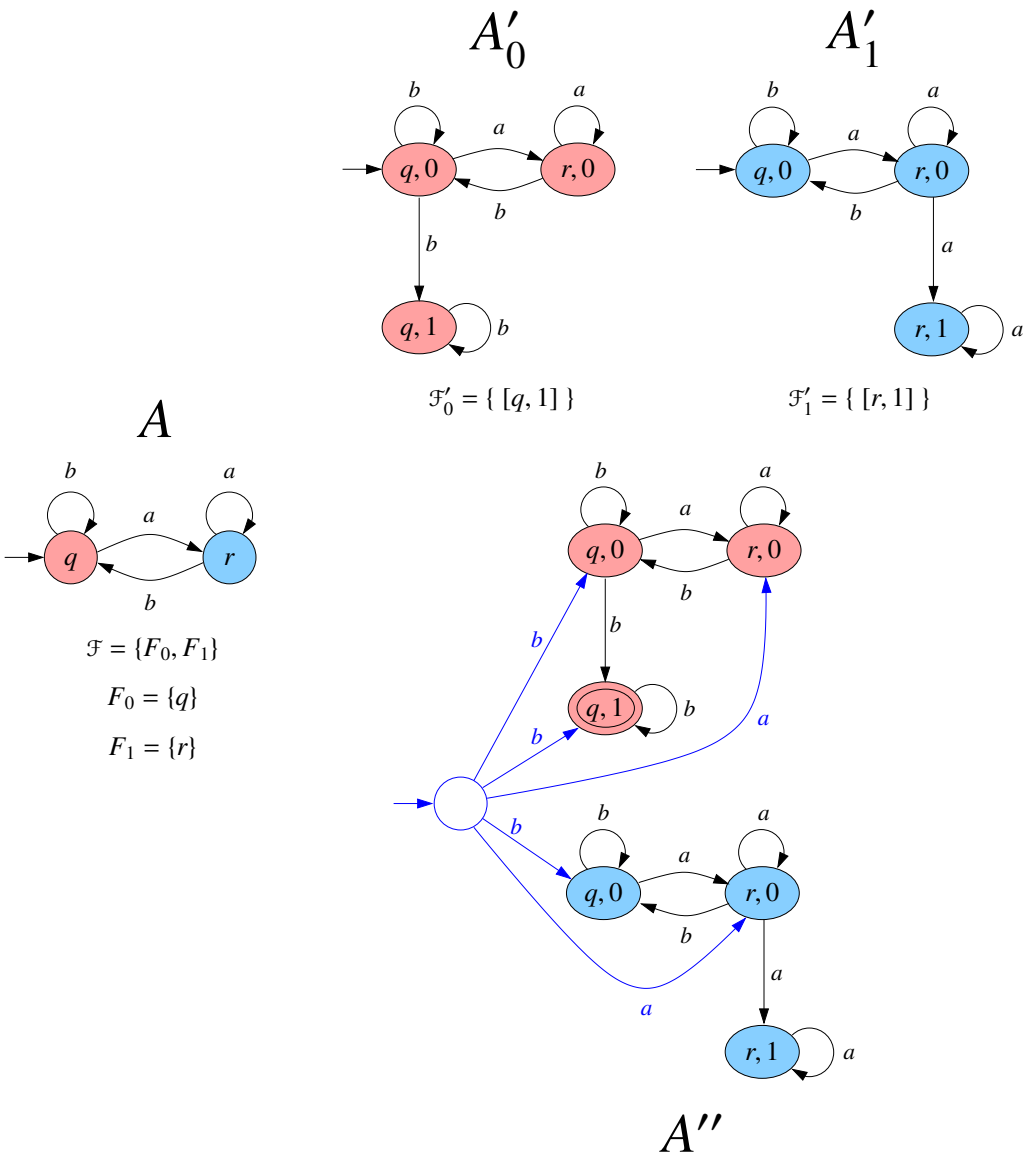


Figure 11.9: A Muller automaton and its conversion into a NBA

Chapter 12

Boolean operations: Implementations

The list of operations of Chapter 4 can be split into two parts, with the the boolean operations union, intersection, and complement in the first part, and the emptiness, inclusion, and equality tests in the second. This chapter deals with the boolean operations, while the tests are discussed in the next one. Observer that we now leave the membership test out. Observe that a test for arbitrary ω-words does not make sense, because no description formalism can represent arbitrary ω-words. For ω-words of the form $w_1(w_2)^\omega$, where w_1, w_2 are finite words, membership in an ω-regular language L can be implemented by checking if the intersection of L and $\{w_1(w_2)^\omega\}$ is empty.

We provide implementations for ω-languages represented by NBAs and NGAs. We do not discuss implementations on DBAs, because they cannot represent all ω-regular languages.

In Section 12.1 we show that union and intersection can be easily implemented using constructions already presented in Chapter 2. The rest of the chapter is devoted to the complement operation, which is more involved.

12.1 Union and intersection

As already observed in Chapter 2, the algorithm for union of regular languages represented as NFAs also works for NBAs and for NGAs.

One might be tempted to think that, similarly, the intersection algorithm for NFAs also works for NBAs. However, this is not the case. Consider the two Büchi automata A_1 and A_2 of Figure 12.1. The Büchi automaton $A_1 \cap A_2$ obtained by applying algorithm *IntersNFA*(A_1, A_2) in page

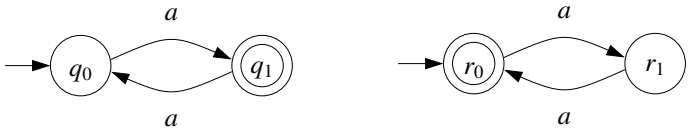


Figure 12.1: Two Büchi automata accepting the language a^ω

64 (more precisely, by interpreting the output of the algorithm as a Büchi automaton) is shown in Figure 12.2. It has no accepting states, and so $L_\omega(A_1) = L_\omega(A_2) = \{a^\omega\}$, but $L_\omega(A_1 \cap A_2) = \emptyset$.

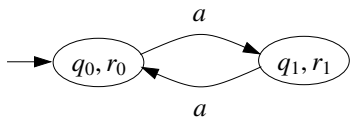


Figure 12.2: The automaton $A_1 \cap A_2$

What happened? A run ρ of $A_1 \cap A_2$ on an ω-word w is the result of pairing runs ρ_1 and ρ_2 of A_1 and A_2 on w. Since the accepting set of $A_1 \cap A_2$ is the cartesian product of the accepting sets of A_1 and A_2, ρ is accepting if ρ_1 and ρ_2 *simultaneously* visit accepting states infinitely often. This condition is too strong, and as a result $L_\omega(A_1 \cap A_2)$ can be a strict subset of $L_\omega(A_1) \cap L_\omega(A_2)$.

This problem is solved by means of the observation we already made when dealing with NGAs: the run ρ visits states of F_1 and F_2 infinitely often if and only if the following two conditions hold:

(1) ρ eventually visits F_1; and

(2) every visit of ρ to F_1 is eventually followed by a visit to F_2 (with possibly further visits to F_1 in-between), and every visit to F_2 is eventually followed by a visit to F_1 (with possibly further visits to F_1 in-between).

We proceed as in the translation NGA $\rightarrow$ NBA. Intuitively, we take two "copies" of the pairing $[A_1, A_2]$, and place them one of top of the other. The first and second copies of a state $[q_1, q_2]$ are called $[q_1, q_2, 1]$ and $[q_1, q_2, 2]$, respectively. The transitions leaving the states $[q_1, q_2, 1]$ such that $q_1 \in F_1$ are redirected to the corresponding states of the second copy, i.e., every transition of the form $[q_1, q_2, 1] \xrightarrow{a} [q_1', q_2', 1]$ is replaced by $[q_1, q_2, 1] \xrightarrow{a} [q_1', q_2', 2]$. Similarly, the transitions leaving the states $[q_1, q_2, 2]$ such that $q_2 \in F_2$ are redirected to the first copy. We choose $[q_{01}, q_{02}, 1]$, as initial state, and declare the states $[q_1, q_2, 1]$ such that $q_1 \in F_1$ as accepting.

Example 12.1 Figure 12.3 shows the result of the construction for the NBAs A_1 and A_2 of Figure 12.1, after removing the states that are not reachable form the initial state. Since q_0 is not an

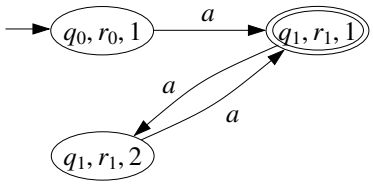


Figure 12.3: The NBA $A_1 \cap_\omega A_2$ for the automata A_1 and A_2 of Figure 12.1

accepting state of A_1, the transition $[q_0, r_0, 1] \xrightarrow{a} [q_1, r_1, 1]$ is not redirected. However, since q_1 is

an accepting state, transitions leaving $[q_1, r_1, 1]$ must jump to the second copy, and so we replace $[q_1, r_1, 1] \xrightarrow{a} [q_0, r_0, 1]$ by $[q_1, r_1, 1] \xrightarrow{a} [q_0, r_0, 2]$. Finally, since r_0 is an accepting state of A_2, transitions leaving $[q_0, r_0, 2]$ must return to the first copy, and so we replace $[q_0, r_0, 2] \xrightarrow{a} [q_1, r_1, 2]$ by $[q_0, r_0, 2] \xrightarrow{a} [q_1, r_1, 1]$. The only accepting state is $[q_1, r_1, 1]$, and the language accepted by the NBA is a^ω. □

To see that the construction works, observe first that a run ρ of this new NBA still corresponds to the pairing of two runs ρ_1 and ρ_2 of A_1 and A_2, respectively. Since all transitions leaving the accepting states jump to the second copy, ρ is accepting iff it visits both copies infinitely often, which is the case iff ρ_1 and ρ_2 visit states of F_1 and F_2, infinitely often, respectively.

Algorithm *IntersNBA*(), shown below, returns an NBA $A_1 \cap_\omega A_2$. As usual, the algorithm only constructs states reachable from the initial state.

IntersNBA(A_1, A_2)
Input: NBAs $A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$, $A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$
Output: NBA $A_1 \cap_\omega A_2 = (Q, \Sigma, \delta, q_0, F)$ with $L_\omega(A_1 \cap_\omega A_2) = L_\omega(A_1) \cap L_\omega(A_2)$

```
1  Q, δ, F ← ∅
2  q0 ← [q01, q02, 1]
3  W ← { [q01, q02, 1] }
4  while W ≠ ∅ do
5     pick [q1, q2, i] from W
6     add [q1, q2, i] to Q'
7     if q1 ∈ F1 and i = 1 then add [q1, q2, 1] to F'
8     for all a ∈ Σ do
9        for all q1' ∈ δ1(q1, a), q2' ∈ δ(q2, a) do
10          if i = 1 and q1 ∉ F1 then
11             add ([q1, q2, 1], a, [q1', q2', 1]) to δ
12             if [q1', q2', 1] ∉ Q' then add [q1', q2', 1] to W
13          if i = 1 and q1 ∈ F1 then
14             add ([q1, q2, 1], a, [q1', q2', 2]) to δ
15             if [q1', q2', 2] ∉ Q' then add [q1', q2', 2] to W
16          if i = 2 and q2 ∉ F2 then
17             add ([q1, q2, 2], a, [q1', q2', 2]) to δ
18             if [q1', q2', 2] ∉ Q' then add [q1', q2', 2] to W
19          if i = 2 and q2 ∈ F2 then
20             add ([q1, q2, 2], a, [q1', q2', 1]) to δ
21             if [q1', q2', 1] ∉ Q' then add [q1', q2', 1] to W
22 return (Q, Σ, δ, q0, F)
```

There is an important case in which the construction for NFAs can also be applied to NBAs,

namely when all the states of one of the two NBAs, say A_1 are accepting. In this case, the condition that two runs ρ_1 and ρ_2 on an ω-word w *simultaneously* visit accepting states infinitely often is equivalent to the weaker condition that does not require simultaneity: any visit of ρ_2 to an accepting state is a simultaneous visit of ρ_1 and ρ_2 to accepting states.

It is also important to observe a difference with the intersection for NFAs. In the finite word case, given NFAs $A_1, \ldots, A_k$ with $n_1, \ldots, n_k$ states, we can compute an NFA for $L(A_1) \cap \ldots \cap L(A_n)$ with at most $\prod_{i=1}^{k} n_i$ states by repeatedly applying the intersection operation, and this construction is optimal (i.e., there is a family of instances of arbitrary size such that the smallest NFA for the intersection of the languages has the same size). In the NBA case, however, the repeated application of *IntersNBA*() is not optimal. Since *IntersNBA*() introduces an additional factor of 2 in the number of states, for $L_\omega(A_1) \cap \ldots \cap L_\omega(A_k)$ it yields an NBA with $2^{k-1} \cdot n_1 \cdot \ldots \cdot n_k$ states. We obtain a better construction proceeding as in the translation NGA $\rightarrow$ NBA: we produce k copies of $A_1 \times \ldots \times A_k$, and move from the i-th copy to the $(i+1)$-th copy when we hit an accepting state of A_i. This construction yields an NBA with $k \cdot n_1 \cdot \ldots \cdot n_k$ states.

12.2 Complement

So far we have been able to adapt the constructions for NFAs to NBAs. The situation is considerably more involved for complement.

12.2.1 The problems of complement

Recall that for NFAs a complement automaton is constructed by first converting the NFA into a DFA, and then exchanging the final and non-final states of the DFA. For NBAs this approach breaks down completely:

(a) The subset construction does not preserve ω-languages; i.e, a NBA and the result of applying the subset construction to it do not necessarily accept the same ω-language.

The NBA on the left of Figure 12.4 accepts the empty language. However, the result of applying the subset construction to it, shown on the right, accepts a^ω. Notice that both automata accept the same *finite* words.

Figure 12.4: The subset construction does not preserve ω-languages

(b) The subset construction cannot be replaced by another determinization procedure, because no such procedure exists: As we have seen in Proposition 11.4, some languages are accepted by NBAs, but not by DBAs.

(c) The automaton obtained by exchanging accepting and non-accepting states in a given DBA does not necessarily recognize the complement of the language.

In Figure 12.1, A_2 is obtained by exchanging final and non-final states in A_1. However, both A_1 and A_2 accept the language a^ω. Observe that as automata for finite words they accept the words over the letter a of even and odd length, respectively.

Despite these discouraging observations, NBAs turn out to be closed under complement. For the rest of the chapter we fix an NBA $A = (Q, \Sigma, \delta, q_0, F)$ with n states, and use Figure 12.5 as running example. Further, we abbreviate "infinitely often" to "i.o.". We wish to build an automaton

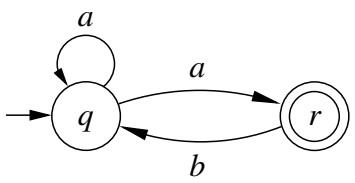


Figure 12.5: Running example for the complementation procedure

$\overline{A}$ satisfying:

no path of $dag(w)$ visits accepting states of A i.o.
if and only if
some run of w in $\overline{A}$ visits accepting states of $\overline{A}$ i.o.

We give a summary of the procedure. First, we define the notion of *ranking*. For the moment it suffices to say that a ranking of w is the result of decorating the nodes of $dag(w)$ with numbers. This can be done in different ways, and so, while a word w has one single dag $dag(w)$, it may have many rankings. The essential property of rankings will be:

no path of $dag(w)$ visits accepting states of A i.o.
if and only if for some ranking $R(w)$
every path of $dag(w)$ visits nodes of odd rank i.o.

In the second step we profit from the determinization construction for co-Büchi automata. Recall that the construction maps $dag(w)$ to a run ρ of a new automatonsuch that: every path of $dag(w)$ visits accepting states of A i.o. if and only if ρ visits accepting states of the new automaton i.o. We apply the same construction to map every ranking $R(w)$ to a run ρ of a new automaton B such that

every path of $dag(w)$ visits nodes of odd rank i.o. (in $R(w)$)
if and only if
the run ρ visits states of B i.o.

This immediately implies $L_\omega(B) = \overline{L_\omega(A)}$. However, the automaton B may in principle have an infinite number of states! In the final step, we show that a finite subautomaton $\overline{A}$ of B already recognizes the same language as B, and we are done.

12.2.2 Rankings and ranking levels

Recall that, given $w \in al^\omega$, the directly acyclic graph $dag(w)$ is the result of bundling together the runs of A on w. A *ranking* of $dag(w)$ is a mapping $R(w)$ that associates to each node of $dag(w)$ a natural number, called a *rank*, satisfying the following two properties:

(a) the rank of a node is greater than or equal to the rank of its children, and

(b) the rank of an accepting node is even.

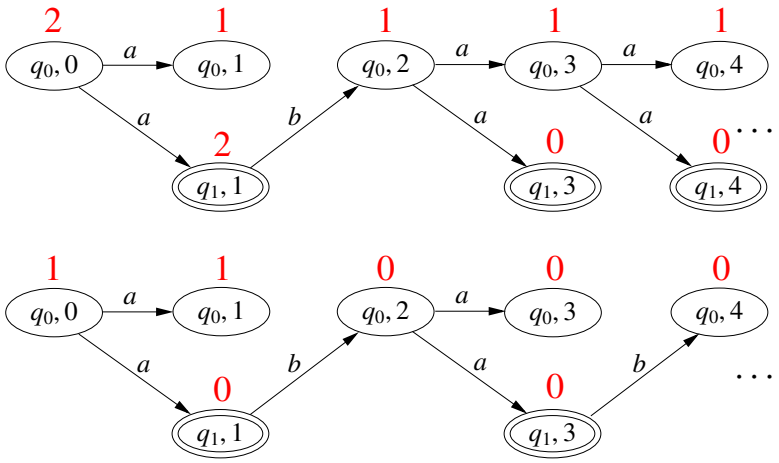


Figure 12.6: Rankings for $dag(aba^\omega)$ and $dag((ab)^\omega)$

The ranks of the nodes in an infinite path form a non-increasing sequence, and so there is a node such that all its (infinitely many) successors have the same rank; we call this number the *stable rank* of the path. Figure 12.6 shows rankings for $dag(aba^\omega)$ and $dag((ab)^\omega)$. Both have one single infinite path with stable rank 1 and 0, respectively. We now prove the fundamental property of rank ings:

Proposition 12.2 *No path of $dag(w)$ visits accepting nodes of A i.o. if and only if for some ranking $R(w)$ every infinite path of $dag(w)$ visits nodes of odd rank i.o.*

Proof: If all infinite paths of a ranking R have odd stable rank, then each of them contains only finitely many nodes with even rank. Since accepting nodes have even ranks, no path visits accepting nodes i.o.

For the other direction, assume that no path of $dag(w)$ visits accepting nodes of A i.o. Give each accepting node $\langle q, l\rangle$ the rank $2k$, where k is the maximal number of accepting nodes in the paths starting at $\langle q, l\rangle$, and give a non-accepting nodes rank $2k+1$, where $2k$ is the maximal rank of its descendants with even rank. In the ranking so obtained every infinite path visits nodes of even rank only finitely often, and therefore it visits nodes of odd rank i.o. □

Recall that the i-th level of $dag(w)$ is defined as the set of nodes of $dag(w)$ of the form $\langle q, i\rangle$. Let $\mathcal{R}$ be the set of all ranking levels. Any ranking r of $dag(w)$ can be decomposed into an infinite sequence $lr_1, lr_2, \ldots$ of level rankings by defining $lr_i(q) = r(\langle q, i\rangle)$ if $\langle q, i\rangle$ is a node of $dag(w)$, and $lr_i(q) = \bot$ otherwise. For example, if we represent a level ranking lr of our running example by the column vector

$$\begin{bmatrix} lr(q_0) \\ lr(q_1) \end{bmatrix},$$

then the rankings of Figure 12.6 correspond to the sequences

$$\begin{bmatrix} 2 \\ \bot \end{bmatrix}\begin{bmatrix} \bot \\ 2 \end{bmatrix}\begin{bmatrix} 1 \\ \bot \end{bmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix}^{\omega}$$

$$\begin{bmatrix} 1 \\ \bot \end{bmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix}\left(\begin{bmatrix} 0 \\ \bot \end{bmatrix}\begin{bmatrix} 0 \\ 0 \end{bmatrix}\right)^{\omega}$$

For two level rankings lr and lr' and a letter $a \in \Sigma$, we write $lr \overset{a}{\mapsto} lr'$ if for every $q' \in Q$:

- $lr'(q') = \bot$ iff no $q \in Q$ satisfies $q \xrightarrow{a} q'$, and
- $lr(q) \geq lr'(q')$ for every $q \in Q$ satisfying $q \xrightarrow{a} q'$.

12.2.3 A (possible infinite) complement automaton

We construct an NBA B an infinite number of states (and many initial states) whose runs on an ω-word w are the rankings of $dag(w)$. The automaton accepts a ranking R iff every infinite path of R visits nodes of odd rank i.o.

We start with an automaton without any accepting condition:

- The states are all the possible ranking levels.
- The initial states are the levels rl_n defined by: $rl_n(q_0) = n$, and $lr_n(q) = \bot$ for every $q \neq q_0$.
- The transitions are the triples (rl, a, rl'), where rl and rl' are ranking levels, $a \in \Sigma$, and $rl \overset{a}{\mapsto} rl'$ holds.

The runs of this automaton on w clearly correspond to the rankings of $dag(w)$. Now we apply the same construction we used for determinization of co-Büchi automata. We decorate the ranking levels with a set of 'owing' states, namely those that owe a visit to a state of odd rank, and take as accepting states the *breakpoints* i.e., the levels with an empty set of 'owing' states. We get the Büchi automaton B:

- The states are all pairs $[rl, O]$, where rl is a ranking level and O is a subset of the states q for which $rl(q) \in \mathbb{N}$.
- The initial states are all pairs of the form $[lr, \{q_0\}]$ where $lr(q_0)$ is an even number and $lr_0(q) = \perp$ for every $q \neq q_0$, and of the form $[lr, \emptyset]$, where $lr(q_0)$ is an odd number and $lr_0(q) = \perp$ for every $q \neq q_0$.
- The transitions are the triples $[rl, O]\delta a[rl', O']$ if $lr \overset{a}{\mapsto} lr'$ and
  - $O \neq \emptyset$ and $O' = \{q' \in \delta(O, a) \mid lr'(q') \text{ is even }\}$, or
  - $O = \emptyset$ and $O' = \{q' \in Q \mid lr'(q') \text{ is even }\}$.
- The accepting states (breakpoints) are the pairs $[rl, \emptyset]$.

B accepts a ranking iff it contains infinitely many breakpoints. As we saw in the construction for co-Büchi automata, this is the case iff every infinite path of $dag(w)$ visits nodes of odd rank i.o., and so iff A does not accept w.

The remaining problems with this automaton are that its number of states is infinite, and that it has many initial states. Both can be solved by proving the following assertion: there exists a number k such that for every word w, if $dag(w)$ admits an odd ranking, then it admits an odd ranking whose initial node $\langle q_0, 0\rangle$ has rank k. (Notice that, since ranks cannot increase along paths, every node has rank at most k.) If we are able to prove this, then we can eliminate all states corresponding to ranking levels in which some node is mapped to a number larger than k: they are redundant. Moreover, the initial state is now fixed: it is the level ranking that maps q_0 to k and all other states to $\perp$.

Proposition 12.3 *Let n be the number of states of A. For every word $w \in \Sigma^\omega$, if w is rejected by A then $dag(w)$ has a ranking such that*

(a) every infinite path of $dag(w)$ visits nodes of odd rank i.o., and

(b) the initial node $\langle q_0, 0\rangle$ has rank $2n$.

Proof: In the proof we call a ranking satisfying (a) an *odd ranking*. Assume w is rejected by A. We construct an odd ranking in which $\langle q_0, 0\rangle$ has rank at most $2n$. Then we can just change the rank of the initial node to $2n$, since the change preserves the properties of a ranking.

In the sequel, given two DAGs D, D', we denote by $D' \subseteq D$ the fact that D' can be obtained from D through deletion of some nodes and their adjacent edges.

Assume that A rejects w. We describe an odd ranking for $dag(w)$. We say that a node $\langle q, l\rangle$ is *red* in a (possibly finite) DAG $D \subseteq dag(w)$ iff only finitely many nodes of D are reachable from $\langle q, l\rangle$. The node $\langle q, l\rangle$ is *yellow* in D iff all the nodes reachable from $\langle q, l\rangle$ (including itself) are not accepting. In particular, yellow nodes are not accepting. Observe also that the children of a red node are red, and the children of a yellow node are red or yellow. We inductively define an infinite sequence $D_0 \supseteq D_1 \supseteq D_2 \supseteq \ldots$ of DAGs as follows:

- $D_0 = dag(w)$;
- $D_{2i+1} = D_{2i} \setminus \{\langle q, l\rangle \mid \langle q, l\rangle \text{ is red in } D_{2i}\}$;
- $D_{2i+2} = D_{2i+1} \setminus \{\langle q, l\rangle \mid \langle q, l\rangle \text{ is yellow in } D_{2i+1}\}$.

Figure 12.7 shows D_0, D_1, and D_2 for $dag(aba^\omega)$. D_3 is the empty dag.

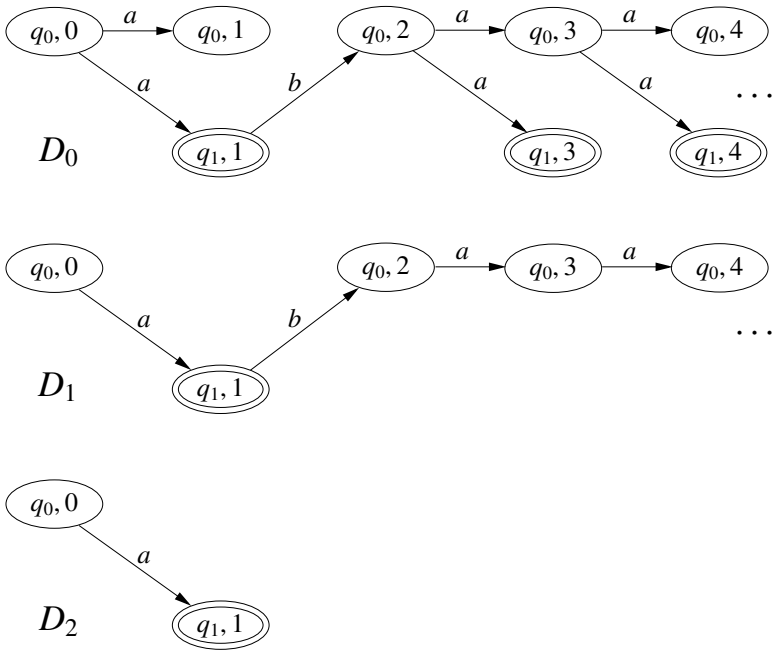


Figure 12.7: The DAGs D_0, D_1, D_2 for $dag(aba^\omega)$

Consider the function f that assigns to each node of $dag(w)$ a natural number as follows:

$$f(\langle q, l\rangle) = \begin{cases} 2i & \text{if } \langle q, l\rangle \text{ is red in } D_{2i} \\ 2i+1 & \text{if } \langle q, l\rangle \text{ is yellow in } D_{2i+1} \end{cases}$$

We prove that f is an odd ranking. The proof is divided into three parts:

(1) f assigns all nodes a number in the range $[0 \ldots 2n]$.

(2) If $\langle q', l' \rangle$ is a child of $\langle q, l \rangle$, then $f(\langle q', l' \rangle) \leq f(\langle q, l \rangle)$.

(3) If $\langle q, l \rangle$ is an accepting node, then $f(\langle q, l \rangle)$ is even.

Part (1). We show that for every $i \geq 0$ there exists a number l_i such that for all $l \geq l_i$, the DAG D_{2i} contains at most $n - i$ nodes of the form $\langle q, l \rangle$. This implies that D_{2n} is finite, and so that D_{2n+1} is empty, which in turn implies that f assigns all nodes a number in the range $[0 \ldots 2n]$.

The proof is by an induction on i. The case where $i = 0$ follows from the definition of G_0: indeed, in $dag(w)$ all levels $l \geq 0$ have at most n nodes of the form $\langle q, l \rangle$. Assume now that the hypothesis holds for i; we prove it for $i + 1$. Consider the DAG D_{2i}. If D_{2i} is finite, then D_{2i+1} is empty; D_{2i+2} is empty as well, and we are done. So assume that D_{2i} is infinite. We claim that D_{2i+1} contains some yellow node. Assume, by way of contradiction, that no node in D_{2i+1} is yellow. Since D_{2i} is infinite, D_{2i+1} is also infinite. Moreover, since D_{2i+1} is obtained by removing all red nodes from D_{2i}, every node of D_{2i+1} has at least one child. Let $\langle q_0, l_0 \rangle$ be an arbitrary node of D_{2i+1}. Since, by the assumption, it is not yellow, there exists an accepting node $\langle q'_0, l'_0 \rangle$ reachable from $\langle q_0, l_0 \rangle$. Let $\langle q_1, l_1 \rangle$ be a child of $\langle q'_0, l'_0 \rangle$. By the assumption, $\langle q_1, l_1 \rangle$ is also not yellow, and so there exists an accepting node $\langle q'_1, l'_1 \rangle$ reachable from $\langle q_1, l_1 \rangle$. We can thus construct an infinite sequence of nodes $\langle q_j, l_j \rangle$, $\langle q'_j, l'_j \rangle$ such that for all i the node $\langle q'_j, l'_j \rangle$ is accepting, reachable from $\langle q_j, l_j \rangle$, and $\langle q_{j+1}, l_{j+1} \rangle$ is a child of $\langle q'_j, l'_j \rangle$. Such a sequence, however, corresponds to a path in $dag(w)$ visiting infinitely many accepting nodes, which contradicts the assumption that A rejects w, and the claim is proved.

So, let $\langle q, l \rangle$ be a yellow node in D_{2i+1}. We claim that we can take $l_{i+1} = l$, that is, we claim that for every $j \geq l$ the dag D_{2i+2} contains at most $n - (i + 1)$ nodes of the form $\langle q, j \rangle$. Since $\langle q, l \rangle$ is in D_{2i+1}, it is not red in D_{2i}. Thus, infinitely many nodes of D_{2i} are reachable from $\langle q, l \rangle$. By König's Lemma, D_{2i} contains an infinite path $\langle q, l \rangle, \langle q_1, l + 1 \rangle, \langle q_2, l + 2 \rangle, \ldots$. For all $k \geq 1$, infinitely many nodes of D_{2i} are reachable from $\langle q_k, l + k \rangle$, and so $\langle q_k, l + k \rangle$ is not red in D_{2i}. Therefore, the path $\langle q, l \rangle, \langle q_1, l + 1 \rangle, \langle q_2, l + 2 \rangle, \ldots$ exists also in D_{2i+1}. Recall that $\langle q, l \rangle$ is yellow. Hence, being reachable from $\langle q, l \rangle$, all the nodes $\langle q_k, l + k \rangle$ in the path are yellow as well. Therefore, they are not in D_{2i+2}. It follows that for all $j \geq l$ the number of nodes of the form $\langle q, j \rangle$ in D_{2i+2} is strictly smaller than their number in D_{2i}, which, by the induction hypothesis, is $n - i$. So there are at most $n - (i + 1)$ nodes of the form $\langle q, j \rangle$ in D_{2i+2}, and the claim is proved.

Part(2). Follows from the fact that the children of a red node in D_{2i} are red, and the children of a yellow node in D_{2i+1} are yellow. Therefore, if a node has rank i, all its successors have rank at most i or lower.

Part(3). Nodes that get an odd rank are yellow at D_{2i+1} for some i, and so not accepting. □

Example 12.4 We construct the complements $\overline{A}_1$ and $\overline{A}_2$ of the two possible NBAs over the alphabet $\{a\}$ having one state and one transition: $B_1 = (\{q\}, \{a\}, \delta, \{q\}, \{q\})$ and $B_2 = (\{q\}, \{a\}, \delta, \{q\}, \emptyset)$,

where $\delta(q, a) = \{q\}$. The only difference between B_1 and B_2 is that the state q is accepting in B_1, but not in B_2. We have $L_\omega(A_1) = a^\omega$ and $L_\omega(A_2) = \emptyset$.

We begin with $\overline{B}_1$. A state of $\overline{B}_1$ is a pair $\langle lr, O\rangle$, where lr is the rank of node q (since there is only one state, we can identify lr and $lr(q)$). The initial state is $\langle 2, \{q\}\rangle$, because q has even rank and so it "owes" a visit to a node of odd rank. Let us compute the successors of $\langle 2, \{q\}\rangle$ under the letter a. Let $\langle lr', O'\rangle$ be a successor. Since $\delta(q, a) = \{q\}$, we have $lr' \neq \bot$, and since q is accepting, we have $lr' \neq 1$. So either $lr' = 0$ or $lr' = 2$. In both cases the visit to a node of odd rank is still "owed', which implies $O' = \{q\}$. So the successors of $\langle 2, \{q\}\rangle$ are $\langle 2, \{q\}\rangle$ and $\langle 0, \{q\}\rangle$. Consider now the successors $\langle lr'', O''\rangle$ of $\langle 0, \{q\}\rangle$. We have $lr'' \neq \bot$ and $lr'' \neq 1$ as before, but now, since ranks cannot increase a long a path, we also have $lr'' \neq 2$. So $lr'' = 0$, and, since the visit to the node of odd rank is still ' owed", the only successor of $\langle 0, \{q\}\rangle$ is $\langle 0, \{q\}\rangle$. Since the set of owing states is never empty, $\overline{B}_1$ has no accepting states, and so it recognizes the empty language. $\overline{B}_1$ is shown on the left of Figure 12.8. Let us now construct $\overline{B}_2$. The difference with $\overline{B}_2$ is that, since q is no longer

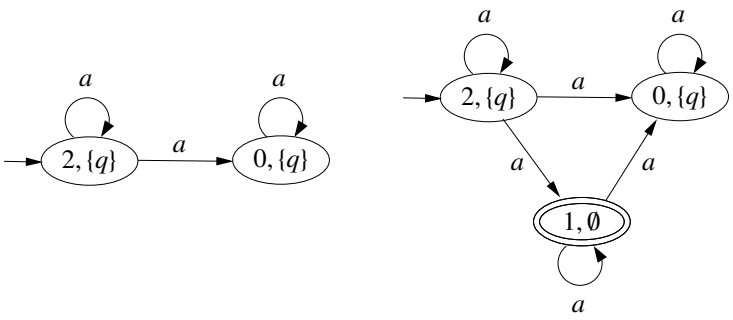


Figure 12.8: The NBAs $\overline{B}_1$ and $\overline{B}_2$

accepting, it can also have odd rank 1. So $[2, \{q\}]$ has three successors: $[2, \{q\}]$, $[1, \emptyset]$, and $[0, \{q\}]$. The successors of $[1, \emptyset]$ are $[1, \emptyset]$ and $[0, \{q\}]$, and the only successor of $[0, \{q\}]$ is $[0, \{q\}]$. The only accepting state is $[1, \emptyset]$, and $\overline{B}_2$ recognizes a^ω. □

The pseudocode for the complementation algorithm is shown below. In the code, $\mathcal{R}$ denotes the set of all level rankings, and lr_0 denotes the level ranking given by $lr(q_0) = 2|Q|$ and $lr(q) = \bot$ for every $q \neq q_0$. Recall also that $lr \overset{a}{\mapsto} lr'$ holds if for every $q' \in Q$: $lr'(q') = \bot$ iff no $q \in Q$ satisfies $q \overset{a}{\longrightarrow} q'$, and $lr(q) \geq lr'(q')$ for every $q \in Q$ satisfying $q \overset{a}{\longrightarrow} q'$.

CompNBA(A)
Input: NBA $A = (Q, \Sigma, \delta, q_0, F)$
Output: NBA $\overline{A} = (\overline{Q}, \Sigma, \overline{\delta}, \overline{q}_0, \overline{F})$ with $L_\omega(\overline{A}) = \overline{L_\omega(A)}$

```
1  Q̄, δ̄, F̄ ← ∅
2  q̄_0 ← [lr_0, {q_0}]
3  W ← { [lr_0, {q_0}] }
4  while W ≠ ∅ do
5     pick [lr, P] from W; add [lr, P] to Q̄
6     if P = ∅ then add [lr, P] to F̄
7     for all a ∈ Σ, lr' ∈ ℛ such that lr ↦^a lr' do
8        if P ≠ ∅ then P' ← {q ∈ δ(P, a) | lr'(q) is even }
9        else P' ← {q ∈ Q | lr'(q) is even }
10       add ([lr, P], a, [lr', P']) to δ̄
11       if [lr', P'] ∉ Q̄ then add [lr', P'] to W
12 return (Q̄, Σ, δ̄, q̄_0, F̄)
```

Complexity. Let n be the number of states of A. Since a level ranking is a mapping $lr\colon Q \to \{\bot\} \cup [0, 2n]$, there are at most $(2n+2)^n$ level rankings. So $\overline{A}$ has at most $(2n+2)^n \cdot 2^n \in n^{O(n)}$ states. Since $n^{O(n)} = 2^{O(n \cdot \log n)}$, we have introduced an extra $\log n$ factor in the exponent with respect to the subset construction for automata on finite words. The next section shows that this factor is unavoidable.

12.2.4 The size of $\overline{A}$

We exhibit a family $\{L_n\}_{n \geq 1}$ of infinitary languages such that L_n is accepted by an automaton with $n+2$ states and any Büchi automaton accepting the complement of L_n has at least $n! \in 2^{\Theta(n \log n)}$ states.

Let $\Sigma_n = \{1, \ldots, n, \#\}$. We associate to a word $w \in \Sigma_n^\omega$ the following directed graph $G(w)$: the nodes of $G(w)$ are $1, \ldots, n$ and there is an edge from i to j if w contains infinitely many occurrences of the word ij. Define L_n as the language of infinite words $w \in A^\omega$ for which $G(w)$ has a cycle and define $\overline{L}_n$ as the complement of L_n.

We first show that for all $n \geq 1$, L_n is recognized by a Büchi automaton with $n+2$ states. Let A_n be the automaton shown in Figure 12.9. We show that A_n accepts L_n.
(1) If $w \in L_n$, then A_n accepts w.
Choose a cycle $a_{i_1} a_{i_2} \ldots a_{i_k} a_{i_1}$ of $G(w)$. We construct an accepting run of A_n by picking q_{i_1} as initial state and iteratively applying the following rule:

> If the current state is q_{i_j}, stay there until the next occurrence of the word $a_{i_j} a_{i_j+1}$ in w, then use a_{i_j} to move to r, and use a_{i_j+1} to move to q_{i_j+1}.

By the definition of $G(w)$, r is visited infinitely often, and so w is accepted.
(2) If A_n accepts w, then $w \in L_n$.

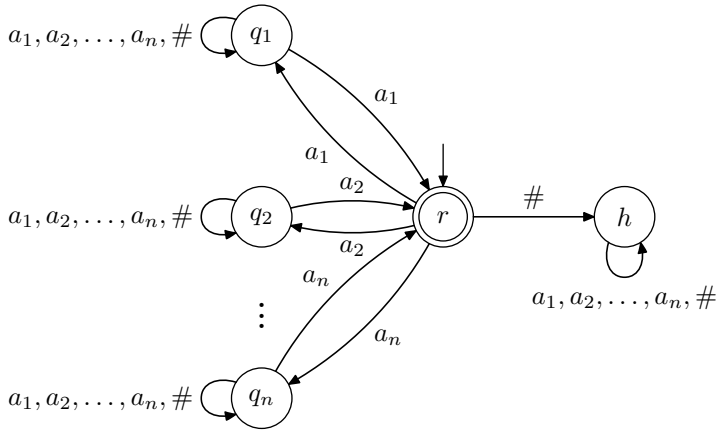


Figure 12.9: The automaton A_n

Let ρ be a run of A_n accepting w, and let $Q_\rho = inf(\rho) \cap \{q, \ldots, q_n\}$. Since ρ is accepting, it cannot stay in any of the q_i forever, and so for each $q_i \in Q_\rho$ there is $q_j \in Q_\rho$ such that the sequence $q_i r q_j$ appears infinitely often in ρ. Therefore, for every $q_i \in Q_\rho$ there is $q_j \in Q_\rho$ such that $a_i a_j$ appears infinitely often in w, or, in other words, such that $(a_i, a_j) \in G(w)$. Since Q_ρ is finite, $G(w)$ contains a cycle, and so $w \in L_n$.

Proposition 12.5 *For all $n \geq 1$, every NBA recognizing $\overline{L}_n$, has at least $n!$ states.*

Proof: We need some preliminaries. Given a permutation $\tau = \langle \tau(1), \ldots, \tau(n) \rangle$ of $\langle 1, \ldots, n \rangle$, we identify τ and the word $\tau(1) \ldots \tau(n)$. We make two observations:

(a) $(\tau\#)^\omega \in \overline{L}_n$ for every permutation τ.
The edges of $G((\tau\#)^\omega)$ are $\langle \tau(1), \tau(a_2) \rangle$, $\langle \tau(a_2), \tau(a_3) \rangle$, $\ldots$, $\langle \tau(a_{n-1}), \tau(a_n) \rangle$, and so $G((\tau\#)^\omega)$ is acyclic.

(b) If a word w contains infinitely many occurrences of two different permutations τ and τ' of $1 \ldots n$, then $w \in L_n$.
Since τ and τ' are different, there are i and j in $\{1, \ldots, n\}$ such that i precedes j in τ and j precedes i in τ'. Since w contains infinitely many occurrences of τ, $G(w)$ has a path from i to j. Since it contains infinitely many occurrences of τ', $G(w)$ has a path from j to i. So $G(w)$ contains a cycle, and so $w \in L_n$.

Now, let A be a Büchi automaton recognizing $\overline{L}_n$, and let τ, τ' be two arbitrary permutations of $(1, \ldots, n)$. By (a), there exist runs ρ and ρ' of A accepting $(\tau\#)^\omega$ and $(\tau'\#)^\omega$, respectively. We prove that the intersection of $\inf(\rho)$ and $\inf(\rho')$ is empty. This implies that A contains at least as many final states as permutations of $(1, \ldots, n)$, which proves the Proposition.

We proceed by contradiction. Assume $q \in \inf(\rho) \cap \inf(\rho')$. We build an accepting run ρ'' by "combining" ρ and ρ' as follows:

(0) Starting from the initial state of ρ, go to q following the run ρ.

(1) Starting from q, follow ρ' until having gone through a final state, and having read at least once the word τ'; then go back to q (always following ρ').

(2) Starting from q, follow ρ until having gone through a final state, and having read at least once the word τ; then go back to q (always following ρ).

(3) Go to (1).

The word accepted by ρ'' contains infinitely many occurrences of both τ and τ'. By (b), this word belongs to L_n, contradicting the assumption that A recognizes $\overline{L}_n$. □

Exercises

Exercise 87 Show that for every DBA A with n states there is an NBA B with $2n$ states such that $L_\omega(B) = \overline{L_\omega(A)}$.

Exercise 88 Give algorithms that directly complement deterministic Muller and parity automata, without going through Büchi automata.

Exercise 89 Let $A = (Q, \Sigma, q_0, \delta, \{\langle F_0, G_0\rangle, \ldots, \langle F_{m-1}, G_{m-1}\rangle\})$ be deterministic. Which is the relation between the languages recognized by A as a deterministic Rabin automaton and as a deterministic Streett automaton?

Exercise 90 Consider Büchi automata with universal accepting condition (UBA): an ω-word w is accepted if *every* run of the automaton on w is accepting, i.e., if *every* run of the automaton on w visits final states infinitely often.

Recall that automata on finite words with existential and universal accepting conditions recognize the same languages. Prove that is no longer the case for automata on ω-words by showing that for every UBA there is a DBA automaton that recognizes the same language. (This implies that the ω-languages recognized by UBAs are a proper subset of the ω-regular languages.)

Hint: On input w, the DBA checks that every path of $dag(w)$ visits some final state infinitely often. The states of the DBA are pairs (Q', O) of sets of the UBA where $O \subseteq Q'$ is a set of "owing" states (see below). Loosely speaking, the transition relation is defined to satisfy the following property: after reading a prefix w' of w, the DBA is at the state (Q', O) given by:

- Q' is the set of states reached by the runs of the UBA on w'.
- O is the subset of states of Q' that "owe" a visit to a final state of the UBA. (See the construction for the complement of a Büchi automaton.)

Chapter 13

Emptiness check: Implementations

We present efficient algorithms for the emptiness check. We fix an NBA $A = (Q, \Sigma, \delta, q_0, F)$. Since transition labels are irrelevant for checking emptiness, in this Chapter we redefine δ from a subset of $Q \times \Sigma \times Q$ into a subset of $Q \times Q$ as follows:

$$\delta := \{(q, q') \in Q \times Q \mid (q, a, q') \in \delta \text{ for some } a \in \Sigma\}$$

Since in many applications we have to deal with very large Büchi automata, we are interested in *on-the-fly* algorithms that do not require to know the Büchi automaton in advance, but check for emptiness while constructing it. More precisely, we assume the existence of an oracle that, provided with a state q returns the set $\delta(q)$.

We need a few graph-theoretical notions. If $(q, r) \in \delta$, then r is a *successor* of a and q is a *predecessor* of r. A *path* is a sequence $q_0, q_1, \ldots, q_n$ of states such that q_{i+1} is a successor of q_i for every $i \in \{0, \ldots, n-1\}$; we say that the path *leads* from q_0 to q_n. Notice that a path may consist of only one state; in this case, the path is *empty*, and leads from a state to itself. A *cycle* is a path that leads from a state to itself. We write $q \rightsquigarrow r$ to denote that there is a path from q to r.

Clearly, A is nonempty if it has an *accepting lasso*, i.e., a path $q_0 q_1 \ldots q_{n-1} q_n$ such that $q_n = q_i$ for some $i \in \{0, \ldots, n-1\}$, and at least one of $\{q_i, q_{i+1}, \ldots, q_{n-1}\}$ is accepting. The lasso consists of a path $q_0 \ldots q_i$, followed by a nonempty cycle $q_i q_{i+1} \ldots q_{n-1} q_i$. We are interested in emptiness checks that on input A report EMPTY or NONEMPTY, and in the latter case return an accepting lasso, as a *witness* of nonemptiness.

13.1 Algorithms based on depth-first search

We present two emptiness algorithms that explore A using depth-first search (DFS). We start with a brief description of depth-first search and some of its properties.

A depth-first search (DFS) of A starts at the initial state q_0. If the current state q still has unexplored outgoing transitions, then one of them is selected. If the transition leads to a not yet discovered state r, then r becomes the current state. If all of q's outgoing transitions have been

explored, then the search "backtracks" to the state from which q was discovered, i.e., this state becomes the current state. The process continues until q_0 becomes the current state again and all its outgoing transitions have been explored. Here is a pseudocode implementation (ignore the algorithm *DFS_Tree* for the moment).

```
DFS(A)
Input: NBA A = (Q, Σ, δ, q_0, F)
1  S ← ∅
2  dfs(q_0)
3  proc dfs(q)
4     add q to S
5     for all r ∈ δ(q) do
6        if r ∉ S then dfs(r)
7     return
```

```
DFS_Tree(A)
Input: NBA A = (Q, Σ, δ, q_0, F)
Output: Time-stamped tree (S, T, d, f)
1  S ← ∅
2  T ← ∅; t ← 0
3  dfs(q_0)
4  proc dfs(q)
5     t ← t + 1; d[q] ← t
6     add q to S
7     for all r ∈ δ(q) do
8        if r ∉ S then
9           add (q, r) to T; dfs(r)
10    t ← t + 1; f[q] ← t
11    return
```

Observe that *DFS* is nondeterministic, because we do not fix the order in which the states of $\delta(q)$ are examined by the **for**-loop. Since, by hypothesis, every state of an automaton is reachable from the initial state, we always have $S = Q$ after termination. Moreover, after termination every state $q \neq q_0$ has a distinguished input transition, namely the one that, when explored by the search, led to the discovery of q. It is well-known that the graph with states as nodes and these transitions as edges is a tree with root q_0, called a *DFS-tree*. If some path of the DFS-tree leads from q to r, then we say that q is an *ascendant* of r, and r is a *descendant* of q (in the tree).

It is easy to modify *DFS* so that it returns a DFS-tree, together with *timestamps* for the states. The algorithm, which we call *DFS_Tree* is shown above. While timestamps are not necessary for conducting a search itself, many algorithms based on depth-first search use them for other purposes[1]. Each state q is assigned two timestamps. The first one, $d[q]$, records when q is first discovered, and the second, $f[q]$, records when the search finishes examining the outgoing transitions of q. Since we are only interested in the relative order in which states are discovered and finished, we can assume that the timestamps are integers ranging between 1 and $2|Q|$. Figure 13.1 shows an example.

[1] In the rest of the chapter, and in order to present the algorithms is more compact form, we omit the instructions for computing the timestamps, and just assume they are there.

In our analyses we also assume that at every time point a state is *white*, *grey*, or *black*. A state q is white during the interval $[\,0, d[q]\,]$, grey during the interval $(\,d[q], f[q]\,]$, and black during the interval $(\,f[q], 2|Q|\,]$. So, loosely speaking, q is white, if it has not been yet discovered, grey if it has already been discovered but still has unexplored outgoing edges, or black if all its outgoing edges have been explored. It is easy to see that at all times the grey states form a path (*the grey path*) starting at q_0 and ending at the state being currently explored, i.e., at the state q such that *dfs*(q) is being currently executed; moreover, this path is always part of the DFS-tree.

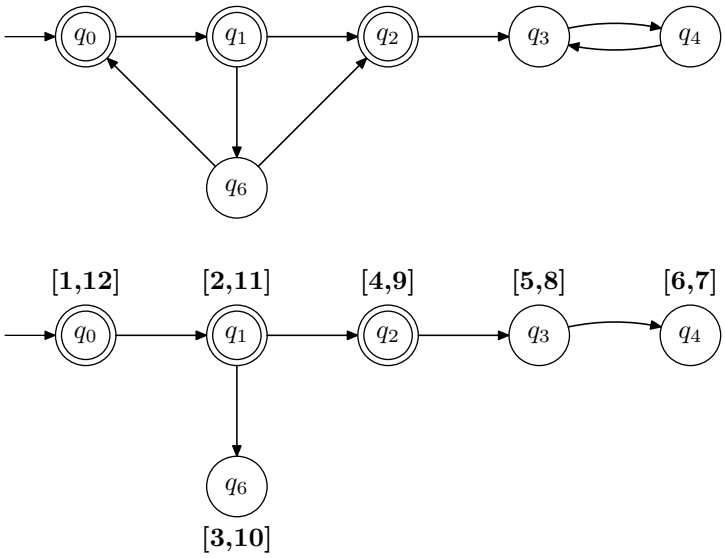


Figure 13.1: An NBA (the labels of the transitions have been omitted), and a possible run of *DFS_Tree* on it. The numeric intervals are the discovery and finishing times of the states, shown in the format $[d[q], f[q]]$.

We recall two important properties of depth-first search. Both follow easily from the fact that a procedure call suspends the execution of the caller, which is only resumed after the execution of the callee terminates.

Theorem 13.1 (Parenthesis Theorem) *In a DFS-tree, for any two states q and r, exactly one of the following four conditions holds, where $I(q)$ denotes the interval $(\,d[q], f[q]\,]$, and $I(q) \prec I(r)$ denotes that $f[q] < d[r]$ holds.*

- *$I(q) \subseteq I(r)$ and q is a descendant of r, or*
- *$I(r) \subseteq I(q)$ and r is a descendant of q, or*
- *$I(q) \prec I(r)$, and neither q is a descendant of r, nor r is a descendant of q, or*
- *$I(r) \prec I(q)$, and neither q is a descendant of r, nor r is a descendant of q.*

Theorem 13.2 (White-path Theorem) *In a DFS-tree, r is a descendant of q (and so $I(r) \subseteq I(q)$) if and only if at time $d[q]$ state r can be reached from q in A along a path of white states.*

13.1.1 The nested-DFS algorithm

To determine if A is empty we can search for the accepting states of A, and check if at least one of them belongs to a cycle. A naïve implementation proceeds in two phases, searching for accepting states in the first, and for cycles in the second. The runtime is quadratic: since an automaton with n states and m transitions has $\mathcal{O}(n)$ accepting states, and since searching for a cycle containing a given state takes $O(n + m)$ time, we obtain a $O(n^2 + nm)$ bound.

The nested-DFS algorithm runs in time $\mathcal{O}(n+m)$ by using the first phase not only to discover the reachable accepting states, but also to *sort* them. The searches of the second phase are conducted according to the order determined by the sorting. As we shall see, conducting the search in this order avoids repeated visits to the same state.

The first phase is carried out by a DFS, and the accepting states are sorted by increasing *finishing* (not discovery!) time. This is known as the *postorder* induced by the DFS. Assume that in the second phase we have already performed a search starting from the state q that has failed, i.e., no cycle of A contains q. Suppose we proceed with a search from another state r (which implies $f[q] < f[r]$), and this search discovers some state s that had already been discovered by the search starting at q. We claim that *it is not necessary to explore the successors of s again*. More precisely, we claim that $s \not\rightsquigarrow r$, and so it is useless to explore the successors of s, because the exploration cannot return any cycle containing r. The proof of the claim is based on the following lemma:

Lemma 13.3 *If $q \rightsquigarrow r$ and $f[q] < f[r]$ in some DFS-tree, then some cycle of A contains q.*

Proof: Let π be a path leading from q to r, and let s be the first node of π that is discovered by the DFS. By definition we have $d[s] \leq d[q]$. We prove that $s \neq q$, $q \rightsquigarrow s$ and $s \rightsquigarrow q$ hold, which implies that some cycle of A contains q.

- $q \neq s$. If $s = q$, then at time $d[q]$ the path π is white, and so $I(r) \subseteq I(q)$, contradicting $f[q] < f[r]$.
- $q \rightsquigarrow s$. Obvious, because s belongs to π.
- $s \rightsquigarrow q$. By the definition of s, and since $s \neq q$, we have $d[s] \leq d[q]$. So either $I(q) \subseteq I(s)$ or $I(s) \prec I(q)$. We claim that $I(s) \prec I(q)$ is not possible. Since at time $d[s]$ the subpath of π leading from s to r is white, we have $I(r) \subseteq I(s)$. But $I(r) \subseteq I(s)$ and $I(s) \prec I(q)$ contradict $f[q] < f[r]$, which proves the claim. Since $I(s) \prec I(q)$ is not possible, we have $I(q) \subseteq I(s)$, and hence q is a descendant of s, which implies $s \rightsquigarrow q$.

□

Example 13.4 The NBA of Figure 13.1 contains a path from q_1 to q_0, and the DFS-tree displayed satisfied$f[q_1] = 11 < 12 = f[q_0]$. As guaranteed by lemma 13.3, some cycle contains q_1, namely the cycle $q_1q_6q_0$. □

To prove the claim, we assume that $s \rightsquigarrow r$ holds and derive a contradiction. Since s was already discovered by the search starting at q, we have $q \rightsquigarrow s$, and so $q \rightsquigarrow r$. Since $f[q] < f[r]$, by Lemma 13.3 some cycle of A contains q, contradicting the assumption that the search from q failed.

Hence, during the second phase we only need to explore a transition at most once, namely when its source state is discovered for the first time. This guarantees the correctness of the following algorithm:

- Perform a DFS on A from q_0, and output the accepting states of A in postorder[2]. Let $q_1, \ldots, q_k$ be the output of the search, i.e., $f[q_1] < \ldots < f[q_k]$.
- For $i = 1$ to k, perform a DFS from the state q_i, with the following changes:
  - If the search visits a state q that was already discovered by any of the searches starting at $q_1, \ldots, q_{i-1}$, then the search backtracks.
  - If the search visits q_i, it stops and returns NONEMPTY.
- If none of the searches from $q_1, \ldots, q_k$ returns NONEMPTY, return EMPTY.

Example 13.5 We apply the algorithm to the example of Figure 13.1. Assume that the first DFS runs as in Figure 13.1. The search outputs the accepting states in postorder, i.e., in the order q_2, q_1, q_0. Figure 13.2 shows the transitions explored during the searches of the second phase. The search from q_2 explores the transitions labelled by $2.1, 2.2, 2.3$. The search from q_1 explores the transitions $1.1, \ldots, 1.5$. Notice that the search backtracks after exploring 1.1, because the state q_2 was already visited by the previous search. This search is successful, because transition 1.5 reaches state q_1, and so a cycle containing q_1 has been found. □

The running time of the algorithm can be easily determined. The first DFS requires $O(|Q| + |\delta|)$ time. During the searches of the second phase each transition is explored at most once, and so they can be executed together in $O(|Q| + |\delta|)$ time.

Nesting the two searches

Recall that we are looking for algorithms that return an accepting lasso when A is nonempty. The algorithm we have described is not good for this purpose. Define the *DFS-path* of a state as the unique path of the DFS-tree leading from the initial state to it. When the second phase answers NONEMPTY, the DFS-path of the state being currently explored, say q, is an accepting cycle, but

[2]Notice that this does not require to apply any sorting algorithm, it suffices to output an accepting state immediately after blackening it.

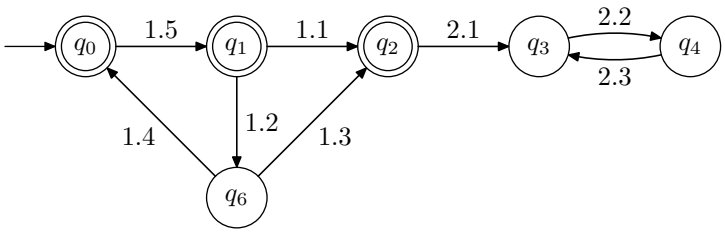


Figure 13.2: The transitions explored during the search starting at q_i are labelled by the index i. The search starting at q_1 stops with NONEMPTY.

usually not an accepting lasso. For an accepting lasso we can prefix this path with the DFS-path of q obtained during the first phase. However, since the first phase cannot foresee the future, it does not know which accepting state, if any, will be identified by the second phase as belonging to an accepting lasso. So either the first search must store the DFS-paths of *all* the accepting states it discovers, or a third phase is necessary, in which a new DFS-path is recomputed.

This problem can be solved by *nesting* the first and the second phases: Whenever the first DFS blackens an accepting state q, we immediately launch a second DFS to check if q is reachable from itself. We obtain the nested-DFS algorithm, due to Courcoubetis, Vardi, Wolper, and Yannakakis:

- Perform a DFS from q_0.
- Whenever the search blackens an accepting state q, launch a new DFS from q. If this second DFS visits q again (i.e., if it explores some transition leading to q), stop with NONEMPTY. Otherwise, when the second DFS terminates, continue with the first DFS.
- If the first DFS terminates, output EMPTY.

A pseudocode implementation is shown below; for clarity, the program on the left does not include the instructions for returning an accepting lasso. A variable *seed* is used to store the state from which the second DFS is launched. The instruction **report** X produces the output X and stops the execution. The set S is usually implemented by means of a hash-table. Notice that it is not necessary to store states $[q, 1]$ and $[q, 2]$ separately. Instead, when a state q is discovered, either during the first or the second searches, then it is stored at the hash address, and two extra bits are used to store which of the following three possibilities hold: only $[q, 1]$ has ben discovered so far, only $[q, 2]$, or both. So, if a state is encoded by a bitstring of length c, then the algorithm needs $c + 2$ bits of memory per state.

NestedDFS(A)
Input: NBA $A = (Q, \Sigma, \delta, q_0, F)$
Output: EMP if $L_\omega(A) = \emptyset$
NEMP otherwise

```
1  S ← ∅
2  dfs1(q0)
3  report EMP
4  proc dfs1(q)
5     add [q, 1] to S
6     for all r ∈ δ(q) do
7        if [r, 1] ∉ S then dfs1(r)
8     if q ∈ F then { seed ← q; dfs2(q) }
9     return
10 proc dfs2(q)
11    add [q, 2] to S
12    for all r ∈ δ(q) do
13       if r = seed then report NEMP
14       if [r, 2] ∉ S then dfs2(r)
15    return
```

NestedDFSwithWitness(A)
Input: NBA $A = (Q, \Sigma, \delta, q_0, F)$
Output: EMP if $L_\omega(A) = \emptyset$
NEMP otherwise

```
1  S ← ∅; succ ← false
2  dfs1(q0)
3  report EMP
4  proc dfs1(q)
5     add [q, 1] to S
6     for all r ∈ δ(q) do
7        if [r, 1] ∉ S then dfs1(r)
8        if succ = true then return [q, 1]
9     if q ∈ F then
10       seed ← q; dfs2(q)
11       if succ = true then return [q, 1]
12    return
13 proc dfs2(q)
14    add [q, 2] to S
15    for all r ∈ δ(q) do
16       if [r, 2] ∉ S then dfs2(r)
17       if succ = true then return [q, 2]
18       if r = seed then
19          report NEMP; succ ← true
20    return
```

The algorithm on the right shows how to modify *NestedDFS* so that it returns an accepting lasso. It uses a global boolean variable *succ* (for success), initially set to false. If at line 11 the algorithm finds that $r = seed$ holds, it sets *success* to true. This causes procedure calls in *dfs1*(q) and *dfs2*(q) to be replaced by **return**$[q, 1]$ and **return**$[q, 2]$, respectively. The lasso is produced in reverse order, i.e., with the initial state at the end.

A small improvement

We show that *dfs2*(q) can already return NONEMPTY if it discovers a state that belongs to the DFS-path of q in `dfs1`. Let q_k be an accepting state. Asssume that *dfs1*(q_0) discovers q_k, and that the

DFS-path of q_k in *dfs1* is $q_0 q_1 \ldots q_{k-1} q_k$. Assume further that *dfs2*(q_k) discovers q_i for some $0 \leq i \leq k-1$, and that the DFS-path of q_i in *dfs2* is $q_k q_{k+1} \ldots q_{k+l} q_i$. Then the path $q_0 q_1 \ldots q_{k-1} q_k \ldots q_{k+l} q_i$ is a lasso, and, since q_k is accepting, it is an accepting lasso. So stopping with NONEMPTY is correct. Implementing this modification requires to keep track during `dfs1` of the states that belong to the DFS-path of the state being currently explored. Notice, however, that we do not need information about their order. So we can use a set P to store the states of the path, and implement P as e.g. a hash table. We do not need the variable *seed* anymore, because the case $r = seed$ is subsumed by the more general $r \in P$.

ImprovedNestedDFS(A)
Input: NBA $A = (Q, \Sigma, \delta, q_0, F)$
Output: EMP if $L_\omega(A) = \emptyset$, NEMP otherwise

```
1  S ← ∅; P ← ∅
2  dfs1(q0)
3  report EMP

4  proc dfs1(q)
5     add [q, 1] to S; add q to P
6     for all r ∈ δ(q) do
7        if [r, 1] ∉ S then dfs1(r)
8     if q ∈ F then dfs2(q)
9     remove q from P
10    return

11 proc dfs2(q)
12    add [q, 2] to S
13    for all r ∈ δ(q) do
14       if r ∈ P then report NEMP
15       if [r, 2] ∉ S then dfs2(r)
16    return
```

Evaluation

The strong point of the the nested-DFS algorithm are its very modest space requirements. Apart from the space needed to store the stack of calls to the recursive *dfs* procedure, the algorithm just needs two extra bits for each state of A. In many practical applications, A can easily have millions or tens of millions of states, and each state may require many bytes of storage. In these cases, the two extra bits per state are negligible.

The algorithm, however, also has two important weak points: It cannot be extended to NGAs, and it is not optimal, in a formal sense defined below. We discuss these two points separately.

The nested-DFS algorithm works by identifying the accepting states first, and then checking if they belong to some cycle. This principle no longer works for the acceptance condition of NGAs, where we look for cycles containing at least one state *of each family* of accepting states. No better procedure than translating the NGA into an NBA has been described so far. For NGAs having a large number of accepting families, the translation may involve a substantial penalty in performance.

A search-based algorithm for emptiness checking explores the automaton A starting from the initial state. At each point t in time, the algorithm has explored a subset of the states and the transitions of the algorithm, which form a sub-NBA $A_t = (Q_t, \Sigma, \delta_t, q_0, F_t)$ of A (i.e., $Q_t \subseteq Q$, $\delta_t \subseteq \delta$, and $F_t \subseteq F$)). Clearly, a search-based algorithm can have only reported NONEMPTY at a time t if A_t contains an accepting lasso. A search-based algorithm is *optimal* if the converse holds, i.e., if it reports NONEMPTY at the earliest time t such that A_t contains an accepting lasso. It is easy to see that *NestedDFS* is not optimal. Consider the automaton on top of Figure 13.5. Initially,

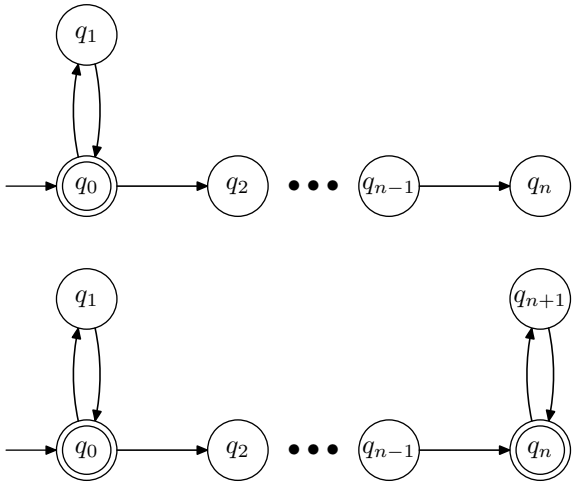


Figure 13.3: Two bad examples for *NestedDFS*

the algorithm chooses between the transitions (q_0, q_1) and (q_0, q_2). Assume it chooses (q_0, q_1) (the algorithm does not know that there is a long tail behind q_2). The algorithm explores (q_0, q_1) and then (q_1, q_0) at some time t. The automaton A_t already contains an accepting lasso, but, since q_0 has not been blackened yet, *dfs1* continues its execution with (q_0, q_2), and explores *all* transitions before *dfs2* is called for the first time, and reports NONEMPTY. So the time elapsed between the first moment at which the algoritm has enough information to report NONEMPTY, and the moment at which the report occurs, can be arbitrarily large.

The automaton at the bottom of Figure 13.5 shows another problem of *NestedDFS* related to non-optimality. If it selects (q_0, q_1) as first transition, then, since q_n precedes q_0 in postorder, *dfs2(q_n)* is executed before *dfs2(q_0)*, and it succeeds, reporting the lasso $q_0 q_2 \ldots q_n q_{n+1} q_n$, instead of the much shorter lasso $q_0 q_1 q_0$.

In the next section we describe an optimal algorithm that can be easily extended to NGAs. The price to pay is a higher memory consumption. As we shall see, the new algorithm needs to assign a number to each state, and store it (apart from maintaining other data structures).

13.1.2 The two-stack algorithm

Recall that the nested-DFS algorithm searches for accepting states of A, and then checks if they belong to some cycle. The two-stack algorithm proceeds the other way round: It searches for states that belong to some cycle of A by means of a single DFS, and checks whether they are accepting.

A first observation is that by the time the DFS blackens a state, it already has explored enough to decide whether it belongs to a cycle:

Lemma 13.6 *Let A_t be the sub-NBA of A containing the states and transitions explored by the DFS up to (and including) time t. If a state q belongs to some cycle of A, then it already belongs to some cycle of $A_{f[q]}$.*

Proof: Let π be a cycle containing q, and consider the snapshot of the DFS at time $f[q]$. Let r be the last state of π after q such that all sttaes in the subpath from q to r are black. We have $f[r] \leq f[q]$. If $r = q$, then π is a cycle of $A_{f[q]}$, and we are done. If $r \neq q$, let s be the successor of r in π (see Figure 13.4). We have $f[r] < f[q] < f[s]$. Moreover, since all successors of r have

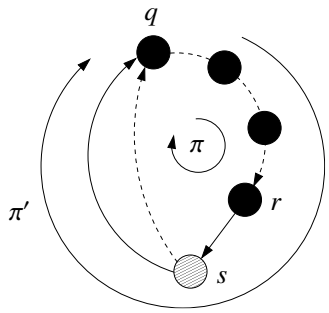


Figure 13.4: Illustration of the proof of Lemma 13.6

necessarily been discovered at time $f[r]$, we have $d[s] < f[r] < f[q] < f[s]$. By the Parenthesis theorem, s is a DFS-ascendant of q. Let π' be the cycle obtained by concatenating the DFS-path from s to q, the prefix of π from q to r, and the transition (r, s). By the Parenthesis Theorem, all the transitions in this path have been explored at time $f[q]$, and so the cycle belongs to $A_{f[q]}$ □

This lemma suggests to maintain during the DFS a set C of *candidates*, containing the states for which it is not yet known whether they belong to some cycle or not. A state is added to C when it is discovered. While the state is grey, the algorithm tries to find a cycle containing it. If it succeeds, then the state is removed from C. If not, then the state is removed from C when it is blackened. At any time t, the candidates are the currently grey states that do not belong to any cycle of A_t.

Assume that at time t the set C indeed contains the current set of candidates, and that the DFS explores a new transition (q, r). We need to update C. If r has not been discovered yet (i.e., if it does not belong to A_t), the addition of r and (q, r) to A_t does not create any new cycle, and the update just adds r to C. If r belongs to A_t but no path of A_t leads from r to q, again no new cycle is created, and the set C does not change. But if r belongs to A_t, and $r \rightsquigarrow q$ then the addition of (q, r) does create new cycles. Let us assume we can ask an oracle whether $r \rightsquigarrow q$ holds, and the oracle answers 'yes'. Then we have already learnt that both q and r belong to some cycle of A, and so both of them must be removed from C. However, we may have to remove other states as well. Consider the DFS of Figure 13.5: after adding (q_4, q_1) to the set of explored transitions at time 5, all of q_1, q_2, q_3, q_4 belong to a cycle. The fact that these are the states discovered by the DFS between the

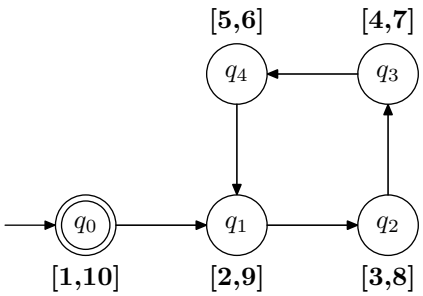


Figure 13.5: A DFS on an automaton

discoveries of q_1 and q_4 suggests to implement C using a *stack*: By pushing states into C when they are discovered, and removing when they are blackened or earlier (if some cycle contains them), the update to C after exploring a transition (q, r) can be performed very easily: it suffices to pop from C until r is hit (in Figure 13.5 we pop q_4, q_3, q_2, and q_1). Observe also that removing q from C when it is blackened does not require to inspect the complete stack; since every state is removed *at the latest* when it is blackened, if q has not been removed yet, then it is necessarily at the top of C. (This is the case of state q_0 in Figure 13.5). So it suffices to inspect the top of the stack: if q is at the top, we pop it; otherwise q is not in the stack, and we do nothing.

This leads to our first attempt at an algorithm, shown on the top left corner of Figure 13.6. When a state q is discovered it is pushed into C (line 5), and its successors explored (lines 6-12). When exploring a successor r, if r has not been discovered yet then *dfs*(r) is called (line 7). Otherwise the oracle is consulted (line 8), and if $r \rightsquigarrow q$ holds at the time (i.e., in the part of A explored so far), then states are popped from C until r is hit (lines 9-12). Then the algorithm checks if q has already been removed by inspecting the top of the stack (line 13), and removes q if that is the case.

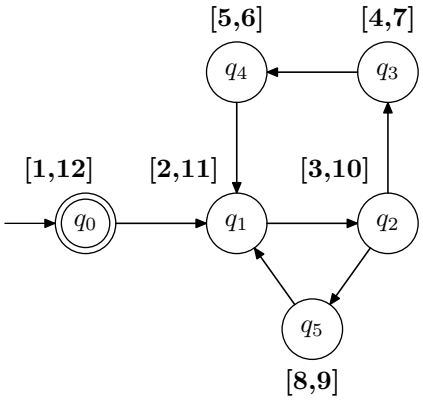
The NBA below *FirstAttempt* shows, however, that the algorithm needs to be patched. After exploring (q_4, q_1) the states q_4, q_3, q_2, q_1 are popped from C, in that order, and C contains only q_0. Now the DFS backtracks to state q_3, and explores (q_3, q_5), pushing q_5 into the stack. Then the DFS explores (q_5, q_1), and pops from C until it hits q_1. But this leads to an incorrect result, because, since q_1 no longer belongs to C, the algorithm pops all states from C, and when it pops q_0 it reports

FirstAttempt(A)
Input: NBA $A = (Q, \Sigma, \delta, q_0, F)$
Output: EMP if $L_\omega(A) = \emptyset$
NEMP otherwise

```
1  S, C ← ∅;
2  dfs(q0)
3  report EMP
4  proc dfs(q)
5     add q to S; push(q, C)
6     for all r ∈ δ(q) do
7        if r ∉ S then dfs(r)
8        else if r ⇝ q then
9           repeat
10             s ← pop(C)
11             if s ∈ F then report
NEMP
12          until s = r
13    if top(C) = q then pop(C)
```

SecondAttempt(A)
Input: NBA $A = (Q, \Sigma, \delta, q_0, F)$
Output: EMP if $L_\omega(A) = \emptyset$
NEMP otherwise

```
1  S, C ← ∅;
2  dfs1(q0)
3  report EMP
4  proc dfs(q)
5     add q to S; push(q, C)
6     for all r ∈ δ(q) do
7        if r ∉ S then dfs(r)
8        else if r ⇝ q then
9           repeat
10             s ← pop(C)
11             if s ∈ F then report
NEMP
12          until s = r
13          push(r, C)
14    if top(C) = q then pop(C)
```



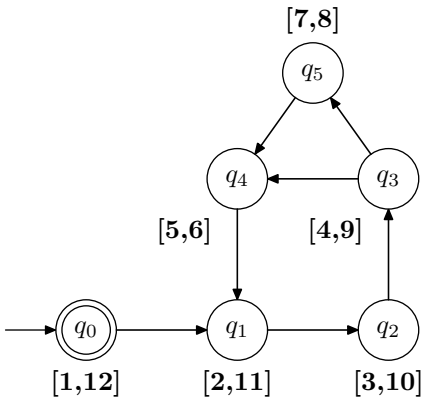


Figure 13.6: Two incorrect attempts at an emptiness checking algorithm

NONEMPTY.

This problem can be solved as follows: If the DFS explores (q, r) and $r \rightsquigarrow q$ holds, then it pops from C until r is popped, *and pushes r back into the stack*. This second attempt is shown at the top right of the figure. However, the NBA below *SecondAttempt* shows that it is again incorrect. After exploring (q_4, q_1) (with stack content $q_4q_3q_2q_1q_0$), the states q_4, q_3, q_2, q_1 are popped from C, in that order, and q_1 is pushed again. C contains now q_1q_0. The DFS explores (q_3, q_5) next, pushing q_5, followed by (q_5, q_4). Since $q_4 \rightsquigarrow q_5$ holds, the algorithm pops from C until q_4 is found. But, again, since q_4 does not belong to C, the result is incorrect.

A patch for this problem is to change the condition of the repeat loop: If the DFS explores (q, r) and $r \rightsquigarrow q$ holds, we cannot be sure that r is still in the stack. So we pop until either *r or some state discovered before r is hit*, and then we push this state back again. In the example, after exploring (q_5, q_4) with stack content $q_5q_1q_0$, the algorithm pops q_5 and q_1, and then pushes q_1 back again. This new patch leads to the *OneStack* algorithm:

OneStack(A)
Input: NBA $A = (Q, \Sigma, \delta, q_0, F)$
Output: EMP if $L_\omega(A) = \emptyset$, NEMP otherwise

```
1  S, C ← ∅;
2  dfs(q0)
3  report EMP
4  dfs(q)
5     add q to S; push(q, C)
6     for all r ∈ δ(q) do
7        if r ∉ S then dfs(r)
8        else if r ⇝ q then
9           repeat
10             s ← pop(C); if s ∈ F then report NEMP
11          until d[s] ≤ d[r]
12          push(s, C)
13    if top(C) = q then pop(C)
```

Example 13.7 Figure 13.7 shows a run of *OneStack* on the NBA shown at the top. The NBA has no accepting states, and so it is empty. However, during the run we will see how the algorithm answers NONEMPTY (resp. EMPTY) when f (res. h) is the only accepting state. The discovery and finishing times are shown at the top. Observe that the NBA has three sccs: $\{a, b, e, f, g\}$, $\{h\}$, and $\{c, d, i, j\}$, with roots a, h, and i, respectively.

Below the NBA at the top, the figure shows different snapshots of the run of *OneStack*. At each snapshot, the current grey path is shown in red/pink. The dotted states and transitions have not been discovered yet, while dark red states have already been blackened. The current content of stack C is shown on the right (ignore stack L for the moment).

- The first snapshot is taken immediately before state j is blackened. The algorithm has just explored the transition (j, i), has popped the states c, d, j, i from C, and has pushed state i back. The states c, d, j, i have been identified as belonging to some cycle.
- The second snapshot is taken immediately after state i is blackened. State i has been popped from C at line 13. Observe that after this the algorithm backtracks to $dfs(h)$, and, since state h is at the top of the stack, it pops h from C at line 13. So h is never popped by the repeat loop, and so even if h is accepting the algorithm does not report NONEMPTY.
- The third snapshot is taken immediately before state f is blackened. The algorithm has just explored the transition (f, a), has popped the states f, g, b, a from C, and has pushed state a back. The states f, g, b, a have been identified as belonging to some cycle. If state f is accepting, at this point the algorithm reports EMPTY and stops.
- The fourth snapshot is taken immediately after state e is discovered. The state has been pushed into C.
- The final snapshot is taken immediately before a is blackened and the run terminates. State a is going to be removed from C at line 13, and the run terminates.

Observe that when the algorithm explores transition (b, c) it calls the oracle, which answers $c \not\rightsquigarrow b$, and no states are popped from C.

□

The algorithm looks now plausible, but we still must prove it correct. We have two proof obligations:

- If A is nonempty, then *OneStack* reports NONEMPTY. This is equivalent to: every state that belongs to some cycle is eventually popped during the repeat loop.
- If *OneStack* reports NONEMPTY, then A is nonempty. This is equivalent to: every state popped during the repeat loop belongs to some cycle.

These properties are shown in Propositions 13.8 and 13.11 below.

Proposition 13.8 *If q belongs to a cycle, then q is eventually popped by the repeat loop.*

Proof: Let π be a cycle containing q, let q' be the last successor of q along π such that at time $d[q]$ there is a white path from q to q', and let r be the successor of q' in π. Since r is grey or black at time $d[q]$, we have $d[r] \leq d[q] \leq d[q']$. By the White-path Theorem, q' is a descendant of q, and so the transition (q', r) is explored before q is blackened. So when (q', r) is explored, q has not been popped at line 13. Since $r \rightsquigarrow q'$, either q has already been popped by at some former execution of the repeat loop, or it is popped now, because $d[r] \leq d[q']$. □

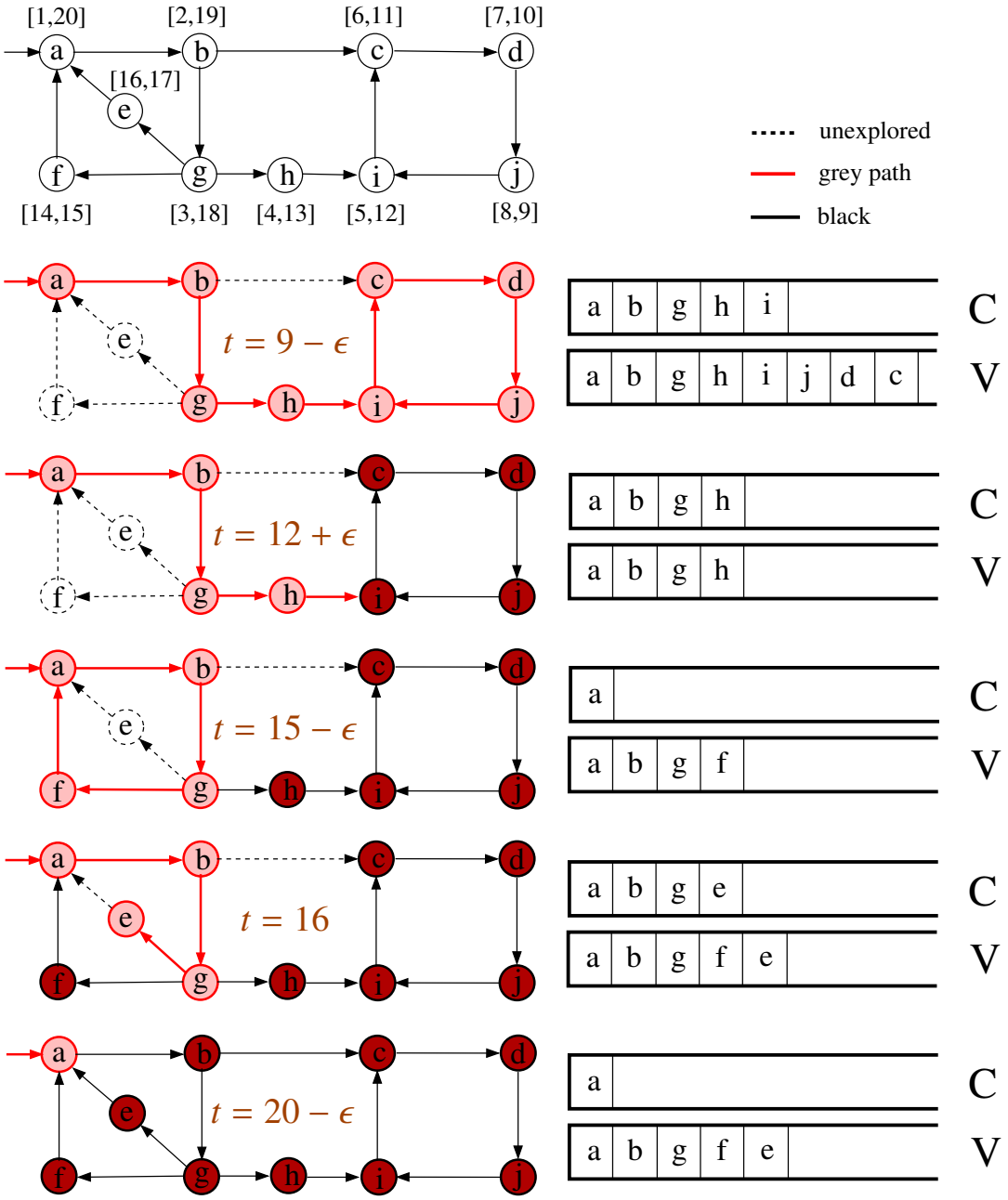


Figure 13.7: A run of *OneStack*

Actually, the proof of Proposition 13.8 proves not only that q is eventually popped by the repeat loop, but also that for every cycle π containing q, the repeat loop pops q immediately after all transitions of π have been explored, or earlier. But this is precisely the optimality property, which leads to:

Corollary 13.9 *OneStack is optimal.*

The property that every state popped during the repeat loop belongs to some cycle has a more involved proof. A *strongly connected component* (*scc*) of A is a maximal set of states $S \subseteq Q$ such that $q \rightsquigarrow r$ for every $q, r \in S$.[3] The first state of a reachable scc that is discovered by the DFS is called the *root* of the scc (with respect to this DFS). In other words, if q is the root of an scc, then $d[q] \leq d[r]$ for every state r of the scc. The following lemma states an important invariant of the algorithm. If a root belongs to C at line 9, before the repeat loop is executed, then it still belongs to C after the loop finishes and the last popped state is pushed back. So, loosely speaking, the repeat loop cannot remove a root from the stack; more precisely, if the loop removes a root, then the push instruction at line 12 reintroduces it again.

Lemma 13.10 *Let ρ be a root. If ρ belongs to C before an execution of the repeat loop at lines 9-11, then all states s popped during the execution of the loop satisfy $d[\rho] \leq d[s]$, and ρ still belongs to C after the repeat loop has terminated and line 12 has been executed.*

Proof: Let t be the time right before an execution of the repeat loop starts at line 9, and assume ρ belongs to C at time t. Since states are removed from C when they are blackened or earlier, ρ is still grey at time t. Since $r \in \delta(q)$ (line 6) and $r \rightsquigarrow q$ (line 8), both q and r belong to the same scc. Let ρ' be the root of this scc. Since ρ' is also grey at time t, and grey states always are a path of the DFS-tree, either ρ is a DFS-ascendant of ρ', or $\rho' \neq \rho$ and ρ' is a DFS-ascendant of ρ. In the latter case we have $\rho' \rightsquigarrow \rho \rightsquigarrow q$, contradicting that ρ' is the root of q's scc. So ρ is a DFS-ascendant of ρ', and so in particular we have $d[\rho] \leq d[\rho'] \leq d[r]$. Since states are added to C when they are discovered, the states of C are always ordered by decreasing discovery time, starting from the top, and so every state s popped before ρ satisfies $d[\rho] \leq d[s]$. If ρ is popped, then, since $d[\rho] \leq d[r]$, the execution of the repeat loop terminates, and ρ is pushed again at line 12. □

Proposition 13.11 *Any state popped during the repeat loop (at line 10) belongs to some cycle.*

Proof: Consider the time t right before a state s is about to be popped at line 10 while the for-loop (lines 6-12) is exploring a transition (q, r). (Notice that the body of the for-loop may have already been executed for other transitions (q, r')). Since the algorithm has reached line 10, we have $r \rightsquigarrow q$ (line 8), and so both q and r belong to the same scc of A. Let ρ be the root of this scc. We show that s belongs to a cycle.

[3] Notice that a path consisting of just a state q and no transitions is a path leading from q to q.

(1) s is a DFS-ascendant of q.
Since s belongs to C at time t, both s and q are grey at time t, and so they belong to the current path of grey states. Moreover, since $dfs(q)$ is being currently executed, q is the last state in the path. So s is a DFS-ascendant of q.

(2) ρ is a DFS-ascendant of s.
Since ρ is a root of the scc of q, at time $d[\rho]$ there is a white path from ρ to q. By the White-path and Parenthesis Theorem, ρ is a DFS-ascendant of q. Together with (1), this implies that either ρ is a DFS-ascendant of s or s is a DFS-ascendant of ρ. By Lemma 13.10 we have $d[\rho] \leq d[s]$, and so by the Parenthesis Theorem ρ is a DFS-ascendant of s.

By (1), (2), and $r \rightsquigarrow q$, we have $\rho \rightsquigarrow s \rightsquigarrow q \rightsquigarrow r \rightsquigarrow \rho$, and so s belongs to a cycle. □

Implementing the oracle

Recall that *OneStack* calls an oracle to decide at time t if $r \rightsquigarrow q$ holds. At first sight the oracle seems difficult to implement. We show that this is not the case.

Assume that *OneStack* calls the oracle at line 8 at some time t. We look for a condition that holds at time t if and only if $r \rightsquigarrow q$, and is easy to check.

Lemma 13.12 *Assume that OneStack(A) is currently exploring a transition (q, r), and the state r has already been discovered. Let R be the scc of A satisfying $r \in R$. Then $r \rightsquigarrow q$ iff some state of R is not black.*

Proof: Assume $r \rightsquigarrow q$. Then r and q belong to R, and since q is not black because (q, r) is being explored, R is not black.

Assume $r \not\rightsquigarrow q$. We consider the colors of the states at the time (q, r) is explored, and show that all the states of R are black. We proceed by contradiction. Assume some state of R is not black. Not all states of R are white because r has already been discovered, and so at least one state $s \in R$ is grey. Since grey states form a path ending at the state whose output transitions are being currently explored, the grey path contains s and ends at q. So $s \rightsquigarrow q$, and, since s and r belong to R, we have $r \rightsquigarrow q$, contradicting the hypothesis. □

By Lemma 13.12, checking if $r \rightsquigarrow q$ holds amounts to checking if all states of R are black or not. This can be done as follows: we maintain a set V of *actiVe* states, where a state is *active* if its scc has not yet been completely explored, i.e., if some state of the scc is not black. Then, checking $r \rightsquigarrow q$ reduces to checking whether r is active. The set V can be maintained by adding a state to it whenever it is discovered, and removing all the states of a scc right after the last of them is blackened. The next lemma shows that the last of them is always the root:

Lemma 13.13 *Let ρ be a root, and let q be a state such that $\rho \rightsquigarrow q \rightsquigarrow \rho$. Then $I(q) \subseteq I(r)$. (In words: The root is the first state of a scc to be grayed, and the last to be blackened)*

Proof: By the definition of a root, at time $d[\rho]$ there is a white path from ρ to q. By the White-path and the Parenthesis Theorems, $I(q) \subseteq I(r)$. □

By this lemma, in order to maintain V it suffices to remove all the states of an scc whenever its root is blackened. So whenever the DFS blackens a state q, we have to perform two tasks: (1) check if q is a root, and (2) if q is a root, remove all the states of q's scc. Checking if q is a root is surprisingly simple:

Lemma 13.14 *When OneStack executes line 13, q is a root if and only if* $\mathbf{top}(C) = q$.

Proof: Assume q is a root. By Lemma 13.10, q still belongs to C after the for loop at lines 6-12 is executed, and so $\mathbf{top}(C) = q$ at line 13.

Assume now that q is not a root. Then there is a path from q to the root ρ of q's scc. Let r be the first state in the path satisfying $d[r] < d[q]$, and let q' be the predecessor of r in the path. By the White-path theorem, q' is a descendant of q, and so when transition (q, r) is explored, q is not yet black. When *OneStack* explores (q', r), it pops all states s from C satisfying $d[s] > d[r]$, and none of these states is pushed back at line 12. In particular, either *OneStack* has already removed q from C, or it removes it now. Since q has not been blackened yet, when *OneStack* executes line 14 for $dfs(q)$, the state q does not belong to C and in particular $q \neq \mathbf{top}(C)$ □

Tasks (2) can be performed very elegantly by implementing V as a second stack, and maintaining it as follows:

- when a state is discovered (greyed), it is pushed into the stack (so states are always ordered in V by increasing discovery time); and
- when a root is blackened, all states of V above it (including the root itself) are popped.

Example 13.15 Figure 13.7 shows the content of L at different times when the policy above is followed. Right before state j is blackened, L contains all states in the grey path. When the root i is blackened, all states above it (including i itself), are popped; these are the states c, d, j, i, which form a scc. State h is also a root, and it is also popped. Then, states f and e are discovered, and pushed into V. Finally, when the root a is blackened, states c, f, g, b, a are popped, which correspond to the third and last scc. □

We show that the states popped when ρ is blackened are exactly those that belong to ρ's scc.

Lemma 13.16 *The states popped from V right after blackening a root ρ are exactly those belonging to ρ's scc.*

Proof: Let q be a state of ρ's scc. Since ρ is a root, we have $d[\rho] \leq d[q]$, and so q lies above ρ in L. So q is popped when ρ is blackened, unless it has been popped before. We show that this cannot be the case. Assume q is popped before ρ is blackened, i.e., when some other root $\rho' \neq \rho$ is blackened at time $f[\rho']$. We collect some facts: (a) $d[\rho] \leq d[q] \leq d[q] \leq f[\rho]$ by Lemma 13.13; (b) $d[\rho'] \leq d[q] < f[\rho']$, because q is in the stack at time $f[\rho']$ (implying $d[q] < f[\rho']$), and it is above ρ' in the stack (implying $d[\rho'] \leq d[q]$); (c) $f[\rho'] < f[\rho]$, because q has not been popped yet at time $f[\rho']$, and so ρ cannot have been blackened yet. From (a)-(c) and the Parenthesis Theorem we get $I(q) \subseteq I(\rho') \subseteq I(\rho)$, and so in particular $\rho \rightsquigarrow \rho' \rightsquigarrow q$. But then, since $q \rightsquigarrow \rho$, we get $\rho \rightsquigarrow \rho' \rightsquigarrow \rho$, contradicting that ρ and ρ' are *different* roots, and so belong to different sccs. □

This finally leads to the two-stack algorithm

```
TwoStack(A)
Input: NBA A = (Q, Σ, δ, q_0, F)
Output: EMP if L_ω(A) = ∅, NEMP otherwise
 1  S, C, V ← ∅;
 2  dfs(q_0)
 3  report EMP
 4  proc dfs(q)
 5     add q to S; push(q, C); push(q, V)
 6     for all r ∈ δ(q) do
 7        if r ∉ S then dfs(r)
 8        else if r ∈ V then
 9           repeat
10              s ← pop(C); if s ∈ F then report NEMP
11           until d[s] ≤ d[r]
12           push(s, C)
13     if top(C) = q then
14        pop(C)
15        repeat s ← pop(V) until s = q
```

The changes with respect to *OneStack* are shown in blue. The oracle $r \rightsquigarrow q$ is replaced by $r \in V$ (line 8). When the algorithm blackens a root (line 13), it pops from V all elements above q, and q itself (line 15).

Observe that V cannot be implemented *only* as a stack, because at line 8 we have to check if a state belongs to the stack or not. The solution is to implement V both as a stack *and* use an additional bit in the hash table for S to store whether the state belongs to V or not, which is possible because $V \subseteq S$ holds at all times. The check at line 8 is performed by checking the value of the bit.

Extension to GBAs

We show that *TwoStack* can be easily transformed into an emptiness check for generalized Büchi automata that does not require to construct an equivalent NBA. Recall that a NGA has in general several sets $\{F_0, \ldots, F_{k-1}\}$ of accepting states, and that a run ρ is accepting if $\inf \rho \cap F_i \neq \emptyset$ for every $i \in \{0, \ldots, k-1\}$. So we have the following characterization of nonemptiness, where $K = \{0, \ldots, k-1\}$:

Fact 13.17 *Let A be a NGA with accepting condition $\{F_0, \ldots, F_{k-1}\}$. A is nonempty iff some scc S of A satisfies $S \cap F_i \neq \emptyset$ for every $i \in K$.*

Since every time the repeat loop at line 15 is executed, it pops from L one scc, we can easily check this condition by modifying line 15 accordingly. However, the resulting algorithm would not be optimal, because the condition is not checked until the scc has been completely explored. To solve this problem, we have a closer look at Proposition 13.11. The proof shows that a state s popped at line 10 belongs to the ssc of state r. So, in particular, all the states popped during an execution of the repeat loop at lines 9-11 belong to the same scc. So we collect the set I of indices of the sets of accepting states they belong to, and keep checking whether $I = K$. If so, then we report NONEMPTY. Otherwise, we attach the set I to the state s that is pushed back into the C at line 12. This yields algorithm *TwoStackNGA*, where $F(q)$ denotes the set of all indices $i \in K$ such that $q \in F_i$:

TwoStackNGA(A)
Input: NGA $A = (Q, \Sigma, \delta, q_0, \{F_0, \ldots, F_{k-1}\})$
Output: EMP if $L_\omega(A) = \emptyset$, NEMP otherwise

```
1   S, C, V ← ∅;
2   dfs(q_0)
3   report EMP

4   proc dfs(q)
5     add [q, F(q)] to S; push([q, F(q)], C); push(q, V)
6     for all r ∈ δ(q) do
7       if r ∉ S then dfs(r)
8       else if r ∈ V then
9         I ← ∅
10        repeat
11          [s, J] ← pop(C);
12          I ← I ∪ J; if I = K then report NEMP
13        until d[s] ≤ d[r]
14        push([s, I], C)
15    if top(C) = (q, I)for some I then
16      pop(C)
17      repeat s ← pop(V) until s = q
```

For the correctness of the algorithm, observe that, at every time t, the states of teh subautomaton A_t can be partitioned into strongly connected components, and each of these components has a root. The key invariant for the correctness proof is the following:

Lemma 13.18 *At every time t, the stack C contains a pair* $[q, I]$ *iff q is a root of* A_t*, and I is the subset of indices* $i \in K$ *such that some state of* F_i *belongs to q's scc.*

Proof: Initially the invariant holds because both A_t and C are empty. We show that whenever a new transition (q, r) is explored, *TwoStackNGA* carries out the necessary changes to keep the invariant. Let t be the time immediately after (q, r) is explored. If r is a new state, then it has no successors in A_t, and so it builds an scc of A_t by itself, with root r. Moreover, all roots before the exploration of (q, r) are alsoroots of A_t. So a new pair $[r, F(r)]$ must be added to C, and that is what $dfs(r)$ does. If $r \notin L$, then $r \not\rightsquigarrow \rho$, the addition of (q, r) has not changed the sccs of the automaton explored so far, and nothing must be done. If $r \in L$, then the addition of (q, r) creates new cycles, and some states stop being roots. More precisely, let NR be the set of states of A_t that belong to a cycle containing both q and r, and let ρ be the state of NR with minimal discovery time. Then the algorithm must remove from C all states of NR with the exception of ρ (and no others). We show that this is exactly what the execution of lines 9-14 achieves. By Proposition 13.8 and Corollary 13.9, all the states of $NR \setminus \{\rho\}$ have already been removed at some former execution of the loop, or are removed now at lines 9-14, because they have discovery time smaller than or equal to $d[r]$. It remains to show that all states popped at line 11 belong to NR (that ρ is not removed follows then from the fact that the state with the lowest discovery time is pushed again at line 14, and that state is ρ). For, this, we have a closer look at the proof of Proposition 13.11. The proposition shows not only that the states popped by the repeat loop belong to some cycle, but also that they all belong to cycles that containing q and r (see the last line of the proof), and we are done. □

We can now easily prove:

Proposition 13.19 *TwoStackNGA(A) reports NONEMPTY iff A is nonempty. Moreover, TwoStackNGA is optimal.*

Proof: If *TwoStackNGA*(A) reports NONEMPTY, then the repeat loop at lines 10-13 pops some pair $[q, K]$. By Lemma 13.18, q belongs to a cycle of A containing some state of F_i for every $i \in K$.

If A is nonempty, then some scc S of A satisfies $S \cap F_i \neq \emptyset$ for every $i \in K$. So there is an earliest time t such that A_t contains an scc $S_t \subseteq S$ satisfying the same property. By Lemma 13.18, *TwoStackNGA*(A) reports NONEMPTY at time t or earlier, and so it is optimal. □

Evaluation

Recall that the two weak points of the nested-DFS algorithm were that it cannot be directly extended to NGAs, and it is not optimal. Both are strong points of the two-stack algorithm.

The strong point of the the nested-DFS algorithm were its very modest space requirements: just two extra bits for each state of A. Let us examine the space needed by the two-stack algorithm. It is conveniet to compute it for empty automata, because in this case both the nested-DFS and the two-stack algorithms must visit all states.

Because of the check $d[s] \leq d[r]$, the algorithm needs to store the discovery time of each state. This is done by extending the hash table S. If a state q can be stored using c bits, then $\log n$ bits are needed to store $d[q]$; however, in practice $d[q]$ is stored using a word of memory, because if the number states of A exceeds 2^w, where w is the number of bits of a word, then A cannot be stored in main memory anyway. So the hash table S requires $c + w + 1$ bits per state (the extra bit being the one used to check membership in L at line 8).

The stacks C and L do not store the states themselves, but the memory addresses at which they are stored. Ignoring hashing collisions, this requires $2w$ additional bits per state. For generalized Büchi automata, we must also add the k bits needed to store the subset of K in the second component of the elements of C. So the two-stack algorithm uses a total of $c + 3w + 1$ bits per state ($c + 3w + k + 1$ in the version for NGA), compared to the $c + 2$ bits required by the nested-DFS algorithm. In most cases $w << c$, and so the influence of the additional memory requirements on the performance is small.

13.2 Algorithms based on breadth-first search

In this section we describe algorithms based on breadth-first search (BFS). No linear BFS-based emptiness check is known, and so this section may look at first sight superfluous. However, BFS-based algorithms can be suitably described using operations and checks on sets, which allows us to implement them using automata as data structures. In many cases, the gain obtained by the use of the data structure more than compensates for the quadratic worse-case behaviour, making the algorithms competitive.

Breadth-first search (BFS) maintains the set of states that have been discovered but not yet explored, often called the *frontier* or *boundary*. A BFS from a set Q_0 of states (in this section we consider searches from an arbitrary set of states of A) initializes both the set of discovered states and its frontier to Q_0, and then proceeds in rounds. In a *forward* search, a round explores the *outgoing* transitions of the states in the current frontier; the new states found during the round are added to the set of discovered states, and they become the next frontier. A *backward* BFS proceeds similarly, but explores the *incoming* instead of the outgoing transitions. The pseudocode implementations of both BFS variants shown below use two variables S and B to store the set of discovered states and the boundary, respectively. We assume the existence of oracles that, given the current boundary B, return either $\delta(B) = \bigcup_{q \in B} \delta(q)$ or $\delta^{-1}(B) = \bigcup_{q \in B} \delta^{-1}(q)$.

ForwardBFS$[A](Q_0)$
Input: NBA $A = (Q, \Sigma, \delta, q_0, F)$, $Q_0 \subseteq Q$

1 $S, B \leftarrow Q_0$;
2 **repeat**
3 $B \leftarrow \delta(B) \setminus S$
4 $S \leftarrow S \cup B$
5 **until** $B = \emptyset$

BackwardBFS$[A](Q_0)$
Input: NBA $A = (Q, \Sigma, \delta, q_0, F)$, $Q_0 \subseteq Q$

1 $S, B \leftarrow Q_0$;
2 **repeat**
3 $B \leftarrow \delta^{-1}(B) \setminus S$
4 $S \leftarrow S \cup B$
5 **until** $B = \emptyset$

Both BFS variants compute the successors or predecessors of a state exactly once, i.e., if in the course of the algorithm the oracle is called twice with arguments B_i and B_j, respectively, then $B_i \cap B_j = \emptyset$. To prove this in the forward case (the backward case is analogous), observe that $B \subseteq S$ is an invariant of the repeat loop, and that the value of S never decreases. Now, let $B_1, S_1, B_2, S_2, \ldots$ be the sequence of values of the variables B and S right before the i-th execution of line 3. We have $B_i \subseteq S_i$ by the invariant, $S_i \subseteq S_j$ for every $j \geq i$, and and $B_{j+1} \cap S_j = \emptyset$ by line 3. So $B_j \cap B_i = \emptyset$ for every $j > i$.

As data structures for the sets S and B we can use a hash table and a queue, respectively. But we can also take the set Q of states of A as finite universe, and use automata for fixed-length languages to represent both S and B. Moreover, we can represent $\delta \subseteq Q \times Q$ by a finite transducer T_δ, and reduce the computation of $\delta(B)$ and $\delta^{-1}(B)$ in line 3 to computing $\mathbf{Post}(B, \delta)$ and $\mathbf{Pre}(B, \delta)$, respectively.

13.2.1 Emerson-Lei's algorithm

A state q of A is *live* if some infinite path starting at q visits accepting states infinitely often. Clearly, A is nonempty if and only if its initial state is live. We describe an algorithm due to Emerson and Lei for computing the set of live states. For every $n \geq 0$, the *n-live* states of A are inductively defined as follows:

- every state is 0-live;
- a state q is $(n + 1)$-live if some path containing at least one transition leads from q to an accepting n-live state.

Loosely speaking, a state is n-live if starting at it it is possible to visit accepting states n-times. Let $L[n]$ denote the set of n-live states of A. We have:

Lemma 13.20 *(a) $L[n] \supseteq L[n + 1]$ for every $n \geq 0$.*

(b) The sequence $L[0] \supseteq L[1] \supseteq L[2] \ldots$ reaches a fixpoint $L[i]$ (i.e., there is a least index $i \geq 0$ such that $L[i + 1] = L[i]$), and $L[i]$ is the set of live states.

Proof: We prove (a) by induction on n. The case $n = 0$ is trivial. Assume $n > 0$, and let $q \in L[n+1]$. There is a path containing at least one transition that leads from q to an accepting state $r \in L[n]$. By induction hypothesis, $r \in L[n-1]$, and so $q \in L[n]$.

To prove (b), first notice that, since Q is finite, the fixpoint $L[i]$ exists. Let L be the set of live states. Clearly, $L \subseteq L[i]$ for every $i \geq 0$. Moreover, since $L[i] = L[i+1]$, every state of $L[i]$ has a proper descendant that is accepting and belongs to $L[i]$. So $L[i] \subseteq L$. □

Emerson-Lei's algorithm computes the fixpoint $L[i]$ of the sequence $L[0] \supseteq L[1] \supseteq L[2] \ldots$. To compute $L[n+1]$ given $L[n]$ we observe that a state is $n+1$-live if some nonempty path leads from it to an n-live accepting state, and so

$$L[n+1] = \mathit{BackwardBFS}(\ \mathbf{Pre}(L[n] \cap F)\ )$$

The pseudocode for the algorithm is shown below on the left-hand-side; the variable L is used to store the elements of the sequence $L[0], L[1], L[2], \ldots$.

EmersonLei(A)
Input: NBA $A = (Q, \Sigma, \delta, q_0, F)$
Output: EMP if $L_\omega(A) = \emptyset$,
NEMP otherwise

```
1  L ← Q
2  repeat
3     OldL ← L
4     L ← Pre(OldL ∩ F)
5     L ← BackwardBFS(L)
6  until L = OldL
7  if q0 ∈ L then report NEMP
8  else report NEMP
```

EmersonLei2(A)
Input: NBA $A = (Q, \Sigma, \delta, q_0, F)$
Output: EMP if $L_\omega(A) = \emptyset$,
NEMP otherwise

```
1  L ← Q
2  repeat
3     OldL ← L
4     L ← Pre(OldL ∩ F) \ OldL
5     L ← BackwardBFS(L) ∪ OldL
6  until L = OldL
7  if q0 ∈ L then report NEMP
8  else report NEMP
```

The repeat loop is executed at most $|Q| + 1$-times, because each iteration but the last one removes at least one state from L. Since each iteration takes $O(|Q| + |\delta|)$ time, the algorithm runs in $O(|Q| \cdot (|Q| + |\delta|))$ time.

The algorithm may compute the predecessors of a state twice. For instance, if $q \in F$ and there is a transition (q, q), then after line 4 is executed the state still belongs to L. The version on the right avoids this problem.

Emerson-Lei's algorithm can be easily generalized to NGAs (we give only the generalization of the first version):

GenEmersonLei(A)
Input: NGA $A = (Q, \Sigma, \delta, q_0, \{F_0, \ldots, F_{m-1}\})$
Output: EMP if $L_\omega(A) = \emptyset$,
NEMP otherwise

```
1  L ← Q
2  repeat
3     OldL ← L
4     for i=0 to m − 1
5        L ← Pre(OldL ∩ F_i)
6        L ← BackwardBFS(L)
7  until L = OldL
8  if q_0 ∈ L then report NEMP
9  else report NEMP
```

Proposition 13.21 *GenEmersonLei(A) reports NEMP iff A is nonempty.*

Proof: For every $k \geq 0$, redefine the n-live states of A as follows: every state is 0-live, and q is $(n+1)$-live if some path having at least one transition leads from q to a n-live state of $F_{(n \bmod m)}$. Let $L[n]$ denote the set of n-live states. Proceeding as in Lemma 13.20, we can easily show that $L[(n+1) \cdot m] \supseteq L[n \cdot m]$ holds for every $n \geq 0$.

We claim that the sequence $L[0] \supseteq L[m] \supseteq L[2 \cdot m] \ldots$ reaches a fixpoint $L[i \cdot m]$ (i.e., there is a least index $i \geq 0$ such that $L[(i+1) \cdot m] = L[i \cdot m]$), and $L[i \cdot m]$ is the set of live states. Since Q is finite, the fixpoint $L[i \cdot m]$ exists. Let q be a live state. There is a path starting at q that visits F_j infinitely often for every $j \in \{0, \ldots, m-1\}$. In this path, every occurrence of a state of F_j is always followed by some later occurrence of a state of $F_{(j+1) \bmod m}$, for every $i \in \{0, \ldots, m-1\}$. So $q \in L[i \cdot m]$. We now show that every state of $L[i \cdot m]$ is live. For every state $q \in L[(i+1) \cdot m]$ there is a path $\pi = \pi_{m-1}\pi_{m-2}\pi_0$ such that for every $j \in \{0, \ldots, m-1\}$ the segment π_j contains at least one transition and leads to a state of $L[i \cdot m + j] \cap F_j$. In particular, π visits states of $F_0, \ldots, F_{m-1}$, and, since $L[(i+1) \cdot m] = L[i \cdot m]$, it leads from a state of $L[(i+1) \cdot m]$ to another state of $L[(i+1) \cdot m]$. So every state of $L[(i+1) \cdot m] = L[i \cdot m]$ is live, which proves the claim.

Since *GenEmersonLei*(A) computes the sequence $L[0] \supseteq L[m] \supseteq L[2 \cdot m] \ldots$, after termination L contains the set of live states. □

13.2.2 A Modified Emerson-Lei's algorithm

There exist many variants of Emerson-Lei's algorithm that have the same worst-case complexity, but try to improve the efficiency, at least in some cases, by means of heuristics. We present here one of these variants, which we call the Modified Emerson-Lei's algorithm (MEL).

Given a set $S \subseteq Q$ of states, let $inf(S)$ denote the states $q \in S$ such that some infinite path starting at q contains only states of S. Instead of computing $\mathbf{Pre}(OldL \cap F)$ at each iteration step, MEL computes $\mathbf{Pre}(inf(OldL) \cap F_i)$.

MEL(*A*)
Input: NGA $A = (Q, \Sigma, \delta, q_0, \{F_0, \ldots, F_{k-1}\})$
Output: EMP if $L_\omega(A) = \emptyset$, NEMP otherwise

```
1  L ← Q;
2  repeat
3     OldL ← L
4     L ← inf(OldL)
5     L ← Pre(L ∩ F)
6     L ← BackwardBFS(L)
7  until L = OldL
8  if q0 ∈ L then report NEMP
9  else report NEMP

10 function inf(S)
11    repeat
12       OldS ← S
13       S ← S ∩ Pre(S)
14    until S = OldS
15    return S
```

In the following we show that MEL is correct, and then compare it with Emerson-Lei's algorithm. As we shall see, while MEL introduces the overhead of repeatedly computing *inf*-operations, it still makes sense in many cases because it reduces the number of executions of the repeat loop.

To prove correctness we claim that after termination L contains the set of live states. Recall that the set of live states is the fixpoint $L[i]$ of the sequence $L[0] \supseteq L[1] \supseteq L[2] \ldots$. By the definition of liveness we have $inf(L[i]) = L[i]$. Define now $L'[0] = Q$, and $L'[n+1] = inf(pre^+(L'[i] \cap \alpha))$. Clearly, MEL computes the sequence $L'[0] \supseteq L'[1] \supseteq L'[2] \ldots$. Since $L[n] \supseteq L'[n] \supseteq L[i]$ for every $n > 0$, we have that $L[i]$ is also the fixpoint of the sequence $L'[0] \supseteq L'[1] \supseteq L'[2] \ldots$, and so MEL computes $L[i]$. Since $inf(S)$ can be computed in time $O(|Q| + |\delta|)$ for any set S, MEL runs in $O(|Q| \cdot (|Q| + |\delta|))$ time.

Interestingly, we have already met Emerson-Lei's algorithm in Chapter **??**. In the proof of Proposition 12.3 we defined a sequence $D_0 \supseteq D_1 \supseteq D_2 \supseteq \ldots$ of infinite acyclic graphs. In the terminology of this chapter, D_{2i+1} was obtained from D_{2i} by removing all nodes having only finitely many descendants, and D_{2i+2} was obtained from D_{2i+1} by removing all nodes having only non-accepting descendants. This corresponds to $D_{2i+1} := \texttt{inf}(D_{2i})$ and $D_{2i+2} := \texttt{pre}^+(D_{2i+1} \cap \alpha)$. So, in fact, we can look at this procedure as the computation of the live states of D_0 using MEL.

13.2.3 Comparing the algorithms

We give two families of examples showing that MEL may outperform Emerson-lei's algorithm, but not always.

A good case for MEL. Consider the automaton of Figure 13.8. The i-th iteration of Emnerson-Lei's algorithm removes q_{n-i+1} The number of calls to *BackwardBFS* is $(n + 1)$, although a simple modification allowing the algorithm to stop if $L = \emptyset$ spares the $(n + 1)$-th operation. On the other hand, the first *inf*-operation of MEL already sets the variable L to the empty set of states, and so, with the same simple modification, the algorithm stops after on iteration.

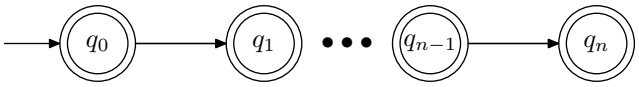


Figure 13.8: An example in which the MEL-algorithm outperforms the Emerson-Lei algorithm

A good case for Emerson-Lei's algorithm. Consider the automaton of Figure 13.9. The i-th iteration, of Emerson-Lei's algorithm removes $q_{(n-i+1),1}$ and $q_{(n-i+1),2}$, and so the algorithm calls *BackwardBFS* $(n + 1)$ times The i-th iteration of MEL-algorithm removes no state as result of the *inf*-operation, and states $q_{(n-i+1),1}$ and $q_{(n-i+1),2}$ as result of the call to *BackwardBFS*. So in this case the *inf*-operations are all redundant.

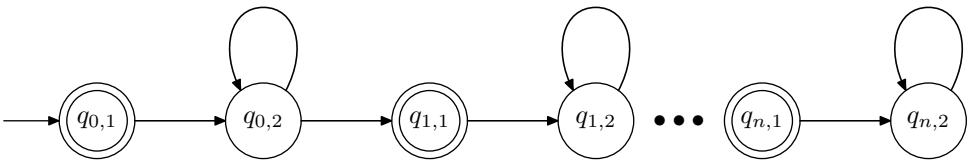
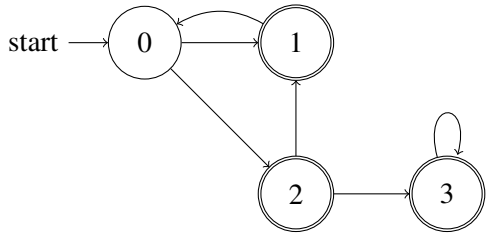


Figure 13.9: An example in which the EL-algorithm outperforms the MEL-algorithm

Exercises

Exercise 91 Which lassos of the following NBA can be found by a run of *NestedDFS*?
(Recall that *NestedDFS* is a nonderterministic algorithm, and so different runs on the same input may return different lassos.)



Exercise 92 A Büchi automaton is *weak* if no strongly connected component contains both accepting and non-accepting states. Show that the following algorithm correctly decides emptiness of weak Büchi automata.
Hint. Consider the root of a scc containing only accepting states.

SingleDFS(A)
Input: weak NBA $A = (Q, \Sigma, \delta, q_0, F)$
Output: EMP if $L_\omega(A) = \emptyset$, NEMP otherwise

```
1  S ← ∅
2  dfs(q_0)
3  report EMP
4  proc dfs(q)
5     add q to S
6     for all r ∈ δ(q) do
7        if r ∉ S then dfs(r)
8        if r ∈ F then report NEMP
9     return
```

Exercise 93 Consider Muller automata whose accepting condition contains one single set of states F, i.e., a run ρ is accepting if $inf(\rho) = F$. Transform *TwoStack* into a linear algorithm for checking emptiness of these automata.
Hint: Consider the version of *TwoStack* for NGAs.

Exercise 94

(1) Given $R, S \subseteq Q$, define $pre^+(R, S)$ as the set of ascendants q of R such that there is a path from q to R that contains only states of S. Give an algorithm to compute $pre^+(R, S)$.

(2) Consider the following modification of Emerson-Lei's algorithm:

MEL2(A)
Input: NBA $A = (Q, \Sigma, \delta, q_0, F)$
Output: EMP if $L_\omega(A) = \emptyset$, NEMP otherwise

```
1  L ← Q
2  repeat
3     OldL ← L
4     L ← pre⁺(L ∩ F, L)
5  until L = OldL
6  if q0 ∈ L then report NEMP
7  else report NEMP
```

Is *MEL2* correct? What is the difference between the sequences of sets computed by *MEL* and *MEL2*?

Chapter 14

Verification and Temporal Logic

Recall that, intuitively, liveness properties are those stating that the system will eventually do something good. More formally, they are properties that are only violated by infinite executions of the systems, i.e., by examining only a finite prefix of an infinite execution it is not possible to determine whether the infinite execution violates the property or not. In this chapter we apply the theory of Büchi automata to the problem of automatically verifying liveness properties.

14.1 Automata-Based Verification of Liveness Properties

In Chapter 8 we introduced some basic concepts about systems: configuration, possible execution, and execution. We extend these notions to the infinite case. An ω-execution of a system is an infinite sequence $c_0c_1c_2\ldots$ of configurations where c_0 is some initial configuration, and for every $i \geq 1$ the configuration c_i is a legal successor according to the semantics of the system of the configuration c_{i-1}. Notice that according to this definition, if a configuration has no legal successors then it does not belong to any ω-execution. Usually this is undesirable, and it is more convenient to assume that such a configuration c has exactly one legal successor, namely c itself. In this way, every reachable configuration of the system belongs to some ω-execution. The terminating executions are then the ω-executions of the form $c_0 \ldots c_{n-1}c_n^\omega$ for some terminating configuration c_n. The set of terminating configurations can usually be identified syntactically. For instance, in a program the terminating configurations are usually those in which control is at some particular program line.

In Chapter 8 we showed how to construct a system NFA recognizing all the executions of a given system. The same construction can be used to define a *system NBA* recognizing all the ω-executions.

Example 14.1 Consider the little program of Chapter 8.

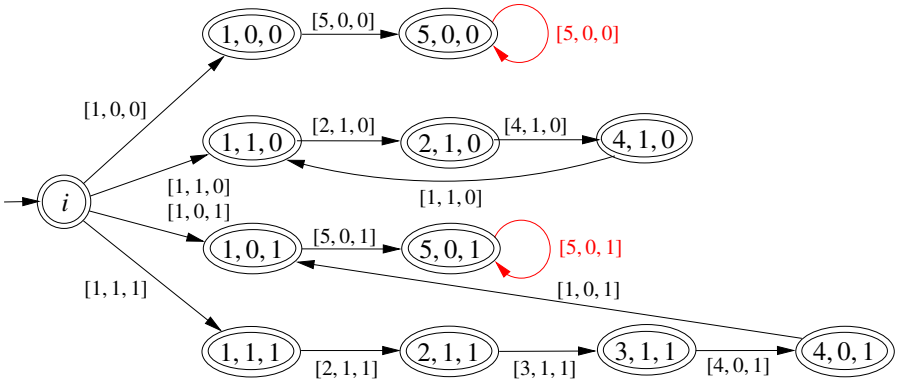


Figure 14.1: System NBA for the program

```
1  while x = 1 do
2    if y = 1 then
3      x ← 0
4    y ← 1 − x
5  end
```

Its system NFA is the automaton of 14.1, but without the red self-loops at states $[5, 0, 0]$ and $[5, 0, 1]$. The system NBA is the result of adding the self-loops □

14.1.1 Checking Liveness Properties

In Chapter 8 we used Lamport's algorithm to present examples of safety properties, and how they can be automatically checked. We do the same now for liveness properties. Figure 14.2 shows again the network of automata modelling the algorithm and its asynchronous product, from which we can easily gain its system NBA. Observe that in this case every configuration has at least a successor, and so no self-loops need to be added.

For $i \in \{0, 1\}$, let NC_i, T_i, C_i be the sets of configurations in which process i is in the non-critical section, is trying to access the critical section, and is in the critical section, respectively, and let Σ stand for the set of all configurations. The *finite waiting* property for process i states that if process i tries to access its critical section, it eventually will. The possible ω-executions that violate the property for process i are represented by the ω-regular expression

$$v_i = \Sigma^* \, T_i \, (\Sigma \setminus C_i)^\omega \, .$$

We can check this property using the same technique as in Chapter 8. We construct the system NBA ωE recognizing the ω-executions of the algorithm (the NBA has just two states), and transform the regular expression v_i into an NBA V_i using the algorithm of Chapter 11. We then

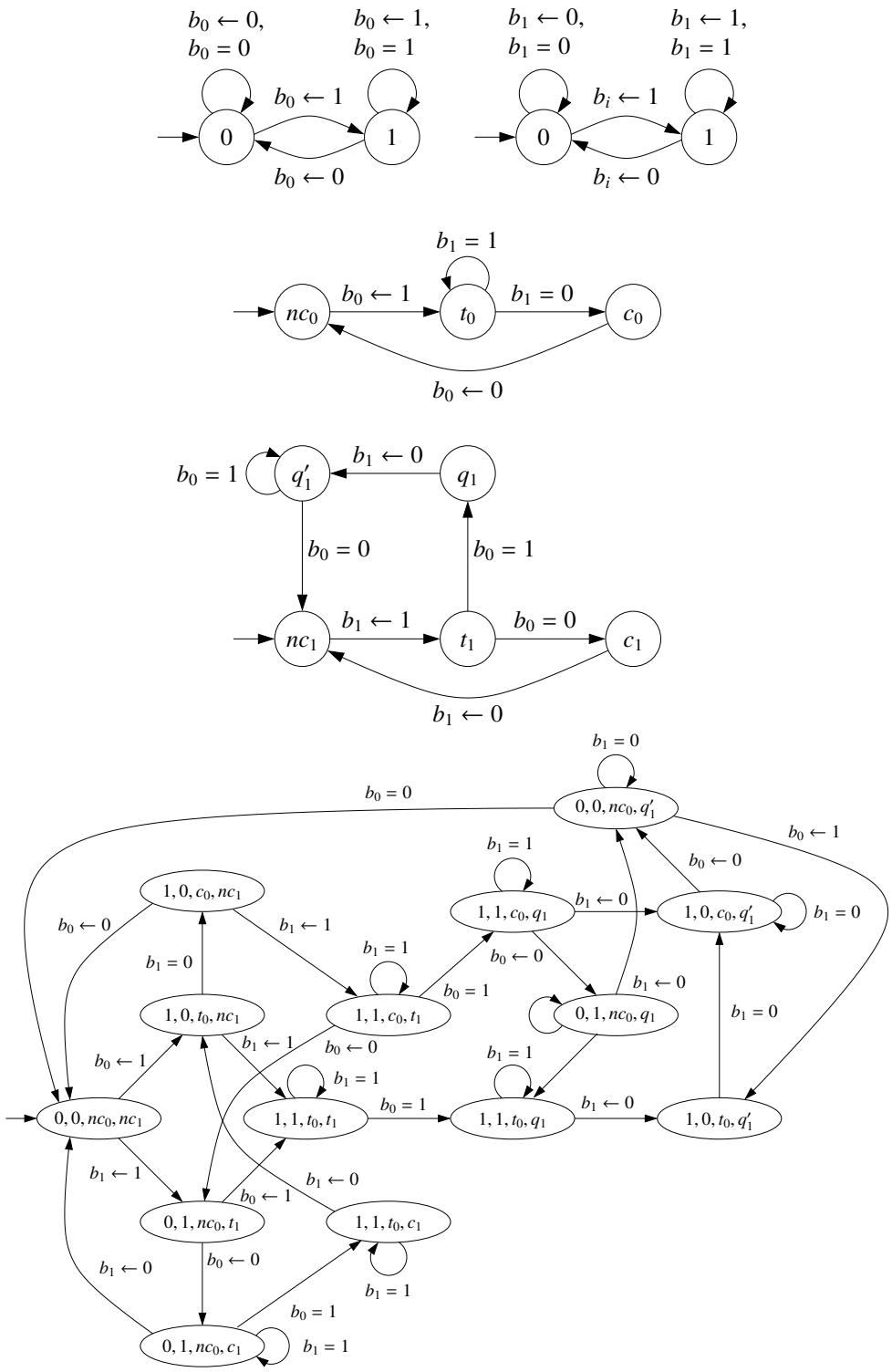


Figure 14.2: Lamport's algorithm and its asynchronous product.

construct an NBA for $\omega E \cap V_i$ using *intersNBA*(), and check its emptiness using one of the algorithms of Chapter 13.

Observe that, since all states of ωE are accepting, we do not need to use the special algorithm for intersection of NBAs, and so we can apply the construction for NFAs.

The result of the check for process 0 yields that the property fails because for instance of the ω-execution

$$[0, 0, nc_0, nc_1]\ \ [1, 0, t_0, nc_1]\ \ [1, 1, t_0, t_1]^\omega$$

In this execution both processes request access to the critical section, but from then on process 1 never makes any further step. Only process 0 continues, but all it does is continuously check that the current value of b_1 is 1. Intuitively, this corresponds to process 1 breaking down after requesting access. But we do not expect the finite waiting property to hold if processes may break down while waiting. So, in fact, our *definition* of the finite waiting property is wrong. We can repair the definition by reformulating the property as follows: in any ω-execution *in which both processes execute infinitely many steps*, if process 0 tries to access its critical section, then it eventually will. The condition that both processes must move infinitely often is called a *fairness assumption*.

The simplest way to solve this problem is to enrich the alphabet of the system NBA. Instead of labeling a transition only with the name of the target configuration, we also label it with the number of the process responsible for the move leading to that configuration. For instance, the transition $[0, 0, nc_0, nc_1] \xrightarrow{[1,0,t_0,nc_1]} [1, 0, t_0, nc_1]$ becomes

$$[0, 0, nc_0, nc_1] \xrightarrow{([1,0,t_0,nc_1],0)} [1, 0, t_0, nc_1]$$

to reflect the fact that $[1, 0, t_0, nc_1]$ is reached by a move of process 0. So the new alphabet of the NBA is $\Sigma \times \{0, 1\}$. If we denote $M_0 = \Sigma \times \{0\}$ and $M_1 = \Sigma \times \{1\}$ for the 'moves" of process 0 and process 1, respectively, then the regular expression

$$inf = (\ (M_0 + M_1)^* M_0 M_1\ )^\omega$$

represents all ω-executions in which both processes move infinitely often, and $L(v_i) \cap L(inf)$ (where v_i is suitably rewritten to account for the larger alphabet) is the set of violations of the reformulated finite waiting property. To check if some ω-execution is a violation, we can construct NBAs for v_i and inf, and compute their intersection. For process 0 the check yields that the properly indeed holds. For process 1 the property still fails because of, for instance, the sequence

$$\begin{array}{ll} (& [0, 0, nc_0, nc_1]\ [0, 1, nc_0, t_1]\ [1, 1, t_0, t_1]\ [1, 1, t_0, q_1] \\ & [1, 0, t_0, q_1']\ [1, 0, c_0, q_1']\ [0, 0, nc_0, q_1']\quad)^\omega \end{array}$$

in which process 1 repeatedly tries to access its critical section, but always lets process 0 access first.

14.2 Linear Temporal Logic

In Chapter 8 and in the previous section we have formalized properties of systems using regular, or ω-regular expressions, NFAs, or NBAs. This becomes rather difficult for all but the easiest properties. For instance, the NBA or the ω-regular expression for the modified finite waiting property are already quite involved, and it is difficult to be convinced that they correspond to the intended property. In this section we introduce a new language for specifying safety and liveness properties, called Linear Temporal Logic (LTL). LTL is close to natural language, but still has a formal semantics.

Formulas of LTL are constructed from a set AP of *atomic propositions*. Intuitively, atomic propositions are abstract names for basic properties of configurations, whose meaning is fixed only after a concrete system is considered. Formally, given a system with a set C of configurations, the meaning of the atomic propositions is fixed by a *valuation function* $\mathcal{V}\colon AP \to 2^C$ that assigns to each abstract name the set of configurations at which it holds.

Atomic propositions are combined by means of the usual Boolean operators and the temporal operators **X** ("next") and **U** ("until"). Formally:

Definition 14.2 *Let AP be a finite set of atomic propositions. The set of LTL formulas over AP, denoted by LTL(AP), is the set of expressions generated by the grammar*

$$\varphi := \mathbf{true} \mid p \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid \mathbf{X}\varphi_1 \mid \varphi_1 \,\mathbf{U}\, \varphi_2 \,.$$

Formulas are interpreted on sequences $\sigma = \sigma_0\sigma_1\sigma_2\ldots$, where $\sigma_i \subseteq AP$ for every $i \geq 0$. We call these sequences *computations*. The set of all computations over AP is denoted by $C(AP)$. The *executable computations* of a system are the computations σ for which there exists an ω-execution $c_0c_1c_2\ldots$ such that for every $i \geq 0$ the set of basic properties satisfied by c_i is exactly σ_i.

Definition 14.3 *Given a computation $\sigma \in C(AP)$, letσ^j denote the suffix $\sigma_j\sigma_{j+1}\,\sigma_{j+2}\ldots$ of σ. The satisfaction relation $\sigma \models \varphi$ (read "σ satisfies φ") is inductively defined as follows:*

- $\sigma \models \mathbf{true}$.
- $\sigma \models p$ *iff* $p \in \sigma(0)$.
- $\sigma \models \neg\varphi$ *iff* $\sigma \not\models \varphi$.
- $\sigma \models \varphi_1 \wedge \varphi_2$ *iff* $\sigma \models \varphi_1$ *and* $\sigma \models \varphi_2$.
- $\sigma \models \mathbf{X}\varphi$ *iff* $\sigma^1 \models \varphi$.
- $\sigma \models \varphi_1 \,\mathbf{U}\, \varphi_2$ *iff there exists* $k \geq 0$ *such that* $\sigma^k \models \varphi_2$ *and* $\sigma^i \models \varphi_1$ *for every* $0 \leq i < k$.

We use the following abbreviations:

- **false**, $\vee$, $\rightarrow$ and $\leftrightarrow$, interpreted in the usual way.

- $\mathbf{F}\varphi = \mathbf{true}\ \mathbf{U}\ \varphi$ ("eventually φ"). According to the semantics above, $\sigma \models \mathbf{F}\varphi$ iff there exists $k \geq 0$ such that $\sigma^k \models \varphi$,.
- $\mathbf{G}\varphi = \neg\mathbf{F}\neg\varphi$ ("always φ" or "globally φ"). According to the semantics above, $\sigma \models \mathbf{G}\varphi$ iff $\sigma^k \models \varphi$ for every $k \geq 0$.

The set of computations that satisfy a formula φ is denoted by $L(\varphi)$. A system *satisfies* φ if *all* its executable computations satisfy φ.

Example 14.4 Consider the little program at the beginning of the chapter. We write some formulas expressing properties of the possible ω-executions of the program. Observe that the system NBA of Figure 14.1has exactly four ω-executions:

$$\begin{array}{lcl} e_1 & = & [1,0,0]\ [5,0,0]^\omega \\ e_2 & = & (\ [1,1,0]\ [2,1,0]\ [4,1,0]\ )^\omega \\ e_3 & = & [1,0,1]\ [5,0,1]^\omega \\ e_4 & = & [1,1,1]\ [2,1,1]\ [3,1,1]\ [4,0,1]\ [1,0,1]\ [5,0,1]^\omega \end{array}$$

Let C be the set of configurations of the program. We choose

$$AP = \{\texttt{at\_1}, \texttt{at\_2}, \ldots, \texttt{at\_5}, \texttt{x=0}, \texttt{x=1}, \texttt{y=0}, \texttt{y=1}\}$$

and define the valuation function $\mathcal{V}\colon AP \to 2^C$ as follows:

- $\mathcal{V}(\texttt{at\_i}) = \{[\ell, x, y] \in C \mid \ell = i\}$ for every $i \in \{1, \ldots, 5\}$.
- $\mathcal{V}(\texttt{x=0}) = \{[\ell, x, y] \in C \mid x = 0\}$, and similarly for $x = 1, y = 0, y = 1$.

Under this valuation, `at_i` expresses that the program is at line i, and `x=j` expresses that the current value of x is j. The executable computations corresponding to the four ω-executions above are

$$\begin{array}{lcl} \sigma_1 & = & \{\texttt{at\_1}, \texttt{x=0}, \texttt{y=0}\}\ \{\texttt{at\_5}, \texttt{x=0}, \texttt{y=0}\}^\omega \\ \sigma_2 & = & (\ \{\texttt{at\_1}, \texttt{x=1}, \texttt{y=0}\}\ \{\texttt{at\_2}, \texttt{x=1}, \texttt{y=0}\}\ \{\texttt{at\_4}, \texttt{x=1}, \texttt{y=0}\}\ )^\omega \\ \sigma_3 & = & \{\texttt{at\_1}, \texttt{x=0}, \texttt{y=1}\}\ \{\texttt{at\_5}, \texttt{x=0}, \texttt{y=1}\}^\omega \\ \sigma_4 & = & \{\texttt{at\_1}, \texttt{x=1}, \texttt{y=1}\}\ \{\texttt{at\_2}, \texttt{x=1}, \texttt{y=1}\}\ \{\texttt{at\_3}, \texttt{x=1}, \texttt{y=1}\}\ \{\texttt{at\_4}, \texttt{x=0}, \texttt{y=1}\} \\ & & \{\texttt{at\_1}, \texttt{x=0}, \texttt{y=1}\}\ \{\texttt{at\_5}, \texttt{x=0}, \texttt{y=1}\}^\omega \end{array}$$

We give some examples of properties:

- $\phi_0 =$ `x=1` $\wedge\ \mathbf{X}$`y=0` $\wedge\ \mathbf{XX}$`at_4`. In natural language: the value of x in the first configuration of the execution is 1, the value of y in the second configuration is 0, and in the third configuration the program is at location 4. We have $\sigma_2 \models \phi_0$, and $\sigma_1, \sigma_3, \sigma_4 \not\models \phi_0$.
- $\phi_1 = \mathbf{F}$`x=0`. In natural language: x eventually gets the value 0. We have $\sigma_1, \sigma_2, \sigma_4 \models \phi_1$, but $\sigma_3 \not\models \phi_1$.

- $\phi_2 = \texttt{x=0}\ \mathbf{U}\ \texttt{at\_5}$. In natural language: x stays equal to 0 until the execution reaches location 5. Notice however that the natural language description is ambiguous: Do executions that never reach location 5 satisfy the property? Do executions that set x to 1 immediately before reaching location 5 satisfy the property? The formal definition removes the ambiguities: the answer to the first question is 'no', to the second 'yes'. We have $\sigma_1, \sigma_3 \models \phi_2$ and $\sigma_2, \sigma_4 \not\models \phi_2$.

- $\phi_3 = \texttt{y=1} \wedge \mathbf{F}(\texttt{y=0} \wedge \texttt{at\_5}) \wedge \neg(\mathbf{F}(\texttt{y=0} \wedge \mathbf{X}\texttt{y=1}))$. In natural language: the first configuration satisfies $y = 1$, the execution terminates in a configuration with $y = 0$, and y never decreases during the execution. This is one of the properties we analyzed in Chapter 8, and it is not satisfied by any ω-execution.

□

Example 14.5 We express several properties of the Lamport-Bruns algorithm (see Chapter 8) using LTL formulas. As system NBA we use the one in which transitions are labeled with the name of the target configuration, and with the number of the process responsible for the move leading to that configuration. We take $AP = \{NC_0, T_0, C_0, NC_1, T_1, C_1, M_0, M_1\}$, with the obvious valuation.

- The mutual exclusion property is expressed by the formula

  $$\mathbf{G}(\neg C_0 \vee \neg C_1)$$

  The algorithm satisfies the formula.

- The property that process i cannot access the critical section without having requested it first is expressed by

  $$\neg(\neg T_i\ \mathbf{U}\ C_i)$$

  Both processes satisfy this property.

- The naïve finite waiting property for process i is expressed by

  $$\mathbf{G}(T_i \rightarrow \mathbf{F}C_i)$$

  The modified version in which both processes must execute infinitely many moves is expressed

  $$(\mathbf{GF}M_0 \wedge \mathbf{GF}M_1) \rightarrow \mathbf{G}(T_i \rightarrow \mathbf{F}C_i)$$

  Observe how fairness assumptions can be very elegantly expressed in LTL. The assumption itself is expressed as a formula ψ, and the property that ω-executions satisfying the fairness assumption also satisfy ϕ is expressed by $\psi \rightarrow \phi$.

  None of the processes satisfies the naïve version of the finite waiting property. Process 0 satisfies the modified version, but process 1 does not.

- The bounded overtaking property for process 0 is expressed by

$$\mathbf{G}(\ T_0 \rightarrow (\neg C_1\ \mathbf{U}\ (C_1\ \mathbf{U}\ (\neg C_1\ \mathbf{U}\ C_0))))$$

  The formula states that whenever T_0 holds, the computation continues with a (possibly empty!) interval at which we see $\neg C_1$ holds, followed by a (possibly empty!) interval at which C_1 holds, followed by a point at which C_0 holds. The property holds.

□

14.3 From LTL formulas to generalized Büchi automata

We present an algorithm that, given a formula $\varphi \in LTL(AP)$ returns a NGA A_φ over the alphabet 2^{AP} recognizing $L(\varphi)$, and then derive a fully automatic procedure that, given a system and an LTL formula, decides whether the executable computations of the system satisfy the formula.

14.3.1 Satisfaction sequences and Hintikka sequences

We introduce the notion of satisfaction sequence and Hintikka sequence for a computation σ and a formula ϕ.

Definition 14.6 *Given a formula ϕ, the* negation *of ϕ is the formula ψ if $\phi = \neg\psi$, and the formula $\neg\phi$ otherwise. The* closure *$cl(\phi)$ of a formula ϕ is the set containing all subformulas of ϕ and their negations. A nonempty set $\alpha \subseteq cl(\phi)$ is an atom of $cl(\phi)$ if it satisfies the following properties:*

(a0) If **true** $\in cl(\phi)$*, then* **true** $\in \alpha$*.*

(a1) For every $\phi_1 \wedge \phi_2 \in cl(\phi)$: $\phi_1 \wedge \phi_2 \in \alpha$ if and only if $\phi_1 \in \alpha$ and $\phi_2 \in \alpha$.

(a2) For every $\neg\phi_1 \in cl(\phi)$: $\neg\phi_1 \in \alpha$ if and only if $\phi_1 \notin \alpha$.

The set containing all atoms of $cl(\phi)$ is denoted by $at(\phi)$.

Observe that if α is the set of all formulas of $cl(\phi)$ satisfied by a computation σ, then α is necessarily an atom. For instance, condition (a1) holds because σ satisfies a conjunction if and only if it satisfies its conjuncts.

Example 14.7 The closure of the formula $p \wedge (p\ \mathbf{U}\ q)$ is

$$\{\ p,\ \neg p,\ q, \neg q,\ p\ \mathbf{U}\ q\ , \neg p\ \mathbf{U}\ q,\ p \wedge (p\ \mathbf{U}\ q), \neg(p \wedge (p\ \mathbf{U}\ q))\}\ .$$

The only two atoms containing $p \wedge (p\ \mathbf{U}\ q)$ are

$$\{p,\ q, p\ p\ \mathbf{U}\ q,\ p \wedge (p\ \mathbf{U}\ q)\} \quad \text{and} \quad \{p,\ \neg q,\ p\ \mathbf{U}\ q,\ p \wedge (p\ \mathbf{U}\ q)\}\ .$$

Let us see why. By (a1), every atom α containing $p \wedge (p \mathbf{U} q)$ must contain p and $p \mathbf{U} q$; by (a2), an atom always contains either a subformula or its negation, but not both, and so α contains neither $\neg p$ nor $\neg(p \mathbf{U} q)$, and exactly one of q and $\neg q$. □

Definition 14.8 *The* satisfaction sequence *for a computation σ and a formula ϕ is the infinite sequence of atoms*

$$sats(\sigma, \phi) = sats(\sigma, \phi, 0)\ sats(\sigma, \phi, 1)\ sats(\sigma, \phi, 2)\ \ldots$$

where $sats(\sigma, \phi, i)$ is the atom containing the formulas of $cl(\phi)$ satisfied by σ^i.

Intuitively, the satisfaction sequence of a computation σ is obtained by "completing" σ: while σ only indicates which atomic propositions hold at each point in time, the satisfaction sequence also indicates which atom holds.

Example 14.9 Let $\phi = p \mathbf{U} q$, $\sigma_1 = \{p\}^\omega$, and $\sigma_2 = (\{p\}\{q\})^\omega$. We have

$$\begin{array}{lcl} sats(\sigma_1, \phi) & = & \{p, \neg q, \neg(p \mathbf{U} q)\}^\omega \\ sats(\sigma_2, \phi) & = & (\{p, \neg q, p \mathbf{U} q\}\{\neg p, q, p \mathbf{U} q\})^\omega \end{array}$$

□

Observe that σ satisfies ϕ if and only if and only if $\phi \in sats(\sigma, \phi, 0)$, i.e., if and only if ϕ belongs to the first atom of σ.

Satisfaction sequences have a *semantic* definition: in order to know which atom holds at a point one must know the semantics of LTL. Hintikka sequences provide a *syntactic* characterization of satisfaction sequences. The definition of a Hintikka sequence does not involve the semantics of LTL, i.e., someone who ignores the semantics can still decide whether a sequence is a Hintikka sequence or not. We prove that a sequence is a satisfaction sequence if and only if it is a Hintikka sequence.

Definition 14.10 *A* pre-Hintikka sequence *for ϕ is an infinite sequence $\alpha = \alpha_0\alpha_1\alpha_2\ldots$ of atoms satisfying the following conditions for every $i \geq 0$:*

(l1) For every $\mathbf{X}\phi \in cl(\phi)$: $\mathbf{X}\phi \in \alpha_i$ if and only if $\phi \in \alpha_{i+1}$.

(l2) For every $\phi_1 \mathbf{U} \phi_2 \in cl(\phi)$: $\phi_1 \mathbf{U} \phi_2 \in \alpha_i$ if and only if $\phi_2 \in \alpha_i$ or $\phi_1 \in \alpha$ and $\phi_1 \mathbf{U} \phi_2 \in \alpha_{i+1}$.

A pre-Hintikka sequence is a Hintikka sequence *if it also satisfies*

(g) For every $\phi_1 \mathbf{U} \phi_2 \in \alpha_i$, there exists $j \geq i$ such that $\phi_2 \in \alpha_j$.

A pre-Hintikka or Hintikka sequence α matches *a computation σ if for every atomic proposition $p \in AP$, $p \in \alpha_i$ if and only if $p \in \sigma_i$.*[1]

[1] Loosely speaking, $\alpha_0\alpha_1\alpha_2\ldots$ matches σ if it extends σ with new formulas.

Observe that conditions (l1) and (l2) are *local*: in order to know if α satisfies them we only need to inspect every pair α_i, α_{i+1} of consecutive atoms. On the contrary, condition (g) is *global*, since the the distance between the indices i and j can be arbitrarily large.

It follows immediately from the definition above that if $\alpha = \alpha_0\alpha_1\alpha_2\ldots$ is a satisfaction sequence, then every pair α_i, α_{i+1} satisfies (l1) and (l2), and the sequence α itself satisfies (g). So every satisfaction sequence is a Hintikka sequence. The following theorem shows that the converse also holds: Every Hintikka sequence if a satisfaction sequence.

Theorem 14.11 *Let σ be a computation and let ϕ be a formula. The unique Hintikka sequence for ϕ matching σ is the satisfaction sequence $sats(\sigma, \phi)$.*

Proof: It follows immediately from the definitions that $sats(\sigma, \phi)$ is a Hintikka sequence for ϕ matching σ. For the other direction, let $\alpha = \alpha_0\alpha_1\alpha_2\ldots$ be a Hintikka sequence for ϕ matching σ, and let ψ be an arbitrary formula of $cl(\phi)$. We prove that for every $i \geq 0$: $\psi \in \alpha_i$ if and only if $\psi \in sats(\sigma, \phi, i)$. The proof is by induction on the structure of ψ.

- $\psi = \mathbf{true}$. Then $\mathbf{true} \in sats(\sigma, \phi, i)$ and, since α_i is an atom, $\mathbf{true} \in \alpha_i$.
- $\psi = p$ for an atomic proposition p. Since α matches σ, we have $p \in \alpha_i$ if and only if $p \in \sigma_i$. By the definition of satisfaction sequence, $p \in \sigma_i$ if and only if $p \in sats(\sigma, \phi, i)$.
- $\psi = \phi_1 \wedge \phi_2$. We have

$$\begin{array}{lll} & \phi_1 \wedge \phi_2 \in \alpha_i & \\ \Leftrightarrow & \phi_1 \in \alpha_i \text{ and } \phi_2 \in \alpha_i & \text{(condition (a1))} \\ \Leftrightarrow & \phi_1 \in sats(\sigma, \phi, i) \text{ and } \phi_2 \in sats(\sigma, \phi, i) & \text{(induction hypothesis)} \\ \Leftrightarrow & \phi_1 \wedge \phi_2 \in sats(\sigma, \phi, i) & \text{(definition of } sats(\sigma, \phi)) \end{array}$$

- $\psi \neq \phi_1$ or $\psi = \mathbf{X}\phi_1$. The proofs are very similar to the last one.
- $\psi = \phi_1 \mathbf{U} \phi_2$. The proof is divided into two parts.
  (a) If $\phi_1 \mathbf{U} \phi_2 \in \alpha_i$, then $\phi_1 \mathbf{U} \phi_2 \in sats(\sigma, \phi, i)$. By condition (l2) of the definition of Hintikka sequence, we have to consider two cases:
  - $\phi_2 \in \alpha_i$. By induction hypothesis, $\phi_2 \in sats(\sigma, \phi)$, and so $\phi_1 \mathbf{U} \phi_2 \in sats(\sigma, \phi, i)$.
  - $\phi_1 \in \alpha_i$ and $\phi_1 \mathbf{U} \phi_2 \in \alpha_{i+1}$. By condition (g), there is at least one index $j \geq i$ such that $\phi_2 \in \alpha_j$. Let j_m be the smallest of these indices. We prove the result by induction on $j_m - i$. If $i = j_m$, then $\phi_2 \in \alpha_j$, and we proceed as in the case $\phi_2 \in \alpha_i$. If $i < j_m$, then since $\phi_1 \in \alpha_i$, we have $\phi_1 \in sats(\sigma, \phi, i)$ (induction on ψ). Since $\phi_1 \mathbf{U} \phi_2 \in \alpha_{i+1}$, we have either $\phi_2 \in \alpha_{i+1}$ or $\phi_1 \in \alpha_{i+1}$. In the first case we have $\phi_2 \in sats(\sigma, \phi, i+1)$, and so $\phi_1 \mathbf{U} \phi_2 \in sats(\sigma, \phi, i)$. In the second case, by induction hypothesis (induction on $j_m - i$), we have $\phi_1 \mathbf{U} \phi_2 \in sats(\sigma, \phi, i+1)$, and so $\phi_1 \mathbf{U} \phi_2 \in sats(\sigma, \phi, i)$.

  (b) If $\phi_1 \mathbf{U} \phi_2 \in sats(\sigma, \phi, i)$, then $\phi_1 \mathbf{U} \phi_2 \in \alpha_i$. We consider again two cases.

– $\phi_2 \in sats(\sigma, \phi, i)$. By induction hypothesis, $\phi_2 \in \alpha_i$, and so $\phi_1 \mathbf{U} \phi_2 \in \alpha_i$.
– $\phi_1 \in sats(\sigma, \phi, i)$ and $\phi_1 \mathbf{U} \phi_2 \in sats(\sigma, \phi, i+1)$. By the definition of a satisfaction sequence, there is at least one index $j \geq i$ such that $\phi_2 \in sats(\sigma, \phi, j)$. Proceed now as in case (a).

□

14.3.2 Constructing the NGA

Given a formula ϕ, we construct a *generalized* Büchi automaton A_ϕ recognizing $L(\phi)$. By the definition of a satisfaction sequence, a computation σ satisfies ϕ if and only if $\phi \in sats(\sigma, \phi, 0)$ and, by Theorem 14.11, $sats(\sigma, \phi)$ is the (unique) Hintikka sequence for ϕ matching σ. So A_ϕ must recognize the computations σ such that the first atom of the Hintikka sequence for ϕ matching σ contains ϕ.

To achieve this goal, we construct A_ϕ so that its runs are all the sequences

$$q_0 \xrightarrow{\sigma_0} \alpha_0 \xrightarrow{\sigma_1} \alpha_1 \xrightarrow{\sigma_2} \dots$$

such that $\sigma = \sigma_0\sigma_1 \dots$ is a computation, and $\alpha = \alpha_0\alpha_1 \dots$ is a pre-Hintikka sequence of ϕ matching σ. Moreover, the sets of accepting states (recall that A_ϕ is a NGA) are chosen so that a run is accepting if and only if the pre-Hintikka sequence is also a Hintikka sequence.

The condition that runs must be pre-Hintikka sequences determines the alphabet, states, transitions, and initial state of A_ϕ:

- The alphabet of A_ϕ is 2^{AP}.
- The states of A_ϕ (apart from the distinguished initial state q_0) are atoms of ϕ.
- The output transitions of the initial state q_0 are the triples $q_0 \xrightarrow{\sigma_0} \alpha$ such that σ_0 matches α (i.e., for every atomic proposition $p \in AP$, $p \in \sigma_0$ if and only if $p \in \alpha_0$), and $\phi \in \alpha$.
- The output transitions of any other state α (where α is an atom) are the triples $\alpha \xrightarrow{\sigma} \beta$ such that σ matches β, and the pair α, β satisfies conditions (l1) and (l2) (where α and β play the roles of α_i resp. α_{i+1}).

The sets of accepting states of A_ϕ are determined by the condition that accepting runs must exactly correspond to the Hintikka sequences. By the definition of a Hintikka sequence, we must guarantee that in every run $q_0 \xrightarrow{\sigma_0} \alpha_0 \xrightarrow{\sigma_1} \alpha_1 \xrightarrow{\sigma_2} \dots$, for every $i \geq 0$ such that α_i contains a subformula $\phi_1 \mathbf{U} \phi_2$, there exists $j \geq i$ such that $\phi_2 \in \alpha_j$. By condition (l2), this amounts to guaranteeing that every run contains infinitely many indices i such that $\phi_2 \in \alpha_i$, or infinitely many indices j such that $\neg(\phi_1 \mathbf{U} \phi_2) \in \alpha_j$. So we choose the sets of accepting states as follows:

- The accepting condition contains a set $F_{\phi_1 \mathbf{U} \phi_2}$ of accepting states for each subformula $\phi_1 \mathbf{U} \phi_2$ of ϕ. An atom belongs to $F_{\phi_1 \mathbf{U} \phi_2}$ if it does not contain $\phi_1 \mathbf{U} \phi_2$ or if it contains ϕ_2.

The pseudocode for the translation algorithm is shown below.

LTLtoNGA(ϕ)
Input: LTL-Formula ϕ over AP
Output: NGA $A_\phi = (Q, 2^{AP}, q_0, \delta, \mathcal{F})$ with $L(A_\phi) = L(\phi)$

```
1  Q = {q_0}; δ ← ∅;
2  for all α ∈ at(φ) such that φ ∈ α do
3      add (q_0, α ∩ AP, α) to δ; add α to W
4  while W ≠ ∅ do
5    pick α from W
6    add α to Q
7    for all φ_1 U φ_2 ∈ cl(φ) do
8      if φ_1 U φ_2 ∉ α or φ_2 ∈ α then add α to F_{φ_1 U φ_2}
9    for all P ⊆ AP do
10     for all β ∈ at(φ) matching P do
11       if α, β satisfies (l1) and (l2) then
12         add (α, P, β) to δ
13         if β ∉ Q then add β to W
14 F ← ∅
15 for all φ_1 U φ_2 ∈ cl(φ) do F ← F ∪ {F_{φ_1 U φ_2}}
16 return (Q, 2^{AP}, q_0, δ, F)
```

Example 14.12 We construct the automaton A_ϕ for the formula $\phi = p \,\mathbf{U}\, q$. The closure $cl(\phi)$ has eight atoms, corresponding to all the possible ways of choosing between p and $\neg p$, q and $\neg q$, $p \,\mathbf{U}\, q$ and $\neg(p \,\mathbf{U}\, q)$. However, we can easily see that the atoms $\{p, q, \neg(p \,\mathbf{U}\, q)\}$, $\{\neg p, q, \neg(p \,\mathbf{U}\, q)\}$, and $\{\neg p, \neg q, p \,\mathbf{U}\, q\}$ have no output transitions, because they would violate condition (l2). So these states can be eliminated, and we are left with the five atoms shown in Figure 14.3, plus the initial state q_0. Figure 14.3 uses some conventions to simplify the graphical representation. Observe that every transition of A_ϕ leading to an atom α is labeled by $\alpha \cap AP$. Therefore, since the label of a transition can be deduced from its target state, they are omitted in the figure. Moreover, since ϕ only has one subformula of the form $\phi_1 \, U \, \phi_2$, the NGA is in fact a NBA, and we can represent the accepting states as for NBAs.

Let $\alpha = \{\neg p, \neg q, \neg(p \,\mathbf{U}\, q)\}$ and $\beta = \{p, \neg q, p \,\mathbf{U}\, q\}$. A_ϕ contains a transition $\alpha \xrightarrow{\{p\}} \beta$ because $\{p\}$ matches β, and α, β satisfy conditions (l1) and (l2). Condition (l1) holds vacuously, because ϕ contains no subformulas of the form $X\psi$; condition (l2) holds because $p \,\mathbf{U}\, q \notin \alpha$ and $q \notin \beta$ and $p \notin \alpha$. There is no transition from β to α because it would violate condition (l2): $p \,\mathbf{U}\, q \in \beta$, but neither $q \in \beta$ nor $p \,\mathbf{U}\, q \in \alpha$. □

It is worth noticing that the NGAs obtained from conversions of LTL formulas satisfy the following property: every word accepted by the NGA has one single accepting run. This follows

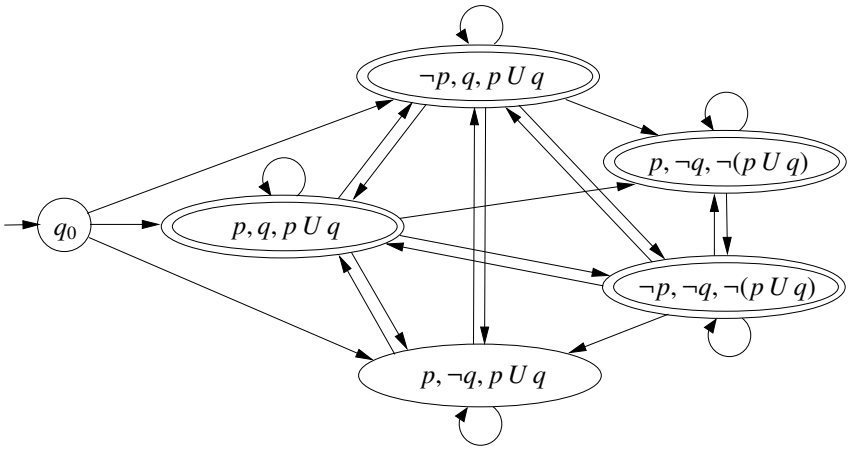


Figure 14.3: NGA (NBA) for the formula $p\ \mathbf{U}\ q$.

from the fact that, loosely speaking, an accepting run is the satisfaction sequence of the computation it accepts, and the satisfaction sequence of a given computation is by definition unique.

14.3.3 Reducing the NGA

The reduction algorithm for NFAs shown in Chapter 3 can be adapted to NBAs and to NGAs. Recall that given an NFA $A = (Q, \Sigma, \delta, q_0, F)$ the algorithm computes the relation *CSR*, the coarsest stable refinement of the equivalence relation $\{F, Q \setminus F\}$, and returns the quotient A/CSR. The relation *CSR* partitions the set of states into blocks, and the states of a block are either all final or all nonfinal (because every equivalence class of *CSR* is included in F or $Q \setminus F$). Moreover, since *CSR* s stable, for every two states q, r of a block of *CSR* and for every $(q, a, q') \in \delta$, there is a transition (r, a, r') such that q', r' belong to the same block. This implies $L(q) = L(r)$, because every run $q \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \cdots q_{n-1} \xrightarrow{a_n} q_n$ can be "matched" by a run $r \xrightarrow{a_1} r_1 \xrightarrow{a_2} r_2 \cdots r_{n-1} \xrightarrow{a_n} r_n$ in such a way that for every $i \geq 1$ the states q_i, r_i belong to the same block, and so, in particular, q_n is final if and only if r_n is final, which implies $a_1 \dots a_n \in L(q)$ if and only if $a_1 \dots a_n \in L(r)$.

Observe that we not only have that q_n and r_n are both final or nonfinal: the same holds for every pair q_i, r_i. Moreover, the property also for ω-words: every infinite run $q \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} q_3 \cdots$ is "matched" by a run $r \xrightarrow{a_1} r_1 \xrightarrow{a_2} r_2 \xrightarrow{a_3} r_3 \cdots$ so that for every $i \geq 1$ the states q_i, r_i are both accepting or non-accepting. This immediately proves $L_\omega(A) = L_\omega(A/CSR)$, and so the reduction algorithm also works for NBAs.

For the extension to NGA, let $A = (Q, \Sigma, \delta, q_0, \{F_1, \dots, F_n\})$ ba a NGA. Consider the partition of Q into blocks given by: two states q, r belong to the same block if for every $i \in \{1, \dots, n\}$ either $q, r \in F_i$ or $q, r \notin F_i$. Define CSR' as the coarsest stable refinement of this new partition. For every two states q, r belonging to the same block of CSR', we now have that every infinite run $q \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} q_3 \cdots$ is "matched" by a run $r \xrightarrow{a_1} r_1 \xrightarrow{a_2} r_2 \xrightarrow{a_3} r_3 \cdots$ so that for every $i \geq 1$

and for every $j \in \{1, \ldots, n\}$ either $q_i, r_i \in F_j$ or $q_i, r_i \notin F_j$. So we get $L_\omega(A) = L_\omega(A/CSR')$.

The algorithm can be applied to the NBA of Figure 14.3. The relation CSR' has three equivalence classes:

$$\begin{array}{lcl} Q_0 & = & \{\ q_0, \{p, \neg q, p\ \mathbf{U}\ q\}\ \} \\ Q_1 & = & \{\ \{p, q, p\ \mathbf{U}\ q\},\ \{\neg p, q, p\ \mathbf{U}\ q\},\ \{\neg p, \neg q, \neg(p\ \mathbf{U}\ q)\}\ \} \\ Q_2 & = & \{\ \{p, \neg q, \neg(p\ \mathbf{U}\ q)\}\ \} \end{array}$$

which leads to the reduced NBA of Figure 14.4.

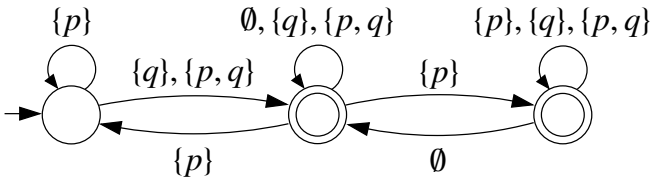


Figure 14.4: Reduced NBA for the formula $p\ \mathbf{U}\ q$.

14.3.4 Size of the NGA

Let n be the length of the formula ϕ. It is easy to see that the set $cl(\phi)$ has size $O(n)$. Therefore, the NGA A_ϕ has at most $O(2^n)$ states. Since ϕ contains at most n subformulas of the form $\phi_1\ \mathbf{U}\ \phi_2$, A_ϕ has at most n sets of final states.

We now prove a matching lower bound on the number of states. We exhibit a family of formulas $\{\phi_n\}_{n\geq 1}$ such that ϕ_n has length $O(n)$, and every NGA recognizing $L_\omega(\phi_n)$ has at least 2^n states.

Consider the family of ω-languages $\{D_n\}_{n\geq 1}$ over the alphabet $\{0, 1, \#\}$ given by $D_n = \{ww\#^\omega \mid w \in \{0,1\}^n\}$. We first show that every NGA recognizing D_n has at least 2^n states. Assume $A = (Q, \{0, 1, \#\}, \delta, q_0, \{F_1, \ldots, F_k\})$ recognizes D_n and $|Q| < 2^n$. Then for every word $w \in \{0,1\}^n$ there is a state q_w such that A accepts $w\#^\omega$ from q_w. By the pigeon hole principle, $q_{w_1} = q_{w_2}$ for two distinct words $w_1, w_2 \in \{0,1\}^n$. But then A accepts $w_1 w_2 \#^\omega$, which does not belong to D_n, contradicting the hypothesis.

It now suffices to construct a family of formulas $\{\phi_n\}_{n\geq 1}$ of size $O(n)$ such that $L_\omega(\phi_n) = D_n$. For this, we take $AP = \{0, 1, \#\}$ with the obvious valuation, and construct the following formulas:

- $\phi_{n1} = \mathbf{G}((0 \vee 1 \vee \#) \wedge \neg(0 \wedge 1) \wedge \neg(0 \wedge \#) \wedge \neg(1 \wedge \#))$.
  This formula expresses that at every position exactly one proposition of AP holds.

- $\phi_{n2} = \neg\# \wedge \left(\bigwedge_{i=1}^{2n-1} X^i \neg\# \right) \wedge X^{2n} G\, \#$.
  This formula expresses that # does not hold at any of the first $2n$ positions, and it holds at at all later positions.

- $\phi_{n3} = \mathbf{G}(\ (0 \rightarrow X^n(0 \vee \#))\ \wedge\ (1 \rightarrow X^n(1 \vee \#))\ )$.
  This formula expresses that if the atomic proposition holding at a position is 0 or 1, then n positions later the atomic proposition holding is the same one, or #.

Clearly, $\phi_n = \phi_{n1} \wedge \phi_{n2} \wedge \phi_{n3}$ is the formula we are looking for. Observe that ϕ_n contains $O(n)$ characters.

14.4 Automatic Verification of LTL Formulas

We can now sketch the procedure for the automatic verification of properties expressed by LTL formulas. The input to the procedure is

- a system NBA A_s obtained either directly from the system, or by computing the asynchronous product of a network of automata;
- a formula ϕ of LTL over a set of atomic propositions AP; and
- a valuation ν: $AP \rightarrow 2^C$, where C is the set of configurations of A_s, describing for each atomic proposition the set of configurations at which the proposition holds.

The procedure follows these steps:

(1) Compute a NGA A_v for the *negation* of the formula ϕ. A_v recognizes all the computations that *violate* ϕ.

(2) Compute a NGA $A_v \cap A_s$ recognizing the executable computations of the system that violate the formula.

(3) Check emptiness of $A_v \cap A_s$.

Steps (1) can be carried out by applying *LTLtoNGA*, and Step (3) by, say, the two-stack algorithm. For Step (2), observe first that the alphabets of A_v and A_s are different: the alphabet of A_v is 2^{AP}, while the alphabet of A_s is the set C of configurations. By applying the valuation ν we transform A_v into an automaton with C as alphabet. Since all the states of system NBAs are accepting, the automaton $A_v \cap A_s$ can be computed by *interNFA*.

It is important to observe that the three steps can be carried out simultaneously. The states of $A_v \cap A_s$ are pairs $[\alpha, c]$, where α is an atom of ϕ, and c is a configuration. The following routine takes a pair $[\alpha, c]$ as input and returns its successors in the NGA $A_v \cap A_s$. The routine first computes the successors of c in A_s. Then for each successor c', it computes first the set P of atomic propositions satisfying c' according to the valuation, and then the set of atoms β such that (a) β matches P and (b) the pair α, β satisfies conditions (l1) and (l2). The successors of $[\alpha, c]$ are the pairs $[\beta, c']$.

Succ($[\alpha, c]$)

```
1  S ← ∅
2    for all c' ∈ δ_s(c) do
3    P ← ∅
4    for all p ∈ AP do
5      if c' ∈ ν(p) then add p to P
6    for all β ∈ at(φ) matching P do
7      if  α, β satisfies (l1) and (l2) then add c' to S
8  return S
```

This routine can be inserted in the algorithm for the emptiness check. For instance, if we use *TwoStack*, then we just replace line 6

6 **for all** $r \in \delta(q)$ **do**

by a call to the routine:

6 **for all** $[\beta, c'] \in Succ([\alpha, c])$ **do**

Exercises

Exercise 95 Let $AP = \{p, q\}$ and let $\Sigma = 2^{AP}$. Give LTL formulas defining the following languages:

$\{p, q\}\ \emptyset\ \Sigma^\omega$ $\qquad$ $\Sigma^*\ \{q\}^\omega$

$\Sigma^*\ \{p\}\ \Sigma^*\ \{q\}\ \Sigma^\omega$ $\qquad$ $\{p\}^*\ \{q\}^*\ \emptyset^*\ \Sigma^\omega$

Exercise 96 Let $AP = \{p, q\}$. Give NBAs accepting the languages defined by the LTL formulas $\mathbf{XG}\neg p$, $(\mathbf{GF}p) \Rightarrow (\mathbf{F}q)$, and $p \wedge \neg\mathbf{XF}p$.

Exercise 97 Two formulas ϕ, ψ of LTL are equivalent, denoted by $\phi \equiv \psi$, if they are satisfied by the same computations. Which of the following equivalences hold?

$$\begin{array}{rcl} \mathbf{X}\varphi \wedge \mathbf{X}\psi & \equiv & \mathbf{X}\varphi \wedge \psi \\ \mathbf{F}\varphi \wedge \mathbf{F}\psi & \equiv & \mathbf{F}\varphi \wedge \psi \\ \mathbf{G}\varphi \wedge \mathbf{G}\psi & \equiv & \mathbf{G}\varphi \wedge \psi \\ \mathbf{FX}\varphi & \equiv & \mathbf{XF}\varphi \\ (\psi_1\ \mathbf{U}\ \varphi) \wedge (\psi_2\ \mathbf{U}\ \varphi) & \equiv & (\psi_1 \wedge \psi_2)\ \mathbf{U}\ \varphi \\ (\varphi\ \mathbf{U}\ \psi_1) \wedge (\varphi\ \mathbf{U}\ \psi_2) & \equiv & \varphi\ \mathbf{U}\ (\psi_1 \wedge \psi_2) \\ \psi\ \mathbf{U}\ \mathbf{F}\varphi & \equiv & \mathbf{F}\varphi \\ \psi\ \mathbf{U}\ \mathbf{GF}\varphi & \equiv & \mathbf{GF}\varphi \end{array} \qquad \begin{array}{rcl} \mathbf{X}\varphi \vee \mathbf{X}\psi & \equiv & \mathbf{X}\varphi \vee \psi \\ \mathbf{F}\varphi \vee \mathbf{F}\psi & \equiv & \mathbf{F}\varphi \vee \psi \\ \mathbf{G}\varphi \vee \mathbf{G}\psi & \equiv & \mathbf{G}\varphi \vee \psi \\ \mathbf{GX}\varphi & \equiv & \mathbf{XG}\varphi \\ (\psi_1\ \mathbf{U}\ \varphi) \vee (\psi_2\ \mathbf{U}\ \varphi) & \equiv & (\psi_1 \vee \psi_2)\ \mathbf{U}\ \varphi \\ (\varphi\ \mathbf{U}\ \psi_1) \vee (\varphi\ \mathbf{U}\ \psi_2) & \equiv & \varphi\ \mathbf{U}\ (\psi_1 \vee \psi_2) \\ \psi\ \mathbf{U}\ \mathbf{G}\varphi & \equiv & \mathbf{G}\varphi \\ \psi\ \mathbf{U}\ \mathbf{FG}\varphi & \equiv & \mathbf{GF}\varphi \end{array}$$

Exercise 98 Prove $\mathbf{FG}p \equiv \mathbf{VFG}p$ and $\mathbf{GF}p \equiv \mathbf{VGF}p$ for *every* sequence $\mathbf{V} \in \{\mathbf{F}, \mathbf{G}\}^*$ of the temporal operators **F** and **G** .
Hint: Observe that, since $\mathbf{FF}p \equiv \mathbf{F}p$ and $\mathbf{GG}p \equiv \mathbf{G}p$, it suffices to prove $\mathbf{FG}p \equiv \mathbf{GFG}p$ and $\mathbf{GF}p \equiv \mathbf{FGF}p$.

Exercise 99 (Santos Laboratory). The *weak until* operator **W** has the following semantics:

- $\sigma \models \phi_1 \ \mathbf{W} \ \phi_2$ iff there exists $k \geq 0$ such that $\sigma^k \models \phi_2$ and $\sigma^i \models \phi_1$ for all $0 \leq i < k$, or $\sigma^k \models \phi_2$ for every $k \geq 0$.

Prove: $p \ \mathbf{W} \ q \equiv \mathbf{G}p \vee (p \ \mathbf{U} \ q) \equiv \mathbf{F}\neg p \rightarrow (p \ \mathbf{U} \ q) \equiv p \ \mathbf{U} \ (q \vee \mathbf{G}p)$.

Exercise 100 (Santos Laboratory). Let $AP = \{p, q, r\}$. Give formulas that hold for the computations satisfying the following properties. If in doubt about what the property really means, choose an interpretation, and explicitely indicate your choice. Here are two solved examples:

- p is false before q: $\mathbf{F}(q) \rightarrow (\neg p \ \mathbf{U} \ q)$.
- p becomes true before q: $\neg q \ \mathbf{W} \ (p \wedge \neg q)$.

Now it is your turn:

- p is false between q and r.
- p precedes q before r.
- p precedes q after r.
- after p and q eventually r.

Exercise 101 (Schwoon). Which of the following formulas of LTL are tautologies ? (A formula is a tautology if all computations satisfy it.) If the formula is not a tautology, give a computation that does not satisfy it.

- $\mathbf{G}p \rightarrow \mathbf{F}p$
- $\mathbf{G}(p \rightarrow q) \rightarrow (\mathbf{G}p \rightarrow \mathbf{G}q)$
- $\mathbf{F}(p \wedge q) \leftrightarrow (\mathbf{F}p \wedge \mathbf{F}q)$
- $\neg\mathbf{F}p \rightarrow \mathbf{F}\neg\mathbf{F}p$
- $(\mathbf{G}p \rightarrow \mathbf{F}q) \leftrightarrow (p \ \mathbf{U} \ (\neg p \vee q))$
- $(\mathbf{FG}p \rightarrow \mathbf{GF}q) \leftrightarrow \mathbf{G}(p \ \mathbf{U} \ (\neg p \vee q))$
- $\mathbf{G}(p \rightarrow \mathbf{X}p) \rightarrow (p \rightarrow \mathbf{G}p)$.

Part III

Pushdown Automata