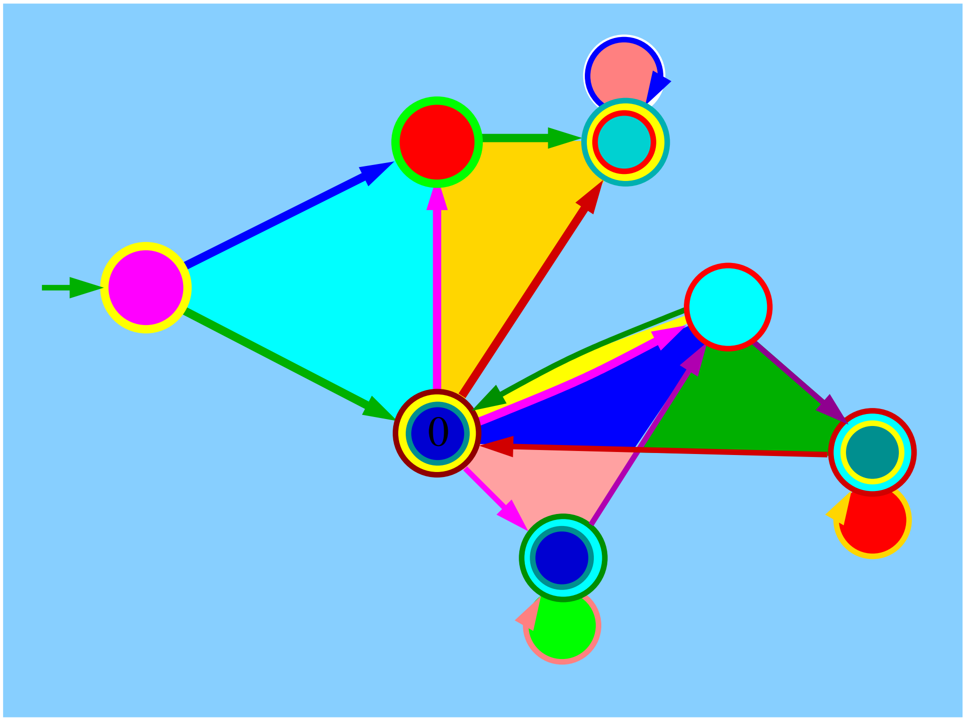
Automata theory

An algorithmic approach



Lecture Notes

Javier Esparza

February 6, 2018

Please read this!

Many years ago — I don't want to say how many, it's depressing — I taught a course on the automata-theoretic approach to model checking at the Technical University of Munich, basing it on lectures notes for another course on the same topic that Moshe Vardi had recently taught in Israel. Between my lectures I extended and polished the notes, and sent them to Moshe. At that time he and Orna Kupferman were thinking of writing a book, and the idea came up of doing it together. We made some progress, but life and other work got in the way, and the project has been postponed so many times that it I don't dare to predict a completion date.

Some of the work that got in the way was the standard course on automata theory in Munich, which I had to teach several times. The syllabus contained both automata on finite and infinite words, and for the latter I used our notes. Each time I had to teach the course again, I took the opportunity to add some new material about automata on finite words, which also required to reshape the chapters on infinite words, and the notes kept growing and evolving. Now they've reached the point where they are in sufficiently good shape to be shown not only to my students, but to a larger audience. So, after getting Orna and Moshe's very kind permission, I've decided to make them available here.

Despite several attempts I haven't yet convinced Orna and Moshe to appear as co-authors of the notes. But I don't give up: apart from the material we wrote together, their influence on the rest is much larger than they think. Actually, my secret hope is that after they see this material in my home page we'll finally manage to gather some morsels of time here and there and finish our joint project. If you think we should do so, tell us! Send an email to: vardi@cs.rice.edu, orna@cs.huji.ac.il, and esparza@in.tum.de.

Sources

I haven't yet compiled a careful list of the sources I've used, but I'm listing here the main ones. I apologize in advance for any omissions.

- The chapter on automata for fixed-length languages ("Finite Universes')' was very influenced by Henrik Reif Andersen's beautiful introduction to Binary Decision Diagrams, available at `www.itu.dk/courses/AVA/E2005/bdd-eap.pdf`.

- The short chapter on pattern matching is influenced by David Eppstein's lecture notes for his course on Design and Analysis of Algorithms, see `http://www.ics.uci.edu/ eppstein/teach.html`.

- As mentioned above, the chapters on operations for Büchi automata and applications to verification are heavily based on notes by Orna Kupferman and Moshe Vardi.

- The chapter on the emptiness problem for Büchi automata is based on several research papers:

  - Jean-Michel Couvreur: On-the-Fly Verification of Linear Temporal Logic. World Congress on Formal Methods 1999: 253-271
  - Jean-Michel Couvreur, Alexandre Duret-Lutz, Denis Poitrenaud: On-the-Fly Emptiness Checks for Generalized Bchi Automata. SPIN 2005: 169-184.
  - Kathi Fisler, Ranan Fraer, Gila Kamhi, Moshe Y. Vardi, Zijiang Yang: Is There a Best Symbolic Cycle-Detection Algorithm? TACAS 2001:420-434
  - Jaco Geldenhuys, Antti Valmari: More efficient on-the-fly LTL verification with Tarjan's algorithm. Theor. Comput. Sci. (TCS) 345(1):60-82 (2005)
  - Stefan Schwoon, Javier Esparza: A Note on On-the-Fly Verification Algorithms. TACAS 2005:174-190.

- The chapter on Linear Arithmetic is heavily based on the work of Bernard Boigelot, Pierre Wolper, and their co-authors, in particular the paper "An effective decision procedure for linear arithmetic over the integers and reals", published in ACM. Trans. Comput. Logic 6(3) in 2005.

Acknowledgments

First of all, thanks to Orna Kupferman and Moshe Vardi for all the reasons explained above (if you haven't read the section "Please read this" yet, please do it now!). Many thanks to Jörg Kreiker, Jan Kretinsky, and Michael Luttenberger for many discussions on the topic of this notes, and for their contributions to several chapters. All three of them helped me to teach the automata course of different ocassions. In particular, Jan contributed a lot to the chapter on pattern matching. Breno Faria helped to draw many figures. He was funded by a program of the Computer Science Department Technical University of Munich. Thanks also to Joe Bedard, Fabio Bove, Birgit Engelmann, Moritz Fuchs, Stefan Krusche, Philipp Müller, Martin Perzl, Marcel Ruegenberg, Franz Saller, Hayk Shoukourian, and Daniel Weißauer, who provided very helpful comments.

Contents

Why this book?

There are excellent textbooks on automata theory, ranging from course books for undergraduates to research monographies for specialists. Why another one?

During the late 1960s and early 1970s the main application of automata theory was the development of lexicographic analyzers, parsers, and compilers. Analyzers and parsers determine whether an input string conforms to a given syntax, while compilers transform strings conforming to a syntax into equivalent strings conforming to another. With these applications in mind, it is natural to look at automata as abstract machines that accept, reject, or transform input strings, and this view deply influenced the textbook presentation of automata theory. Results about the expressive power of machines, equivalences between models, and closure properties, received much attention, while constructions on automata, like the powerset or product construction, often played a subordinate rôle as proof tools. To give a simple example, in many textbooks of the time—and in later textbooks written in the same style—the product construction is not introduced as an algorithm that, given two NFAs recognizing languages L_1 and L_2, constructs a third NFA recognizing their intersection $L_1 \cap L_2$. Instead, the text contains a theorem stating that regular languages are closed under intersection, and the product construction is hidden in its proof. Moreover, it is not presented as an algorithm, but as the mathematical, static definition of the sets of states, transition relation, etc. of the product automaton. Sometimes, the simple but computationally important fact that only states reachable from the initial state need be constructed is not even mentioned.

I claim that this presentation style, summarized by the slogan *automata as abstract machines*, is no longer adequate. In the second half of the 1980s and in the 1990s program verification emerged as a new and exciting application of automata theory. Automata were used to describe the *behaviour*—or intended behaviour—of hardware and software systems, not their syntax, and this shift from syntax to semantics had important consequences. While automata for lexical or syntactical analysis typically have at most some thousands of states, automata for semantic descriptions can easily have tens of millions. In order to handle automata of this size it became imperative to pay special attention to efficient constructions and algorithmic issues, and research in this direction made great progress. Moreover, *automata on infinite words*, a class of automata models originally introduced in the 60s to solve abstract problems in logic, became necessary to specify and verify liveness properties of software. These automata run over words of infinite length, and so they can hardly be seen as machines accepting or rejecting an input: they could only do so after infinite time!

This book intends to reflect the evolution of automata theory. Modern automata theory puts more emphasis on algorithmic questions, and less on expressivity. This change of focus is captured by the new slogan *automata as data structures*. Just as hash tables and Fibonacci heaps are both adequate data structures for representing sets depending when the operations one needs are those of a dictionary or a priority queue, automata are the right data structure for represent sets and relations when the required operations are union, intersection, complement, projections and joins. In this view the algorithmic implementation of the operations gets the limelight, and, as a consequence, they constitute the spine of this book.

The shape of the book is also very influenced by two further design decisions. First, experience tells that automata-theoretic constructions are best explained by means of examples, and that examples are best presented with the help of pictures. Automata on words are blessed with a graphical representation of instantaneous appeal. We have invested much effort into finding illustrative, non-trivial examples whose graphical representation stillfits in one page. Second, for students learning directly from a book, solved exercises are a blessing, an easy way to evaluate progress. Moreover, thay can also be used to introduce topics that, for expository reasons, cannot be presented in the main text. The book contains a large number of solved exercises ranging from simple applications of algorithms to relatively involved proofs.

Chapter 1

Introduction and Outline

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

These lecture notes also deal with the problem of representing and manipulating sets, but with respect to a different set of operations: the *boolean operations of set theory* (union, intersection, and complement with respect to some universe set), some *tests* that check basic properties (if a set is empty, if it contains all elements of the universe, or if it is contained in another one), and operations on *relations*. Table 1.1 formally defines the operations to be supported, where U denotes some 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:

Observe that many other operations, for example 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. However, a simple cardinality argument shows that no data structure can provide finite representations of *all* infinite sets: an infinite universe has uncountably many subsets, but every data structure mapping sets to finite representations only has countably many instances. (Loosely speaking, there are more sets to be represented than representations available.) 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). These notes 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, represent and manipulate sets whose elements are encoded as *words*, i.e., as sequences of letters over an alphabet[1].

Any kind of object can be represented by a word, at least in principle. Natural numbers, for

[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.

Operations on sets		
Complement(X)	:	returns $U \setminus X$.
Intersection(X, Y)	:	returns $X \cap Y$.
Union(X, Y)	:	returns $X \cup Y$.

Tests on sets		
Member(x, X)	:	returns **true** if $x \in X$, **false** otherwise.
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.

Operations on relations		
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, S)	:	returns the relation $R \circ S = \{(x, z) \mid \exists y \in X\ (x, y) \in R \wedge (y, z) \in S\}$
Post(X, R)	:	returns the set $post_R(X) = \{y \in U \mid \exists x \in X\ (x, y) \in R\}$.
Pre(X, R)	:	returns the set $pre_R(X) = \{y \in U \mid \exists x \in X\ (y, x) \in R\}$.

Table 1.1: Operations and tests for manipulation of sets and relations

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. As a matter of fact, whenever a computer stores an object in a file, the computer is representing it as a word over some alphabet, like ASCII or Unicode. So word automata are a very general data 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. Typical examples are real numbers and non-terminating executions of a program. When objects cannot be represented by finite words, computers usually only represent some approximation: a float instead of a real number, or a finite prefix instead of a non-terminating computation. In the second part of the notes we show how to represent sets of infinite objects *exactly* using *automata on infinite words*. While 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".

Outline

Part I presents data structures and algorithms for the well-known class of regular languages.

Chapter 2 introduces the classical data structures for the representation of regular languages: regular expressions, deterministic finite automata (DFA), nondeterministic finite automata (NFA), and nondeterministic automata with ϵ-transitions. We refer to all of them as *automata*. The chapter presents some examples showing how to use automata to finitely represent sets of words, numbers or program states, and describes conversions algorithms between the representations. All algorithms are well known (and can also be found in other textbooks) with the exception of the algorithm for the elimination of ϵ-transitions.

Chapter 3 address the issue of finding small representations for a given set. It shows that there is a unique minimal representation of a language as a DFA, and introduces the classical minimization algorithms. It then shows how to the algorithms can be extended to reduce the size of NFAs.

Chapter 4 describes algorithms implementing boolean set operations and tests on DFAs and NFAs. It includes a recent, simple improvement in algorithms for universality and inclusion.

Chapter 5 presents a first, classical application of the techniques and results of Chapter 4: Pattern Matching. Even this well-known problem gets a new twist when examined from the automata-as-data-structures point of view. The chapter presents the Knuth-Morris-Pratt algorithm as the design of a new data structure, lazy DFAs, for which the membership operation can be performed very efficiently.

Chapter 6 shows how to implement operations on relations. It discusses the notion of encoding (which requires more care for operations on relatrions than for operations on sets), and introduces transducers as data structure.

Chapter 7 presents automata data structures for the important special case in which the universe U of objects is finite. In this case all objects can be encoded by words of the same length, and the set and relation operations can be optimized. In particular, one can then use minimal DFAs as data structure, and directly implement the algorithms without using any minimization algorithm. In the second part of the chapter, we show that (ordered) Binary Decision Diagrams (BDDs) are just a further optimization of minimal DFAs as data structure. We introduce a slightly more general class of deterministic automata, and show that the minimal automaton in this more general class (which is also unique) has at most as many states as the minimal DFA. We then show how to implement the set and relation operations for this new representation.

Chapter 8 applies nearly all the constructions and algorithms of previous chapters to the problem of verifying safety properties of sequential and concurrent programs with bounded-range variables. In particular, the chapter shows how to model concurrent programs as networks of automata, how to express safety properties using automata or regular expressions, and how to automatically check the properties using thealgorithmic constructions of previous chapters.

Chapter 9 introduces first-order logic (FOL) and monadic-second order logic (MSOL) on words as representation allowing us to described a regular language as the set of words satisfying a property. The chapter shows that FOL cannot describe all regular languages, and that MSOL does.

Chapter 10 introduces Presburger arithmetic, and an algorithm to computes an automaton encoding all the solutions of a given formula. In particular, it presents an algorithm to compute an automaton for the solutions of a linear inequality over the naturals or over the integers.

Part II presents data structures and algorithms for ω-regular languages.

Chapter 11 introduces ω-regular expressions and several different classes of ω-automata: deterministic and nondterministic Büchi, generalized Büchi, co-Büchi, Muller, Rabin, and Street automata. It explains the advantages and disadvantages of each class, in particular whether the automata in the class can be determinized, and presents conversion algorithms between the classes.
Chapter 12 presents implementations of the set operations (union, intersection and complementation) for Büchi and generalized Büchi automata. In particular, it presents in detail a complementation algorithm for Büchi automata.
Chapter 13 presents different implementations of the emptiness test for Büchi and generalized Büchi automata. The first part of the chapter presents two linear-time implementations based on depth-first-search (DFS): the nested-DFS algorithm and the two-stack algorithm, a modification of Tarjan's algorithm for the computation of strongly connected components. The second part presents further implemntations based on breadth-first-search.
Chapter 14 applies the algorithms of previous chapters to the problem of verifying liveness properties of programs. After an introductory example, the chapter presents Linear Temporal Logic as property specification formalism, and shows how to algorithmically translate a formula into an equivalent Büchi automaton, that is, a Büchi automaton recognizing the language of all words satisfying the formula. The verification algorithm can then be reduced to a combination of the boolean operations and emptiness check.
Chapter 15 extends the logic approach to regular languages studied in Chapters 9 and 10 to ω-words. The first part of the chapter introduces monadic second-order logic on ω-words, and shows how to construct a Büchi automaton recognizing the set of words satisfying a given formula. The second part introduces linear arithmetic, the first-order theory of thereal numbers with addition, and shows how to construct a Büchi automaton recognizing the encodings of all the real numbers satisfying a given formula.

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 Regular expressions: a language to describe languages

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 $a_1a_2 \dots a_n$ 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 $w_1w_2 = a_1 \dots a_nb_1 \dots b_m$, sometimes also denoted by $w_1 \cdot w_2$. Notice that $\epsilon \cdot w = w = w \cdot \epsilon = w$. For every word w, we define $w^0 = \epsilon$ and $w^{k+1} = w^kw$.

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 $L_1 \cdot L_2 = \{w_1w_2 \in \Sigma^* \mid w_1 \in L_1, w_2 \in L_2\}$. 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 $i \geq 0$.

In this book we use automata to represent sets of objects encoded as languages. Languages can be mathematically described using the standard notation of set theory, but this is often cumbersome. For a concise description of simple languages, *regular expressions* are often the most suitable notation.

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

$$r ::= \emptyset \mid \varepsilon \mid a \mid r_1r_2 \mid r_1 + r_2 \mid r^*$$

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 by*

$$\begin{array}{lll} L(\emptyset) = \emptyset & L(r_1 r_2) = L(r_1) \cdot L(r_2) & L(r^*) = L(r)^* \\ L(\varepsilon) = \{\varepsilon\} & L(r_1 + r_2) = L(r_1) \cup L(r_2) & \\ L(a) = \{a\} & & \end{array}$$

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

We often abuse language, and identify a regular expression and its language. For instance, when there is no risk of confusion we write "the language r" instead of "the language $L(r)$."

Example 2.2 Let $\Sigma = \{0, 1\}$. Some examples of languages expressible by regular expressions are:

- The set of all words: $(0 + 1)^*$. We often use Σ as an abbreviation of $(0 + 1)$, and so Σ^* as an abreviation of $(0 + 1)^*$.
- The set of all words of length at most 4: $(0 + 1 + \varepsilon)^4$.
- The set of all words that begin and end with 0: $0\Sigma^*0$.
- The set of all words containing at least one pair of 0s exactly 5 letters apart. $\Sigma^*0\Sigma^40\Sigma^*$.
- The set of all words containing an even number of 0s: $(1^*01^*01^*)^*$.
- The set of all words containing an even number of 0s and an even number of 1s: $(00 + 11 + (01 + 10)(00 + 11)^*(01 + 10))^*$.

□

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.

2.2.1 Deterministic finite automata

From an operational point of view, a deterministic automaton can be seen as the control unit of a machine that reads input from a *tape* divided into *cells* by means of a *reading head* (see Figure 2.1). Initially, the automaton is in the initial state, the tape contains the word to be read, and the reading head is positioned on the first cell of the tape, see Figure 2.1. At each step, the machine reads the content of the cell occupied by the reading head, updates the current state according to the transition function, and advances the head one cell to the right. The machine accepts a word if the state reached after reading it completely is final.

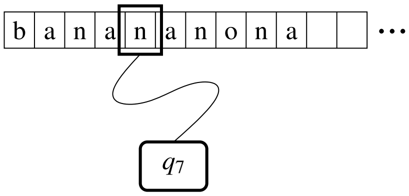


Figure 2.1: Tape with reading head.

Definition 2.3 *A* deterministic automaton (DA) *is a tuple* $A = (Q, \Sigma, \delta, q_0, F)$*, where*

- Q *is a nonempty 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* $a_0 a_1 \ldots a_{n-1}$ *is a sequence* $q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} q_2 \ldots \xrightarrow{a_{n-1}} q_n$*, such that* $q_i \in Q$ *for* $0 \le i \le n$*, and* $\delta(q_i, a_i) = q_{i+1}$ *for* $0 \le i < n - 1$*. A run is* accepting *if* $q_n \in F$*. The automaton* A accepts *a word* $w \in \Sigma^*$ *if it has 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.

Graphically, non-final states of a DFA are represented by circles, and final states by double circles (see the example below). The transition function is represented by labeled directed edges: if $\delta(q, a) = q'$ then we draw an edge from q to q' labeled by a. We also draw an edge into the initial state.

Example 2.4 Figure 2.2 shows the graphical representation of the DFA $A = (Q, \Sigma, \delta, q_0, F)$, where $Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{a, b\}$, $F = \{q_0\}$, and δ is given by the following table

$$\begin{array}{llll} \delta(q_0, a) = q_1 & \delta(q_1, a) = q_0 & \delta(q_2, a) = q_3 & \delta(q_3, a) = q_2 \\ \delta(q_0, b) = q_3 & \delta(q_1, b) = q_2 & \delta(q_2, b) = q_1 & \delta(q_3, b) = q_0 \end{array}$$

The runs of A on *aabb* and *abbb* are

$$\begin{array}{l} q_0 \xrightarrow{a} q_1 \xrightarrow{a} q_0 \xrightarrow{b} q_3 \xrightarrow{b} q_0 \\ q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_2 \xrightarrow{b} q_1 \xrightarrow{b} q_2 \end{array}$$

The first one is accepting, but the second one is not. The DFA recognizes 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.

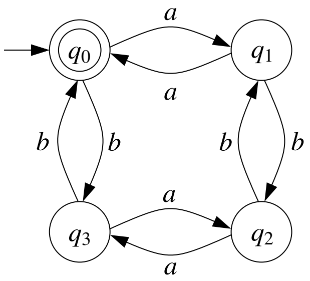


Figure 2.2: A DFA

□

Trap states. Consider the DFA of Figure 2.3 over the alphabet $\{a, b, c\}$. The automaton recognizes the language $\{ab, ba\}$. The pink state on the right is often called a *trap state* or a *garbage collector*: if a run reaches this state, it gets trapped in it, and so the run cannot be accepting. DFAs often have a trap state with many ingoing transitions, and this makes difficult to find a nice graphical representation. So when drawing DFAs we often omit the trap state. For instance, we only draw the black part of the automaton in Figure 2.3. Notice that no information is lost: if a state q has no outgoing transition labeled by a, then we know that $\delta(q, a) = q_t$, where q_t is the trap state.

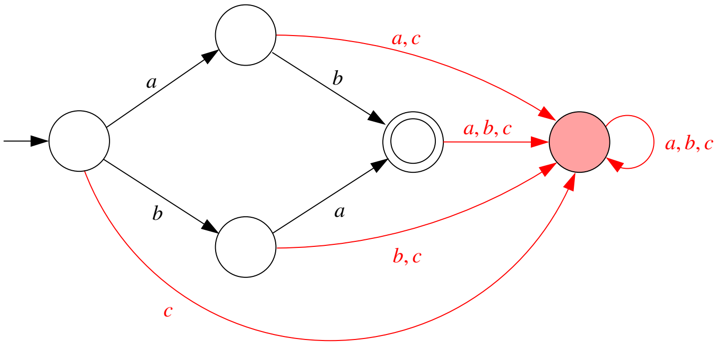


Figure 2.3: A DFA with a trap state

Using DFAs as data structures

In this book we look at DFAs as a data structure. A DFA is a finite representation of a possibly infinite language. In applications, a suitable encoding is used to represent objects (numbers, programs, relations, tuples . . .) as words, and so a DFA actually represents a possibly infinite set of objects. Here are four examples of DFAs representing interesting sets.

Example 2.5 The DFA of Figure 2.4 (drawn without the trap state!) recognizes the strings over the alphabet $\{-, \cdot, 0, 1, \ldots, 9\}$ that encode real numbers with a finite decimal part. We wish to exclude 002, −0, or 3.10000000, but accept 37, 10.503, or −0.234 as correct encodings. A description of the strings in English is rather long: a string encoding a number consists of an integer part, followed by a possibly empty fractional part; the integer part consists of an optional minus sign, followed by a nonempty sequence of digits; if the first digit of this sequence is 0, then the sequence itself is 0; if the fractional part is nonempty, then it starts with ., followed by a nonempty sequence of digits that does not end with 0; if the integer part is −0, then the fractional part is nonempty.

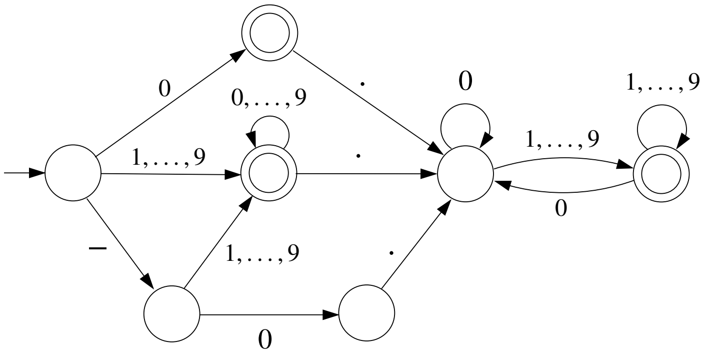


Figure 2.4: A DFA for decimal numbers

□

Example 2.6 The DFA of Figure 2.5 recognizes the binary encodings of all the multiples of 3. For instance, it recognizes 11, 110, 1001, and 1100, which are the binary encodings of 3, 6, 9, and 12, respectively, but not, say, 10 or 111. □

Example 2.7 The DFA of Figure 2.6 recognizes all the nonnegative integer solutions of the inequation $2x - y \leq 2$, using the following encoding. The alphabet of the DFA has four letters, namely

$$\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}0\\1\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}, \text{ and } \begin{bmatrix}1\\1\end{bmatrix} .$$

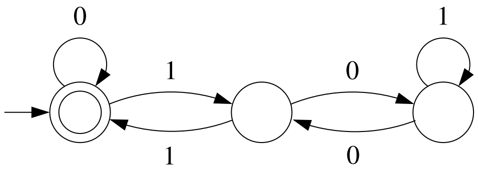


Figure 2.5: A DFA for the multiples of 3 in binary

A word like

$$\begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}$$

encodes a pair of numbers, given by the top and bottom rows, 101100 and 010011. The binary encodings start with the least significant bit, that is

$$\begin{array}{lll} 101100 & \text{encodes} & 2^0 + 2^2 + 2^3 = 13, \text{ and} \\ 010011 & \text{encodes} & 2^1 + 2^4 + 2^5 = 50 \end{array}$$

We see this as an encoding of the valuation $(x, y) := (13, 50)$. This valuation satisfies the inequation, and indeed the word is accepted by the DFA. □

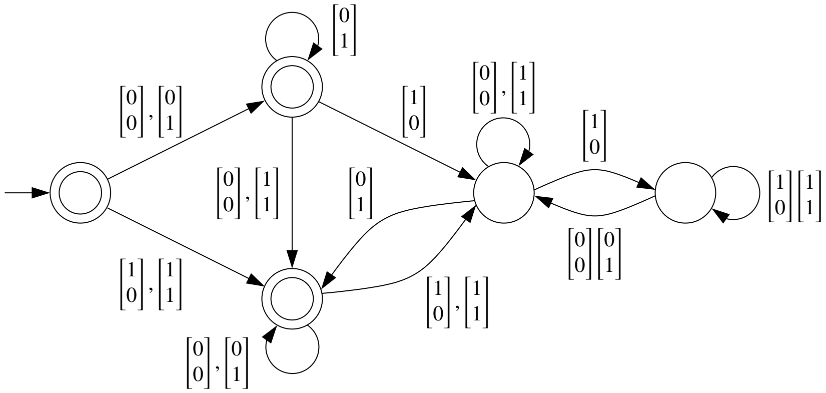

$\begin{bmatrix}0\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix},\begin{bmatrix}0\\1\end{bmatrix}$
$\begin{bmatrix}1\\0\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix},\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}1\\0\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix},\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}0\\1\end{bmatrix}$
$\begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}$
$\begin{bmatrix}1\\0\end{bmatrix},\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}1\\0\end{bmatrix},\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix},\begin{bmatrix}0\\1\end{bmatrix}$

Figure 2.6: A DFA for the solutions of $2x - y \leq 2$.

Example 2.8 Consider the following program with two boolean variables x, y:

```
while $x = 1$ do
   if $y = 1$ then
      $x \leftarrow 0$
   $y \leftarrow 1 - x$
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. 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 DFA of Figure 2.7 recognizes all reachable configurations of the program. For instance, the DFA accepts $[5, 0, 1]$, indicating that it is possible to reach the last line of the program with values $x = 0$, $y = 1$. □

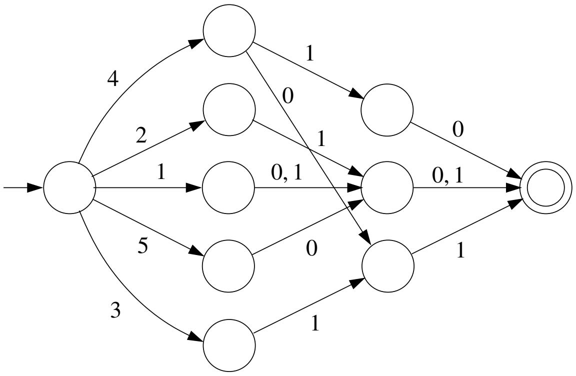


Figure 2.7: A DFA for the reachable configurations of the program of Example 2.8

2.2.2 Non-deterministic finite automata

In a deterministic automaton the next state is completely determined by the current state and the letter read by the head. In particular, this implies that the automaton has exactly one run for each word. Nondeterministic automata have the possibility to choose the state out of a set of candidates (which may also be empty), and so they may have zero, one, or many runs on the same word. The automaton is said to accept a word if *at least one* of these runs is accepting.

Definition 2.9 *A* non-deterministic automaton (NA) *is a tuple* $A = (Q, \Sigma, \delta, Q_0, F)$*, where* Q*,* Σ*, and* F *are as for DAs,* Q_0 *is a nonempty set of* initial states *and*

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

A run *of* A *on input* $a_0 a_1 \ldots a_n$ *is a sequence* $p_0 \xrightarrow{a_0} p_1 \xrightarrow{a_1} p_2 \ldots \xrightarrow{a_{n-1}} p_n$*, such that* $p_i \in Q$ *for* $0 \leq i \leq n$*,* $p_0 \in Q_0$*, and* $p_{i+1} \in \delta(p_i, a_i)$ *for* $0 \leq i < n-1$*. A run is* accepting *if* $p_n \in F$*.*

A word $w \in \Sigma^*$ *is* accepted *by* A *if at least one run of* A *on* w *is accepting. The* language recognized *by* A *is the set* $w \in \Sigma^* L(A) = \{w \in \Sigma^* \mid w \text{ is accepted by } A\}$*.*

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.

If a NA has several initial states, then its language is the union of the sets of words accepted by runs starting at each initial state.

Example 2.10 Figure 2.8 shows a NFA $A = (Q, \Sigma, \delta, Q_0, F)$ where $Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{a, b\}$, $Q_0 = \{q_0\}$, $F = \{q_3\}$, and the transition relation δ is given by the following table

$$\begin{array}{llll} \delta(q_0, a) = \{q_1\} & \delta(q_1, a) = \{q_1\} & \delta(q_2, a) = \emptyset & \delta(q_3, a) = \{q_3\} \\ \delta(q_0, b) = \emptyset & \delta(q_1, b) = \{q_1, q_2\} & \delta(q_2, b) = \{q_3\} & \delta(q_3, b) = \{q_3\} \end{array}$$

A has no run for any word starting with a b. It has exactly one run for abb, and four runs for $abbb$, namely

$$\begin{array}{ll} q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_1 \xrightarrow{b} q_1 \xrightarrow{b} q_1 & q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_1 \xrightarrow{b} q_1 \xrightarrow{b} q_2 \\ q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_1 \xrightarrow{b} q_2 \xrightarrow{b} q_3 & q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_2 \xrightarrow{b} q_3 \xrightarrow{b} q_3 \end{array}$$

Two of these runs are accepting, the other two are not. $L(A)$ is the set of words that start with a and contain two consecutive bs. □

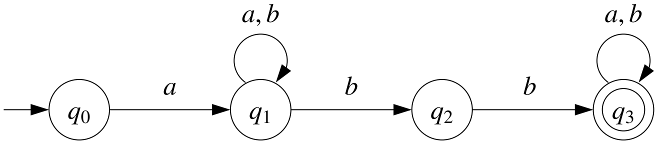


Figure 2.8: A NFA.

After a DA reads a word, we know if it belongs to the language or not. This is no longer the case for NAs: if the run on the word is not accepting, we do not know anything; there might be a different run leading to a final state. So NAs are not very useful as language acceptors. However, they are very important. From the operational point of view, it is often easier to find a NFA for a given language than to find a DFA, and, as we will see later in this chapter, NFAs can be *automatically* transformed into DFAs. From a data structure point of view, there are two further reasons to study NAs. First, many sets can be represented far more compactly as NFAs than as DFAs. So using NFAs may save memory. Second, and more importantly, when we describe DFA- and NFA-implementations of operations on sets and relations in Chapters 4 and Chapter 6 , we will see that one of them takes as input a DFA and returns a NFA. Therefore, NFAs are not only convenient, but also necessary to obtain a data structure implementing all operations.

2.2.3 Non-deterministic finite automata with ϵ-transitions

The state of an NA can only change by reading a letter. NAs with ϵ-transitions can also change their state "spontaneously", by executing an "internal" transition without reading any input. To emphasize this we label these transitions with the empty word ϵ (see Figure 2.9).

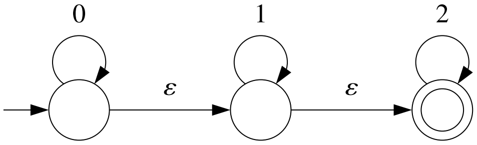


Figure 2.9: A NFA-ϵ.

Definition 2.11 *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*

- $\delta \colon Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$ *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 there exist numbers* $k_0, k_1, \dots, k_n \geq 0$ *such that* A *has an accepting run on the word*

$$\epsilon^{k_0} a_1 \epsilon^{k_1} \dots \epsilon^{k_{n-1}} a_n \epsilon^{k_n} \in (\Sigma \cup \{\epsilon\})^* .$$

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

Notice that the number of runs of a NA-ϵ on a word may be infinite. This is the case when some cycle of the NA-ϵ only contains ε-transitions, and some final state is reachable from the cycle.

NA-ϵ are useful as intermediate representation. In particular, later in this chapter we automatically transform a regular expresion into a NFA in two steps; first we translate the expression into a NFA-ϵ, and then we translate the NFA-ϵ into a NFA.

2.2.4 Non-deterministic finite automata with regular expressions

We generalize NA-ϵ even further. Botzh letters and ϵ are instances of regular expressions. Now we allow *arbitrary*regular expressions as transition labels (see Figure 2.10).

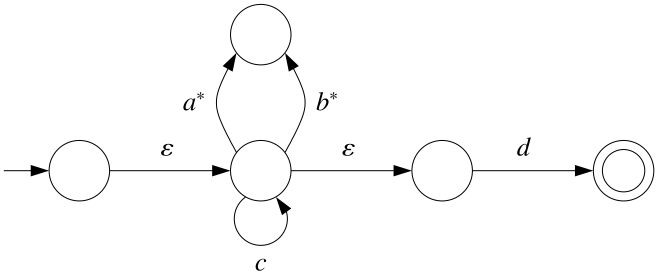


Figure 2.10: A NFA with transitions labeled by regular expressions.

A run leading to a final state accepts all the words of the regular expression obtained by concatenating all the labels of the transitions of the run into one single regular expression. We call these automata *NA-reg*. They are very useful to formulate conversion algorithms between automata and regular expressions, because they generalize both. Indeed, a regular expression can be seen as an automaton with only one transition leading from the initial state to a final state, and labeled by the regular expression.

Definition 2.12 *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 NAs. 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.*

2.2.5 A normal form for automata

For any of the automata classes we have introduced, if a state is not reachable from any initial state, then removing it does not change the language accepted by the automaton. We say that an automaton is in normal form if every state is reachable from initial ones.

Definition 2.13 *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 some initial state.*

Obviously, for every automaton there is an equivalent automaton of the same kind in normal form. In this book we follow this convention:

> **Convention:** Unless otherwise stated, we assume that automata are in normal form. In particular, we assume that if an automaton is an input to an algorithm, then the automaton is in normal form. If the output of an algorithm is an automaton, then the algorithm is expected to produce an automaton in normal form. This condition is a proof obligation when showing that the algorithm is correct.

2.3 Conversion Algorithms between Finite Automata

We show 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 *powerset 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 some initial state 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 (A starts at some 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 $Q' \subseteq Q$. Notice that we may have $Q' = \emptyset$; in this case, $\emptyset$ is a state of B, and, since $\Delta(\emptyset, a) = \emptyset$ for every $a \in \Delta$, a "trap" state.

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. The following conversion algorithm constructs only the reachable states.

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

```
$\mathcal{Q}, \Delta, \mathcal{F} \leftarrow \emptyset$; $q_0 \leftarrow Q_0$
$\mathcal{W} = \{Q_0\}$
while $\mathcal{W} \neq \emptyset$ do
  pick $Q'$ from $\mathcal{W}$
  add $Q'$ to $\mathcal{Q}$
  if $Q' \cap F \neq \emptyset$ then add $Q'$ to $\mathcal{F}$
  for all $a \in \Sigma$ do
    $Q'' \leftarrow \bigcup_{q \in Q'} \delta(q, a)$
    if $Q'' \notin \mathcal{Q}$ then add $Q''$ to $\mathcal{W}$
    add $(Q', a, Q'')$ to $\Delta$
```

The algorithm is written in pseudocode, with abstract sets as data structure. Like nearly all the algorithms presented in the next chapters, it is a *workset algorithm*. Workset algorithms maintain a set of objects, the *workset*, waiting to be processed. Like in mathematical sets, the elements of the workset are not ordered, and the workset contains at most one copy of an element (i.e., if an element already in the workset is added to it again, the workset does not change). For most of the algorithms in this book, the workset can be implemented as a hash table.

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

Figure 2.11 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 workset 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 $\Sigma = \{a, b\}$

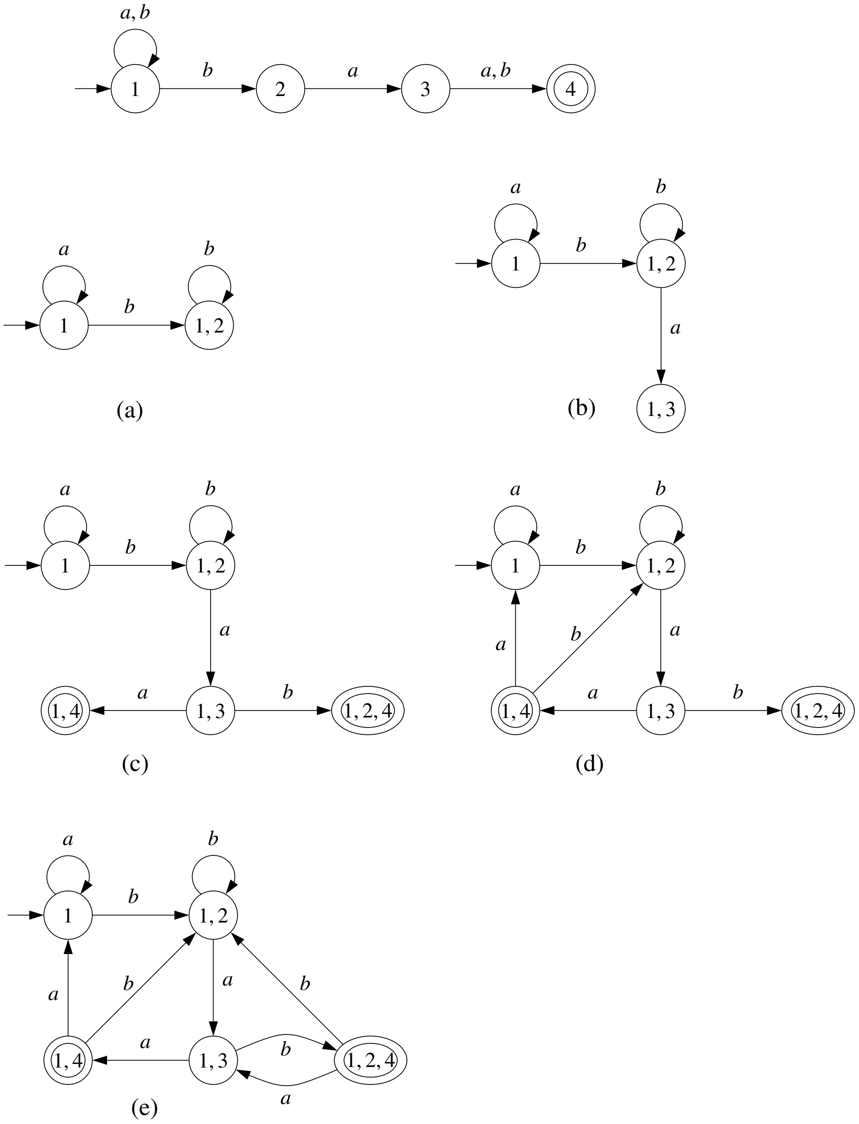


Figure 2.11: Conversion of a NFA into a DFA.

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 2.12(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 2.12(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 $a_1a_2\ldots a_n$ and it reads the letter a_{n+1}, then it moves to the state storing $a_2\ldots a_{n+1}$. States are final if the first letter they store is an a. The interpreted version of the DFA is shown on right of Figure 2.12(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 $\hat{\delta}(q, w\,\sigma) = \delta(\hat{\delta}(q,w),\sigma)$ for all $w \in \Sigma^*$ and for all $\sigma \in \Sigma$. Since $|Q| < 2^n$, there must be two words $u\,a\,v_1$ and $u\,b\,v_2$ of length n for which $\hat{\delta}(q_0, u\,a\,v_1) = \hat{\delta}(q_0, u\,b\,v_2)$. But then we would have that $\hat{\delta}(q_0, u\,a\,v_1\,u) = \hat{\delta}(q_0, u\,b\,v_2\,u)$; that is, either both $u\,a\,v_1\,u$ and $u\,b\,v_2\,u$ are accepted by A_n or neither do. Since, however, $|a\,v_1\,u| = |b\,v_2\,u| = n$, 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 $\Sigma \cup \{\epsilon\}$.

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.

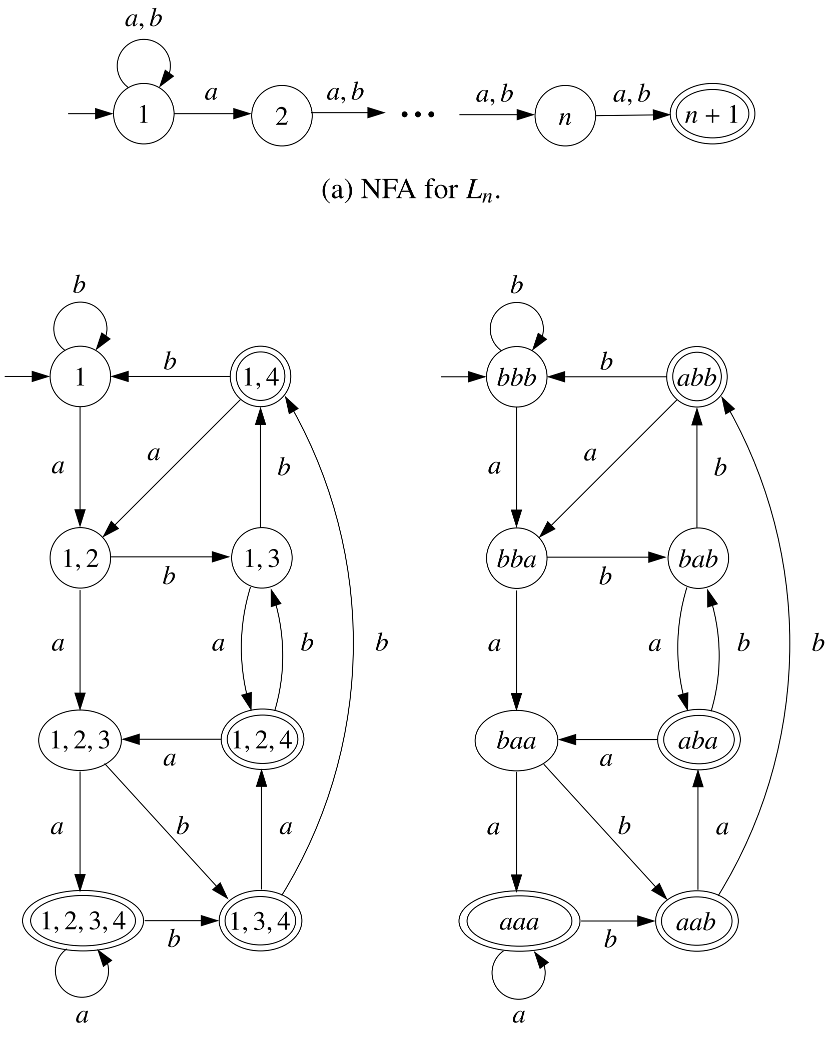
The new transitions are *shortcuts*: If A has transitions (q, α, q') and (q', β, q'') such that $\alpha = \epsilon$ or $\beta = \epsilon$, then the shortcut $(q, \alpha\beta, q'')$ is added. (Notice that either $\alpha\beta = a$ for some $a \in \Sigma$, or $\alpha\beta = \epsilon$.) Shortcuts may generate 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. If A has a run accepting a *nonempty* word before saturation, 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

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

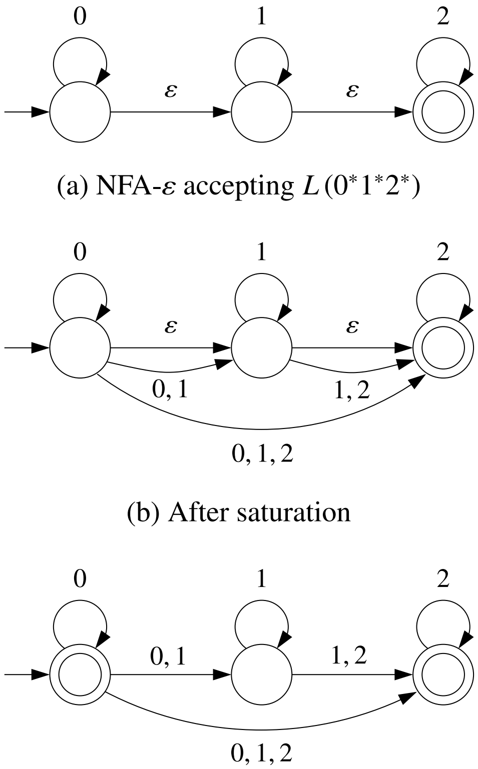
However, removing ε-transitions immediately after saturation may not preserve the language. The NFA-ϵ of Figure 2.13(a) accepts ε. After saturation we get the NFA-ϵ of Figure 2.13(b). Removing all ε-transitions yields an NFA that no longer accepts ε. To solve this problem, if A



(a) NFA for L_n.

(b) DFA for L_3 and interpretation.

Figure 2.12: NFA for L_n, and DFA for L_3.



(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 2.13: Conversion of an NFA-ϵ into an NFA by shortcutting ϵ-transitions.

accepts ε from some initial state, then we mark that state as final, which clearly does not change the language. To decide whether A accepts ε, we check if some state reachable from some initial state by a sequence of ε-transitions is final. Figure 2.13(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 four phases: saturation, ϵ-check, removal of all ϵ-transitions, and normalization.

We show that it is possible to carry all four steps in a single pass. We present a workset algorithm *NFAϵtoNFA* that carries the ϵ-check while saturating, and generates only the reachable states. Furthermore, the algorithm avoids constructing some redundant shortcuts. For instance, for the NFA-ϵ of Figure 2.13(a) the algorithm does not construct the transition labeled by 2 leading from the state in the middle to the state on the right. the pseudocode for the algorithm is as follows, where $\alpha, \beta \in \Sigma \cup \{\epsilon\}$, and $a \in \Sigma$.

```
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  $Q'_0 \leftarrow Q_0$
 2  $Q' \leftarrow Q_0$; $\delta' \leftarrow \emptyset$; $F' \leftarrow F \cap Q_0$
 3  $\delta'' \leftarrow \emptyset$; $W \leftarrow \{(q, \alpha, q') \in \delta \mid q \in Q_0\}$
 4  while $W \neq \emptyset$ do
 5      pick $(q_1, \alpha, q_2)$ from $W$
 6      if $\alpha \neq \epsilon$ then
 7          add $q_2$ to $Q'$; add $(q_1, \alpha, q_2)$ to $\delta'$; if $q_2 \in F$ then add $q_2$ to $F'$
 8          for all $q_3 \in \delta(q_2, \epsilon)$ do
 9              if $(q_1, \alpha, q_3) \notin \delta'$ then add $(q_1, \alpha, q_3)$ to $W$
10          for all $a \in \Sigma, q_3 \in \delta(q_2, a)$ do
11              if $(q_2, a, q_3) \notin \delta'$ then add $(q_2, a, q_3)$ to $W$
12      else $/ * \alpha = \epsilon * /$
13          add $(q_1, \alpha, q_2)$ to $\delta''$; if $q_2 \in F$ then add $q_1$ to $F'$
14          for all $\beta \in \Sigma \cup \{\varepsilon\}, q_3 \in \delta(q_2, \beta)$ do
15              if $(q_1, \beta, q_3) \notin \delta' \cup \delta''$ then add $(q_1, \beta, q_3)$ to $W$
```

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

Proposition 2.14 *Let A be a NFA-ϵ, and let B = NFAϵtoNFA(A). Then B is a NFA and $L(A) = L(B)$.*

Proof: To show that the algorithm terminates, observe 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 workset, the algorithm eventually exits the loop.

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 some state of $Q'_0 = Q_0$ 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 $q_1 \in Q_0$, and if $\alpha \neq \varepsilon$, then q_2 is reachable in B (after termination). Since new states are added to Q' only at line 7, applying the invariant we get that every state of Q' is reachable in B from some state in Q_0.

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 is shown by inspection. For the inclusion $L(A) \subseteq L(B)$, we first claim that $\epsilon \in L(A)$ implies $\epsilon \in L(B)$. Let $q_0 \xrightarrow{\epsilon} q_1 \ldots q_{n-1} \xrightarrow{\epsilon} q_n$ 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 W (and so eventually picked from it), which implies that q_0 is eventually added to F' at line 13. If $n = 1$, then (q_0, ϵ, q_n) is added to W 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 so (q_0, ε, q_n) is added to W at line 15, ad teh claim is proved. We now show 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

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

such that $q_m \in F$. We have just proved that a transition (q_0, ϵ, 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

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

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 $|Q| \cdot |\Sigma| \cdot |\delta|$ pairs (because for a fixed transition $(q_2, \beta, q_3) \in \delta$ there are $|Q|$ possibilities for q_1 and $|\Sigma|$ possibilities for α). The runtime is dominated by the processing of the pairs, and so it is $\mathcal{O}(|Q| \cdot |\Sigma| \cdot |\delta|)$.

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 2.12. 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.

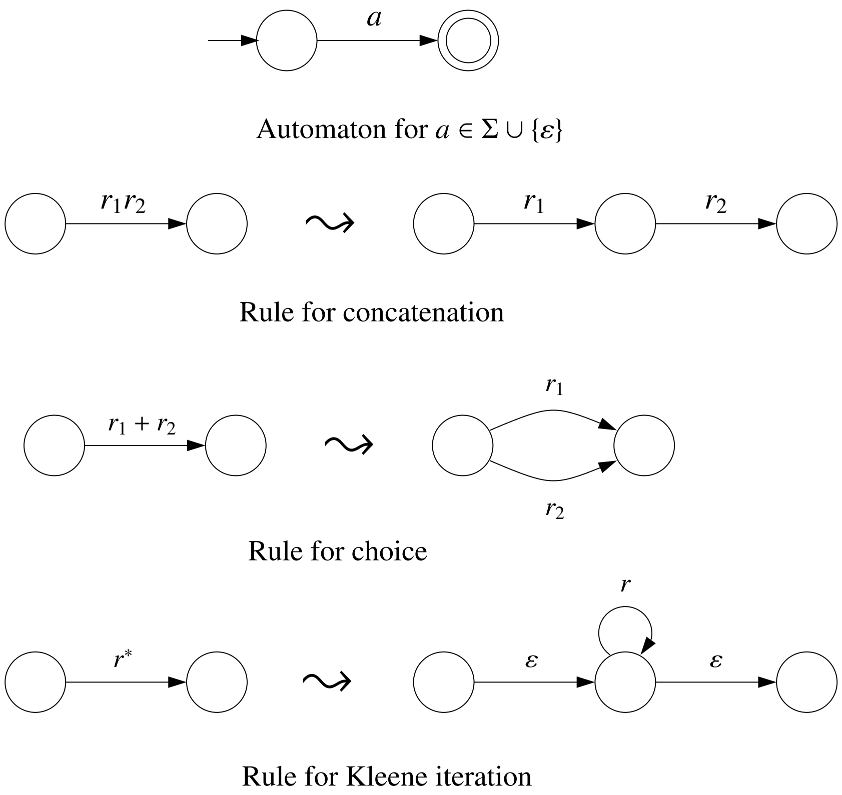
We present algorithms that, given an NFA-reg belonging to one of this subclasses, produces a sequence of NFA-regs, each one recognizing the same language as its predecessor in the sequence, and ending in an NFA-reg of the other subclass.

2.4.1 From regular expressions to NFA-ϵ

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

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

Since the left- and right-hand-sides of each rule denote the same language, the regular expressions before and after preprocessing denote the same language. Moreover, if r is the resulting regular expression, then either $r = \emptyset$, or r does not contain any occurrence of the $\emptyset$ symbol. In the first 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 2.14.



Automaton for $a \in \Sigma \cup \{\varepsilon\}$

Rule for concatenation

Rule for choice

Rule for Kleene iteration

Figure 2.14: Rules converting a regular expression given as NFA-reg into an NFA-ϵ.

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.

The two ϵ-transitions of the rule for Kleen iteration guarantee that the automata before and after applying the rule are equivalent, even if the source and target states of the transition labeled by $r^\star$ have other incoming or outgoing transitions. If the source state has no other outgoing transitions, then we can omit the first ϵ-transition. If the target state has no other incoming transitions, then we can omit the second.

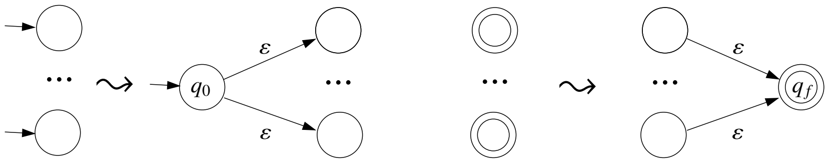
Example 2.15 Consider the regular expression $(a^*b^* + c)^*d$. The result of applying the transformation rules is shown in Figure 2.15 on page 35. □

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.

2.4.2 From NFA-ϵ to regular expressions

Given an NFA-ϵ A, we transform it into an equivalent NFA-reg A_r with two states and one single transition, labeled by a regular expression r. It is again convenient to apply some preprocessing to guarantee that the NFA-ϵ has a single initial state without incoming transitions, and a single final state without outgoing transitions:

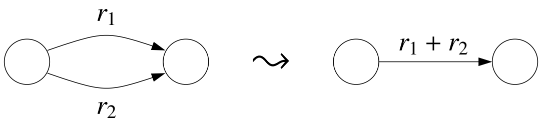
- If A has more than one initial state, or some initial state has an incoming transition, then: Add a new initial state q_0, add ε-transitions leading from q_0 to each initial state, and replace the set of initial states by $\{q_0\}$.
- If A has more than one final state, or some final state has an outgoing transition, then: Add a new state q_f, add ε-transitions leading from each final state to q_f, and replace the set of final states by $\{q_f\}$.



Rule 1: Preprocessing

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: At most one transition between two states

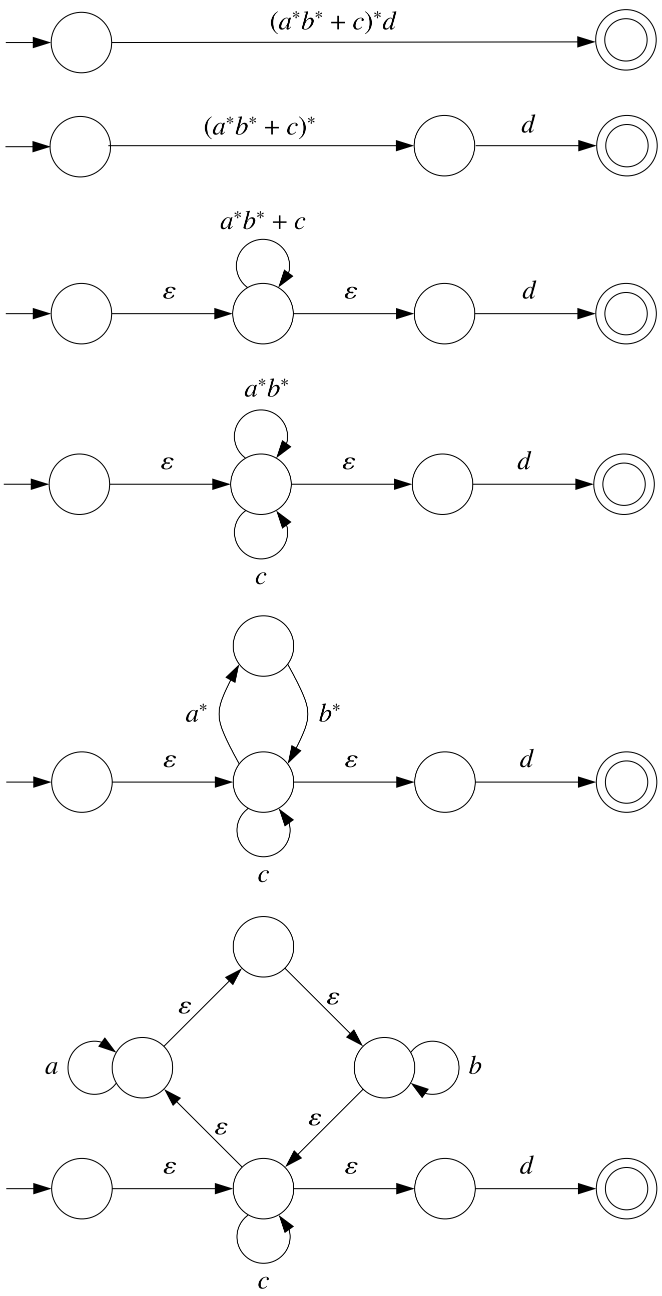
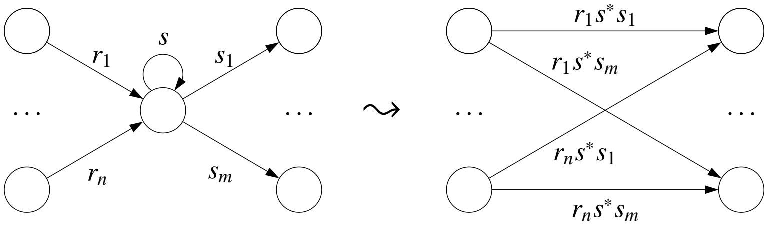


Figure 2.15: The result of converting $(a^*b^* + c)^*d$ into an NFA-ϵ.

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

- 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.



Rule 3: Removing a state

At the end of the last phase we are left with a NFA-reg having exactly two states, the unique initial state q_0 and the unique final state q_f. Moreover, q_0 has no incoming transitions and q_f has no outgoing transitions, because it was initially so and the application of the rules cannot change it. After applying Rule 2 exhaustively, there is exactly one transition from q_0 to q_f. The complete algorithm is:

```
NFAtoRE(A)
Input: NFA-$\epsilon$ $A = (Q, \Sigma, \delta, Q_0, F)$
Output: regular expression $r$ with $L(r) = L(A)$
apply Rule 1;
let $q_0$ and $q_f$ be the initial and final states of $A$;
while $Q \setminus \{q_0, q_f\} \neq \emptyset$ do
    apply exhaustively Rule 2
    pick $q$ from $Q \setminus \{q_0, q_f\}$
    apply Rule 3 to $q$
apply exhaustively Rule 2
return the label of the (unique) transition
```

Example 2.16 Consider the automaton of Figure 2.16(a) on page 37. Parts (b) to(f) of the figure show 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. □

[1] Notice that it can have at most one, because otherwise we would have two parallel edges, contradicting that Rule 2 was applied axhaustively.

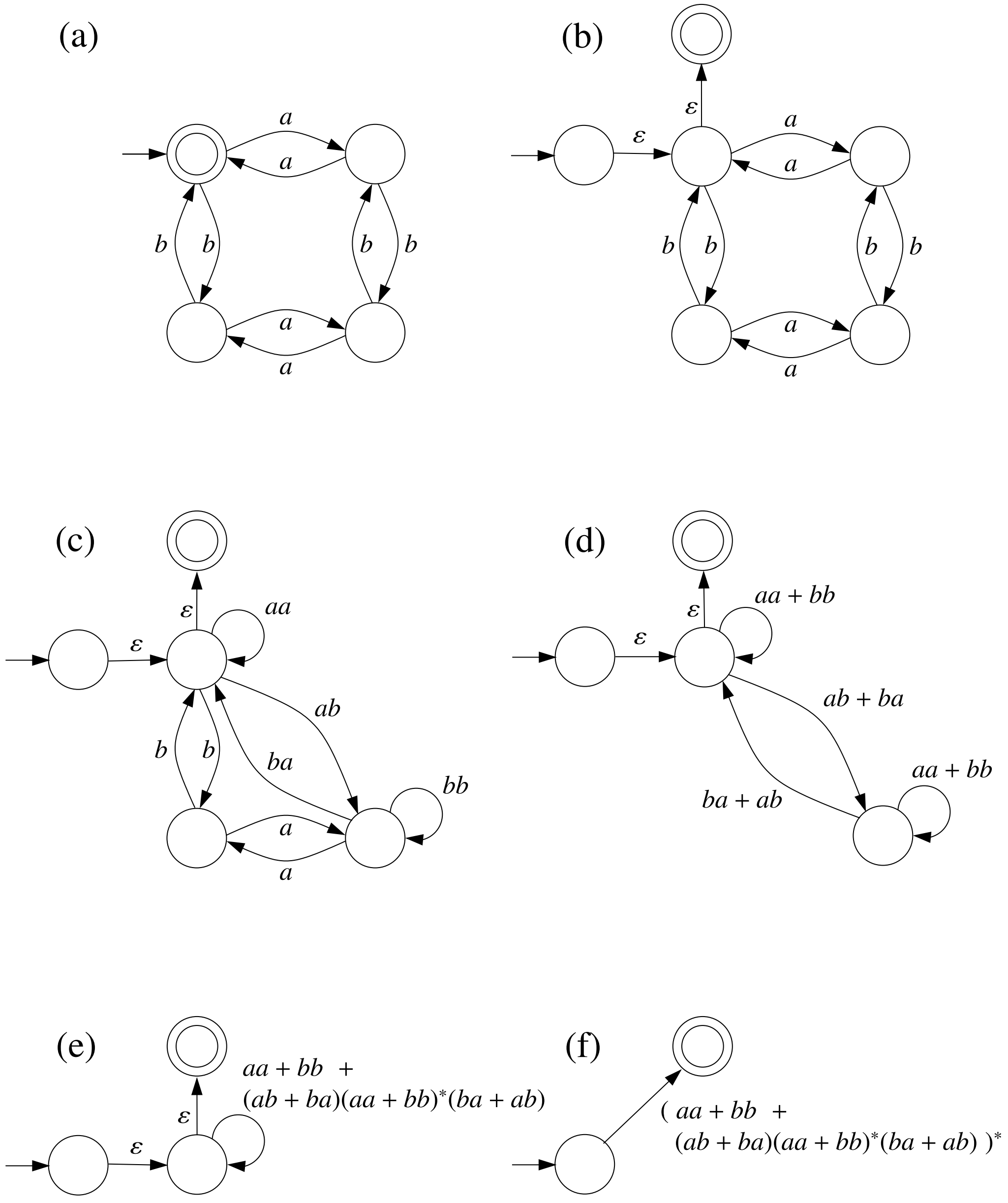


Figure 2.16: Run of *NFA-ϵtoRE*() on a DFA

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

$$\begin{aligned}
Q &= \{q_0, \ldots, q_{n-1}\} \\
\Sigma &= \{a_{ij} \mid 0 \leq i, j \leq n-1\} \\
Q_0 &= \{Q\}\delta \qquad\qquad\qquad\qquad = \{(q_i, a_{ij}, q_j) \mid 0 \leq i, j \leq n-1\} \\
F &= \{Q\}
\end{aligned}$$

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 by 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 or 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 2.16(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 2.17 on page 39 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 2.18 on page 40. Finally, we transform the NFA into a DFA by means of the subset construction. The result is shown in Figure 2.19 on page 40.

Observe that we do not go back to the DFA we started with, but to a different one recognizing the same language. A last step allowing us to close the circle is presented in the next chapter.

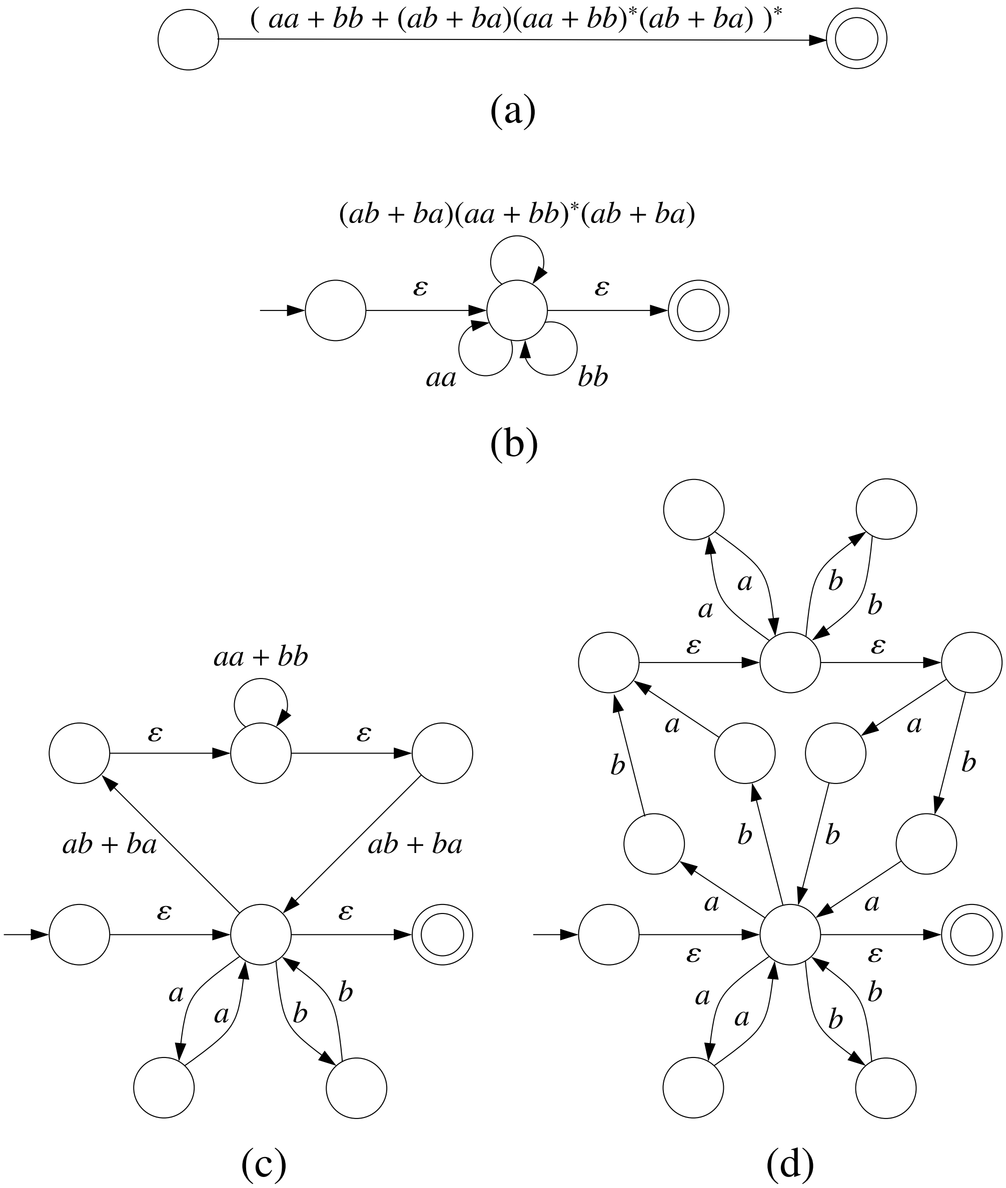

(aa + bb + (ab + ba)(aa + bb)*(ab + ba))*
(a)
(ab + ba)(aa + bb)*(ab + ba)
ε
ε
aa
bb
(b)
aa + bb
ε
ε
ab + ba
ab + ba
ε
ε
a
a
b
b
(c)
a
a
b
b
ε
ε
a
a
b
b
b
b
a
a
ε
ε
a
a
b
b
(d)

Figure 2.17: Constructing a NFA-ϵ for $(aa + bb + (ab + ba)(aa + bb)^*(ab + ba))^*$

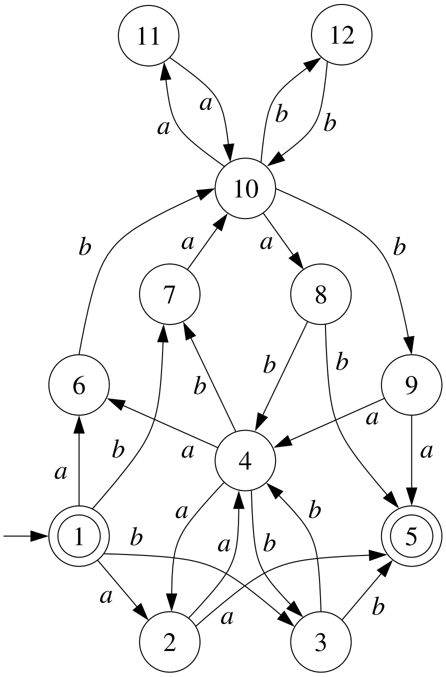


Figure 2.18: NFA for the NFA-ϵ of Figure 2.17(d)

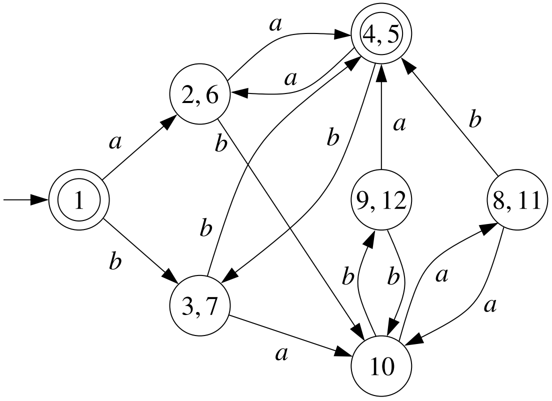


Figure 2.19: DFA for the NFA of Figure 2.18

Exercises

Exercise 1 Give a regular expression for the language of all words over $\Sigma = \{a, b\}$ . . .

(1) . . . beginning and ending with the same letter.

(2) . . . having two occurrences of a at distance 3.

(3) ...with no occurrences of the subword *aa*.

(4) ...containing exactly two occurrences of *aa*.

(5) ...that can be obtained from *abaab* by deleting letters.

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

Exercise 3 (Blondin) Prove that the language of the regular expression $r = (a + \varepsilon)(b^* + ba)^*$ is the set of all words over $\{a, b\}$ that do not contain any occurrence of aa.

Exercise 4 (Inspired by P. Rossmanith) Give syntactic characterizations of the regular expressions r satisfying (a) $L(r) = \emptyset$, (b) $L(r) = \{\varepsilon\}$, (c) $\varepsilon \in L(r)$, (d) the implication $\big(L(rr) = L(r)\big) \Rightarrow \big(L(r) = L(r^*)\big)$.

Exercise 5 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 $L(r \cap r') = L(r) \cap L(r')$. An extended regular expression is *star-free* if it does not contain any occurrence of the Kleene star operation. So, for example, $\overline{ab}$ and $(\overline{\emptyset}\, ab\, \overline{\emptyset}) \cap (\overline{\emptyset}\, ba\, \overline{\emptyset})$ are star-free, but ab^* is not.

A language $L \subseteq \Sigma^*$ is called *star-free* if there exists a star-free extended regular expression r such that $L = L(r)$. For example, Σ^* is star-free, because $\Sigma^* = L\big(\overline{\emptyset}\big)$. Show that the languages of the regular expressions (a) $(01)^*$ and (b) $(01 + 10)^*$ are star-free.

Exercise 6 Consider the language $L \subseteq \{a, b\}^*$ given by the regular expression $a^*b^*a^*a$.

1. Give an NFA-ε that accepts L.
2. Give an NFA that accepts L.
3. Give a DFA that accepts L.

Exercise 7 Let $|w|_\sigma$ denote the number of occurrences of a letter σ in a word w. For every $k \geq 2$, let $L_{k,\sigma} = \{w \in \{a, b\}^* \mid |w|_\sigma \bmod k = 0\}$.

1. Give a DFA with k states that accepts $L_{k,\sigma}$.
2. Show that any NFA accepting $L_{m,a} \cap L_{n,b}$ has at least $m \cdot n$ states.
   (Hint: consider using the pigeonhole principle.)

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. For example, if $L = \{abc, d\}$ then $L_{\text{pref}} = \{abc, ab, a, \varepsilon, d\}$ and $L_{\text{suf}} = \{abc, bc, c, \varepsilon, d\}$.

(1) Given an NFA A, construct NFAs A_{pref} and A_{suf} so that $L(A_{\text{pref}}) = L(A)_{\text{pref}}$ and $L(A_{\text{suf}}) = L(A)_{\text{suf}}$.

(2) Consider the regular expression $r = (ab + b)^*cd$. Give a regular expression r_{pref} so that $L\left(r_{\text{pref}}\right) = L\left(r\right)_{\text{pref}}$.

(3) More generally, give an algorithm that takes an arbitrary regular expression r as input, and returns a regular expression r_{pref} so that $L\left(r_{\text{pref}}\right) = L\left(r\right)_{\text{pref}}$.

Exercise 9 (Blondin) Consider the regular expression $r = (a + ab)^*$.

1. Convert r into an equivalent NFA-ε A.
2. Convert A into an equivalent NFA B.
3. Convert B into an equivalent DFA C.
4. By inspection of C, give an equivalent minimal DFA D.
5. Convert D into an equivalent regular expression r'.
6. Prove formally that $L\left(()\, r\right) = L\left(()\, r'\right)$.

Exercise 10 The *reverse* of a word w, denoted by w^R, is defined as follows: if $w = \varepsilon$, then $w^R = \varepsilon$, and if $w = a_1 a_2 \ldots a_n$ for $n \geq 1$, then $w^R = a_n a_{n-1} \ldots a_1$. The *reverse* of a language L is the language $L^R = \{w^R \mid w \in L\}$.

(1) Give a regular expression for the reverse of $((1 + 01)^*01(0 + 1))^*01$.

(2) Give an algorithm that takes as input a regular expression r and returns a regular expression r^R such that $L\left(r^R\right) = \left(L\left(r\right)\right)^R$.

(3) Give an algorithm that takes as input a NFA A and returns a NFA A^R such that $L\left(A^R\right) = \left(L\left(A\right)\right)^R$.

Exercise 11 Prove or disprove: Every regular language is recognized by a NFA …

(1) …having one single initial state.

(2) …having one single final state.

(3) …whose states are all initial.

(4) …whose states are all final.

(5) …whose initial states have no incoming transitions.

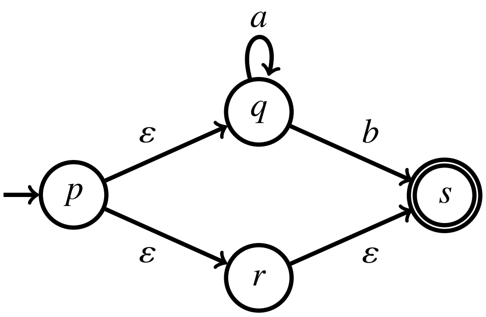
(6) ...whose final states have no outgoing transitions.

(7) ...such that all input transitions of a state (if any) carry the same label.

(8) ...such that all output transitions of a state (if any) carry the same label.

Which of the above hold for DFAs? Which ones for NFA-ϵ?

Exercise 12 Convert the following NFA-ε to an NFA using the algorithm *NFAεtoNFA* from the lecture notes (see Sect. 2.3, p. 31).



Exercise 13 Prove that every finite language (i.e., every language containing a finite number of words) is regular by defining a DFA that recognizes it.

Exercise 14 Let $\Sigma_n = \{1, 2, \ldots, n\}$, and let L_n be the set of all words $w \in \Sigma_n$ such that at least one letter of Σ_n does not appear in w. So, for instance, $1221, 32, 1111 \in L_3$ and $123, 2231 \notin L_3$.

(1) Give a NFA for L_n with $\mathcal{O}(n)$ states and transitions.

(2) Give a DFA for L_n with 2^n states.

(3) Show that any DFA for L_n has at least 2^n states.

(4) Which of (1) and (2) are still possible for $\overline{L_n}$, the set of words containing all letters of Σ_n? Does (3) still hold for $\overline{L_n}$?

Let $M_n \subseteq \{0,1\}^*$ be the set of words of length $2n$ of the form $(0+1)^{j-1}0(0+1)^{n-1}0(0+1)^{n-j}$ for some $1 \le j \le n$. These are the words containing at least one pair of 0s at distance n. For example, $101101, 001001, 000000 \in M_3$ and $101010, 000111, 011110 \notin M_3$.

(5) Give a NFA for M_n with $\mathcal{O}(n)$ states and transitions.

(6) Give a DFA for M_n with $\Omega(2^n)$ states.

(7) Show that any DFA for M_n has at least 2^n states.

Exercise 15 Let $L_n \subseteq \{a, b\}^*$ be the language described by the regular expression $(a + b)^* a(a + b)^n b(a + b)^*$.

1. Give an NFA with $n + 3$ states that accepts L_n.
2. Show that for every $w \in \{a, b\}^*$, if $|w| = n + 1$, then $ww \notin L_n$.
3. Show that any NFA accepting $\overline{L_n}$ has at least 2^{n+1} states. (Hint: use (b) and the pigeonhole principle.)

Exercise 16 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.

Intuitively, we can visualize an automaton with universal accepting condition as executing all runs in parallel. After reading a word w, the automaton is simultaneously in all states reached by all runs labelled by w, and accepts if all those states are accepting.

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

(1) Give an automaton of size $\mathcal{O}(n)$ with universal accepting condition that recognizes L_n.

(2) Prove that every NFA (and so in particular every DFA) recognizing L_n has at least 2^n states.

(3) Give an algorithm that transforms an automaton with universal accepting condition into a DFA recognizing the same language. This shows that automata with universal accepting condition recognize the regular languages.

Exercise 17 (1) Give a regular expression of size $O(n)$ such that the smallest DFA equivalent to it has $\Omega(2^n)$ states.

(2) Give a regular expression of size $O(n)$ *without "+"* such that the smallest DFA equivalent to it has $\Omega(2^n)$ states.

(3) Give a regular expression of size $O(n)$ *without + and of start-height 1* such that the smallest DFA equivalent to it has $\Omega(2^n)$ states. (Paper by Shallit at STACS 2008).

Exercise 18 Let K_n be the complete directed graph with nodes $\{1, \ldots, n\}$ and edges $\{(i, j) \mid 1 \leq i, j \leq n\}$. A path of K_n is a sequence of nodes, and a circuit of K_n is a path that begins and ends at the same node.

Consider the family of DFAs $A_n = (Q_n, \Sigma_n, \delta_n, q_{0n}, F_n)$ given by

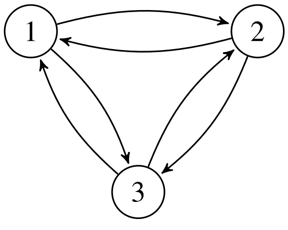
- $Q_n = \{1, \ldots, n, \bot\}$ and $\Sigma_n = \{a_{ij} \mid 1 \leq i, j \leq n\}$;

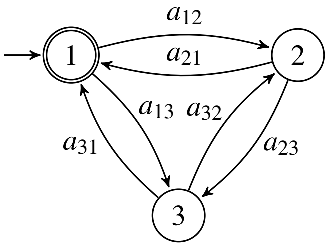
- $\delta_n(\bot, a_{ij}) = \bot$ for every $1 \leq i, j \leq n$ (that is, $\bot$ is a trap state), and

$$\delta_n(i, a_{jk}) = \begin{cases} \bot & \text{if } i \neq j \\ k & \text{if } i = j \end{cases}$$

- $q_{0n} = 1$ and $F_n = \{1\}$.

For example, here are K_3 and A_3:





Every word accepted by A_n encodes a circuit of K_n. For example, the words $a_{12}a_{21}$ and $a_{13}a_{32}a_{21}$, which are accepted by A_3, encode the circuits 121 and 1321 of K_3. Clearly, A_n recognizes the encodings of all circuits of K_n starting at node 1.

A *path expression* r over Σ_n is a regular expression such that every word of $L(r)$ models a path of K_n. The purpose of this exercise is to show that every path expression for $L(A_n)$—and so every regular expression, because any regular expression for $L(A_n)$ is a path expression by definition—must have length $\Omega(2^n)$.

- Let π be a circuit of K_n. A path expression r *covers* π if $L(r)$ contains a word uwv such that w encodes π. Further, r *covers* π^* if $L(r)$ covers π^k for every $k \geq 0$. Let r be a path expression of length m starting at a node i. Prove:

  (a) Either r covers π^*, or it does not cover π^{2m}.

  (b) If r covers π^* and no proper subexpression of r does, then $r = s^*$ for some expression s, and every word of $L(s)$ encodes a circuit starting at a node of π.

- For every $1 \leq k \leq n + 1$, let $[k]$ denote the permutation of $1, 2, \cdots, n + 1$ that cyclically shifts every index k positions to the right. Formally, node i is renamed to $i + k$ if $i + k \leq n + 1$, and to $i + k - (n + 1)$ otherwise. Let $\pi[k]$ be the result of applying the permutation to π. So, for instance, if $n = 4$ and $\pi = 24142$, we get

$$\pi[1] = 35253 \quad \pi[2] = 41314 \quad \pi[3] = 52425 \quad \pi[4] = 13531 \quad \pi[5] = 24142 = \pi$$

  (c) Prove that $\pi[k]$ is a circuit of K_{n+1} that does not pass through node k.

- Define inductively the circuit g_n of K_n for every $n \geq 1$ as follows:

- $g_1 = 11$
- $g_{n+1} = 1\, g_n[1]^{2^n}\, g_n[2]^{2^n} \cdots g_n[n+1]^{2^n}$ for every $n \geq 1$

In particular, we have

$$\begin{aligned} g_1 &= 11 \\ g_2 &= 1\,(22)^2\,(11)^2 \\ g_3 &= 1\,(2\,(33)^2\,(22)^2)^4\,(3\,(11)^2\,(33)^2\,3)^4\,(1\,(22)^2\,(11)^2)^4 \end{aligned}$$

(d) Prove using parts (a)-(c) that every path expression covering g_n has length at least 2^{n-1}.

Exercise 19 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 20 Let L be an arbitrary language over a 1-letter alphabet. Prove that L^* is regular.

Exercise 21 In algorithm *NFAεtoNFA* for the removal of ϵ-transitions, no transition that has been added to the workset, processed *and* removed from the workset is ever added to the workset again. However, transitions may be added to the workset more than once. Give a NFA-ε and a run of *NFAεtoNFA* on it in which this happens.

Exercise 22 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 23 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 present an algebraic reformulation of the algorithm. We represent a NFA as a system of *language equations* with as many variables as states, and solve the system by eliminating variables. A language equation over an alphabet Σ and a set V of variables is an equation of the form $r_1 = r_2$, where r_1 and r_2 are regular expressions over $\Sigma \cup V$. For instance, $X = aX + b$ is a language equation. A solution of a system of equations is a mapping that assigns to each variable X a regular expression over Σ, such that the languages of the left and right-hand-sides of each equation are equal. For instance, $a * b$ is a solution of $X = aX + b$ because $L(a^*b) = L(aa^*b + b)$, and $\emptyset$ and a^* are two different solutions of $X = aX$.

(1) Arden's Lemma states that given two languages $A, B \subseteq \Sigma^*$ with $\varepsilon \notin A$, the smallest language $X \subseteq \Sigma^*$ satisfying $X = AX + B$ is the language A^*B. Prove Arden's Lemma.

(2) Consider the following system of equations, where the variables X, Y represent languages (regular expressions) over the alphabet $\Sigma = \{a, b, c, d, e, f\}$:

$$\begin{array}{rcl} X & = & aX \ + \ bY \ + \ c \\ Y & = & dX \ + \ eY \ + \ f. \end{array}$$

This system has many solutions. For example, $X = Y = \Sigma^*$ is a solution. But there is again 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 X. If $X \notin F$, then the equation has the form

$$X = \sum_{(X,a,Y)\in\delta} aY$$

and if $X \in F$ then

$$X = \left(\sum_{(X,a,Y)\in\delta} aY \right) + \varepsilon$$

if $X \in F$.

(3) Consider the DFA of Figure 2.16(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

$$\begin{array}{rcl} X & = & aY \ + \ bZ \ + \ \varepsilon \\ Y & = & aX \ + \ bW \\ Z & = & bX \ + \ aW \\ W & = & bY \ + \ aZ \end{array}$$

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 2.16.

Exercise 24 (Inspired by R. Majumdar) Consider a deck of cards (with arbitrary many cards) in which black and red cards alternate, the top card is black, and the bottom card is red. The set of possible decks is then given by the regular expression $(BR)^*$. 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 (with 1 as top card) into two piles 12 and 3456, and the riffle yields 345162 (we start the riffle with the first pile). Give a regular expression over the alphabet $\{B, R\}$ describing the possible configurations of the decks after the riffle.
Hint: After the cut, the last card of the first pile can be black or red. In the first case the two piles belong to $(BR)^*B$ and $R(BR)^*$, and in the second case to $(BR)^*$ and $(BR)^*$. Let $Rif(r_1, r_2)$ be the language of all decks obtained by performing a riffle on decks taken from $L(r_1)$ and $L(r_2)$. We are looking for a regular expression for

$$Rif\Big((BR)^*B, R(BR)^*\Big) + Rif\Big((BR)^*, (BR)^*\Big).$$

Use Exercise 23 to set up a system of equations over the variables $X := Rif\Big((BR)^*B, R(BR)^*\Big)$ and $Y := Rif\Big((BR)^*, (BR)^*\Big)$, and solve it.

Exercise 25 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:

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

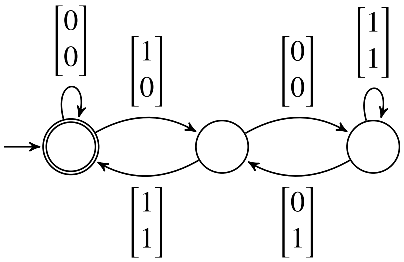
(a) Construct and compare DFAs recognizing the encodings of the even numbers $n \in \mathbb{N}_0$ w.r.t. the unary encoding, where n is encoded by the word 1^n, the MSBF-encoding, and the LSBF-encoding.

(b) Same for the set of numbers divisible by 3.

(c) Give regular expressions corresponding to the languages in (b).

Exercise 26 Consider the following DFA over the alphabet

$$\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}0\\1\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}, \text{ and } \begin{bmatrix}1\\1\end{bmatrix}.$$


$\begin{bmatrix}0\\0\end{bmatrix}$
$\begin{bmatrix}1\\0\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}$
$\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}0\\1\end{bmatrix}$

A word w encodes a pair of natural numbers $(X(w), Y(w))$, where $X(w)$ and $Y(w)$ are obtained by reading the top and bottom rows in MSBF encoding, respectively. For instance, in the word

$$\begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}$$

the top and bottom rows are 101100 and 010011, which in MSBF encoding correspond to 44 and 19

Show that the DFA recognizes the set of words w such that $X(w) = 3 \cdot Y(w)$, i.e., the solutions of the equation $x - 3y = 0$.

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 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.

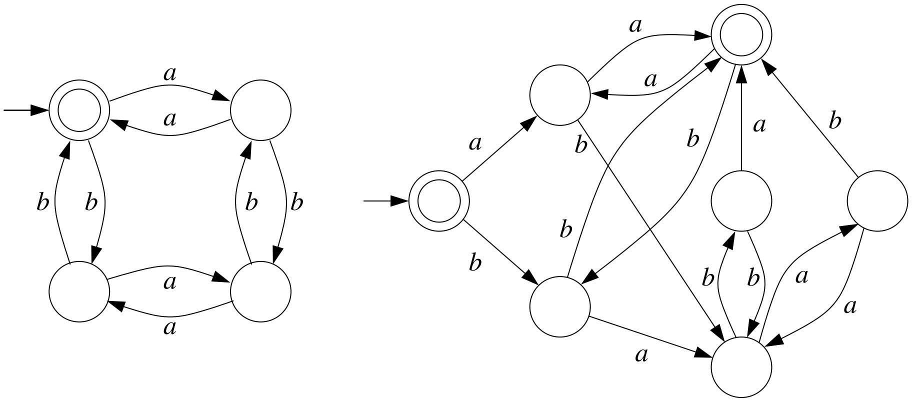


Figure 3.1: Two DFAs for the same language

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* residual of L with respect to w *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 $w \in \Sigma^*$.*

The language L^w satisfies the property

$$wu \in L \Leftrightarrow u \in L^w \tag{3.1}$$

Moreover, L^w is the only language satisfying this property. In other words, if a language L' satisfies $wu \in L \Leftrightarrow u \in L'$ for every word u, then necessarily $L' = L^w$.

Example 3.2 Let $\sigma = \{a, b\}$ and $L = \{a, ab, ba, aab\}$. We compute L^w for all words w by increasing length of w.

- $|w| = 0$. $L^\epsilon = \{a, ab, ba, aab\}$
- $|w| = 1$. $L^a = \{\epsilon, b, ab\}$, $L^b = \{a\}$.
- $|w| = 2$. $L^{aa} = \{b\}$, $L^{ab} = \{\epsilon\}$, $L^{ba} = \{\epsilon\}$, $L^{bb} = \emptyset$.
- $|w| \geq 3$. $L^w = \begin{cases} \epsilon & \text{if } w = aab \\ \emptyset & \text{otherwise} \end{cases}$.

Observe that residuals with respect to different words can be equal. In fact, even though σ^* contains infinitely many words , L has only six residuals, namely the languages $\emptyset$, $\{\epsilon\}$, $\{a\}$, $\{b\}$, $\{\epsilon, b, ab\}$, and $\{a, ab, ba, aab\}$. □

Example 3.3 Even languages containing infinitely many words can have a finite number of residuals. For example, $(a + b)^*$ contains infinitely many words, but it has one single residual: indeed, we have $L^w = (a + b)^*$ for every $w \in \{a, b\}^*$. Another example is the language of the two 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. Let us call this language EE in the following[1]. The language has four residuals,

[1] Notice that EE is a two-letter name for a language, not a concatenation of two languages!

namely the languages 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 $(EE)^{\epsilon} = EE$, $(EE)^{a} = OE$, and $(EE)^{ab} = OO$. □

Example 3.4 The languages of Example 3.2 and 3.3 have finitely many residuals, but this not the case for every language. In general, proving that the number of residuals of a language is finite or infinite can be complicated. To show that a language L has an infinite number of residuals one can use the following general proof strategy:

- Define an infinite set $W = \{w_0, w_1, w_2, \ldots\} \subseteq \Sigma^*$.
- Prove that $L^{w_i} \neq L^{w_j}$ holds for every $i \neq j$. For this, show that for every $i \neq j$ there exists a word $w_{i,j}$ that belongs to exactly one of the sets L^{w_i} and L^{w_j}.

We apply this strategy to two languages:

- Let $L = \{a^n b^n \mid n \geq 0\}$.
  Define $W := \{a^k \mid k \geq 0\}$. For every two distinct words $a^i, a^j \in W$ (i.e., $i \neq j$), we have $b^i \in L^{a^i}$, because $a^i b^i \in L$, but $b^i \notin L^{a^j}$, because $a^j b^i \notin L$. So L has infinitely many residuals.
- Let $L = \{ww \mid w \in \Sigma^*\}$.
  Define $W = \Sigma^*$. For every two distinct words $w, v \in W$ (i.e., $w \neq v$), we have $w \in L^w$, because $ww \in L$, but $w \notin L^v$, because $vw \notin L$. So L has infinitely many residuals.

□

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

Definition 3.5 *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 $A_q = (Q, \Sigma, \delta, q, F)$.*

For every transition $q \xrightarrow{a} q'$ of an automaton, deterministic or not, if a word w is accepted from q', then the word aw is accepted from q. For deterministic automata the converse also holds: since $q \xrightarrow{a} q'$ is the unique transition leaving q labeled by a, if aw is accepted from q, then w is accepted from q'. So we have $aw \in L(q')$ iff $w \in L(q)$ and comparing with Property 3.1 we obtain

$$\text{For every transition } q \xrightarrow{a} q' \text{ of a DA: } L(q') = L(q)^a. \tag{3.2}$$

More generally, we have:

Lemma 3.6 *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 at least one state $q \in Q$ 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 at least one word $w \in \Sigma^*$ 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, that is, $q_0 \xrightarrow{w} q$. We prove $L_A(q) = L^w$. By Property 3.1, it suffices to show that every word u satisfies

$$wu \in L \iff u \in L_A(q) \ .$$

Since A is a DFA, for every word $wu \in \Sigma^*$ the unique run of A on wu is of the form $q_0 \xrightarrow{w} q \xrightarrow{u} q'$. So A accepts wu iff q' is a final state, which is the case iff $u \in L_A(q)$. 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. The proof that $L_A(q) = L^w$ holds is exactly as above. □

Example 3.7 Figure 3.2 shows the result of labeling the states of the two DFAs of Figure 3.1 with the languages they recognize. All these languages are residuals of EE. □

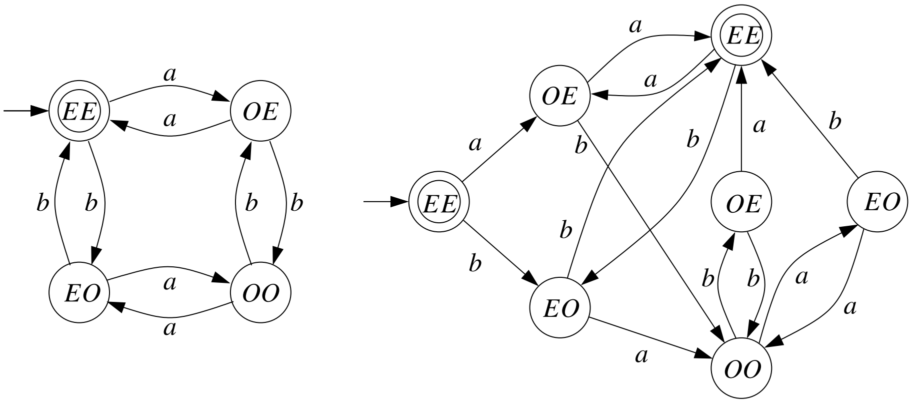


Figure 3.2: Languages recognized by the states of the DFAs of Figure 3.1.

We use the notion of a residual to define the *canonical deterministic automaton* for a given language L. We want this to be a DA whose states are languages, and in which “each state recognizes itself”, i.e., the language recognized from a state q is the language q itself. From this single requirement we can easily derive which should be the initial state, the transitions, and the final states:

- Since the canonical DA must recognize L, and each state must “recognize itself”, the initial state must be the language L itsef.

- By Property 3.2, since state K must recognize the language K, the transitions must be of the form $K \xrightarrow{a} K^a$.

- For the final states, observe that a state q of a deterministic automaton is final iff it recognizes the empty word. Therefore, a state K of the canonical DA is final iff $\epsilon \in K$.

We formalize this construction, and prove its correctness.

Definition 3.8 *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., $Q_L = \{L^w \mid w \in \Sigma^*\}$;*

- *$\delta_L(K, a) = K^a$ for every $K \in Q_L$ and $a \in \Sigma$;*

- *$q_{0L} = L$; and*

- *$F_L = \{K \in Q_L \mid \epsilon \in K\}$.*

Example 3.9 The canonical DA for the language EE is the one 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. The iniial state is EE. Since the empty word has an even number of a and b (zero in both cases), we have $\epsilon \in EE$, and $\epsilon \notin EO, OE, OO$. So the only final state is EE. □

Proposition 3.10 *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 $w = \epsilon$, and we have

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

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 has a minimal number of states. Further, every other minimal DFA for L also has exactly one state for each residual. It remains to show that all these minimal DFAs are isomorphic. For this we observe that, if we know which state recognizes which residual, we can infer which is the initial state, which are the transiitons, and which are the final states. In other words, the transitions, initial and final states of a minimal DFA are completely determined by the residual recognized by each state. Indeed, if state q recognizes residual R, then the a-transition leaving q necessarily leads to the state recognizing R^a; further, q is initial iff $R = L$, and q is final iff $\epsilon \in R$. A more formal proof looks as follows:

Theorem 3.11 *If L is regular, then C_L is the unique minimal DFA up to isomorphism that recognizes 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 3.6 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, and let $\mathcal{L}_A$ be the mapping that assigns to each state q of A the language $L(q)$ recognizied from q. By Lemma 3.6(2), $\mathcal{L}_A$ 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 between A and C_L. First, $\mathcal{L}_A$ is bijective because it is surjective (Lemma 3.6(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 $\epsilon \in L_A(q)$ iff $L_A(q) \in F_L$. □

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

Corollary 3.12 *A DFA is minimal if and only if different states recognize different languages, i.e., $L(q) \neq L(q')$ holds for every two states $q \neq q'$.*

Proof: (⇒): By Theorem 3.11, 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.

(⇐): 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). To compute P_ℓ we iteratively refine an initial partition P_0 while maintaining the following

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 $P_0 = \{Q \setminus F\} = \{Q\}$ 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 3.13 *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 3.14 *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 a partition can have at most $|Q|$ blocks.

```
LanPar(A)
Input: DFA $A = (Q, \Sigma, \delta, q_0, F)$
Output: The language partition $P_\ell$.
if $F = \emptyset$ or $Q \setminus F = \emptyset$ then return $\{Q\}$
else $P \leftarrow \{F, Q \setminus F\}$
while $P$ is unstable do
    pick $B, B' \in P$ and $a \in \Sigma$ such that $(a, B')$ splits $B$
    $P \leftarrow Ref_P[B, a, B']$
return $P$
```

Notice that if all states of a DFA are nonfinal then every state recognizes $\emptyset$, and if all are final then every state recognizes Σ^*. In both cases all states recognize the same language, and the language partition is $\{Q\}$.

Example 3.15 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 have 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 correctness of *LanPar* in two steps. 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 3.16 *LanPar(A) computes CSR.*

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 that this 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 3.17 *CSR is equal to* P_ℓ.

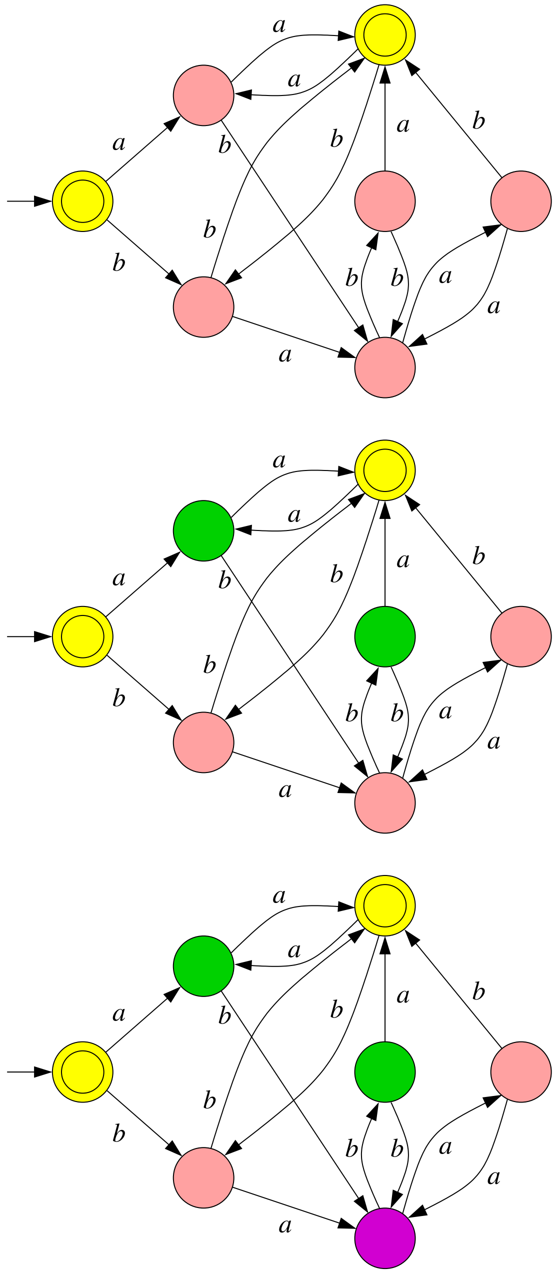


Figure 3.3: Computing the language partition for the DFA on the left of Figure 3.1

Proof: The proof has three parts:

(a) P_ℓ refines P_0. Obvious.

(b) P_ℓ is stable. By Fact 3.13, 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 $w = \epsilon$ 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 $w' \in L\left(q'_1\right)$. 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, $w' \in L\left(q'_1\right)$ iff $w' \in L\left(q'_2\right)$. So $w' \in L\left(q'_2\right)$, which implies $w \in L(q_2)$.

□

3.2.2 Quotienting

It remains to define the quotient of A with respect to a partition. It is convenient to define it not only for DFAs, but more generally for NFAs. 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 3.18 *The* quotient *of a NFA A with respect to a partition P is the NFA* $A/P = (Q_P, \Sigma, \delta_P, Q_{0P}, F_P)$ *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 set of blocks of P that contain at least one state of Q_0; and*
- *F_P is the set of blocks of P that contain at least one state of F.*

Example 3.19 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.

□

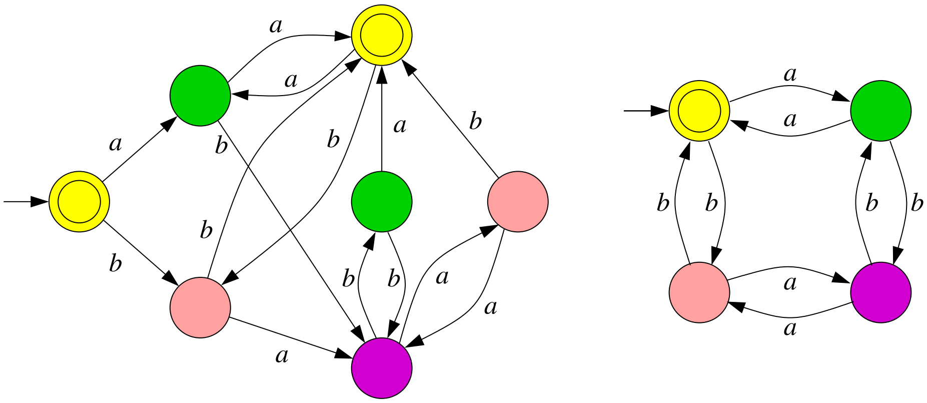


Figure 3.4: Quotient of a DFA with respect to its language partition

We show that A/P_ℓ, the quotient of a DFA 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 refinement of the language partition, i.e., 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. Moreover, if the partition not only refines but is equal to the language partition, then the quotient is a DFA.

Lemma 3.20 *Let A be a NFA, and let P be a partition of the states of A. If P refines P_ℓ, then $L_A(q) = L_{A/P}(B)$ for every state q of A, where B is the block of P containing q; in particular $L(A/P) = L(A)$. Moreover, if A is a DFA and $P = P_\ell$, then A/P is a DFA.*

Proof: Let P be any refinement of P_ℓ. 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 $w = \epsilon$ and we have

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

$|w| > 0$. Then $w = aw'$ for some $a \in \Sigma$. So $w \in L_A(q)$ iff 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:

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

For the second part, show that $(B, a, B_1), (B, a, B_2) \in \delta_{P_\ell}$ implies $B_1 = B_2$. By definition there exist $(q, a, q_1), (q', a, q_2) \in \delta$ for some $q, q' \in B$, $q_1 \in B_1$, and $q_2 \in B_2$. Since q, q' belong to the same block of the language partition, we have $L_A(q) = L_A(q')$. Since A is a DFA, we get $L_A(q_1) = L_A(q_2)$. Since $P = P_\ell$, the states q_1 and q_2 belong to the same block, and so $B_1 = B_2$. □

Proposition 3.21 *The quotient A/P_ℓ is the minimal DFA for L.*

Proof: By Lemma 3.20, A/P_ℓ is a DFA, and its states recognize residuals of L. Moreover, two states of A/P_ℓ recognize different residuals by definition of the language partition. So A/P_ℓ has as many states as residuals, and we are done. □

3.2.3 Hopcroft's algorithm

Algorithm *LanPar* leaves the choice of an adequate refinement triple $[B, a, B']$ open. While every exhaustive sequence of refinements leads to the same result, and so the choice does not affect the correctness of the algorithm, it affects its runtime. Hopcroft's algorithm is a modification of *LanPar* which carefully selects the next triple. When properly implemented, Hopcroft's algorithm runs in time $\mathcal{O}(mn \log n)$ for a DFA with n states over a m-letter alphabet. A full analysis of the algorithm is beyond the scope of this book, and so we limit ourselves to presenting its main ideas.

It is convenient to start by describing an intermediate algorithm, not as efficient as the final one. The intermediate algorithm maintains a workset of pairs (a, B'), called *splitters*. Initially, the workset contains all pairs (a, B') where a is an arbitrary letter and B' is a block of the original partition (that is, either $B' = F$ or $B' = Q \setminus F$). At every step, the algorithm chooses a splitter from the workset, and uses it to split every block of the current partition (if possible). Whenever a block B is split by (a, B') into two new blocks B_0 and B_1, the algorithm adds to the workset all pairs (b, B_0) and (b, B_1) for every letter $b \in \Sigma$.

It is not difficult to see that the intermediate algorithm is correct. The only point requiring a moment of thought is that it suffices to use each splitter at most once. *A priori* a splitter (a, B') could be required at some point of the execution, and then later again. To discard this observe that, by the definition of split, if (a, B') splits a block B into B_0 and B_1, then it does not split any subset of B_0 or B_1. So, after (a, B') is used to split all blocks of a partition, since all future blocks are strict subsets of the current blocks, (a, B') is not useful anymore.

Hopcroft's algorithm improves on the intermediate algorithm by observing that when a block B is split into B_0 and B_1, it is not always necessary to add both (b, B_0) and (b, B_1) to the workset. The fundamental for this is the following proposition:

Proposition 3.22 *Let $A = (Q, \Sigma, \delta, q_0, F)$, let P be a partition of Q, and let B be a block of P. Suppose we refine B into B_0 and B_1. Then, for every $a \in \Sigma$, refining all blocks of P with respect to any two of the splitters (a, B), (a, B_0), and (a, B_1) gives the same result as refining them with respect to all three of them.*

Proof: Let C be a block of P. Every refinement sequence with respect to two of the splitters (there are six possible cases) yields the same partition of C, namely $\{C_0, C_1, C_2\}$, where C_0 and C_1 contain the states $q \in Q$ such that $\delta(q, a) \in B_0$ and $\delta(q, a) \in B_1$, respectively, and C_2 contains the states $q \in Q$ such that $\delta(q, a) \notin B$. □

Now, assume that (a, B') splits a block B into B_0 and B_1. For every $b \in \Sigma$, if (b, B) is in the workset, then adding both (b, B_0) and (b, B_1) is redundant, because we only need two of the three. In this case, Hopcroft's algorithm chooses to replace (b, B) in the workset by (b, B_0) and (b, B_1) (that is, to remove (b, B) and to add (b, B_0) and (b, B_1)). If (b, B) is not in the workset, then in principle we could have two possible cases.

- If (b, B) was already removed from the workset and used to refine, then we only need to add one of (b, B_0) and (b, B_1). Hopcroft's algorithm adds the *smaller* of the two (i.e., (b, B_0) if $|B_0| \leq |B_1|$, and (b, B_1) otherwise).
- If (b, B) has not been added to the workset yet, then it looks as if we would still have to add both of (b, B_0) and (b, B_1). However, a more detailed analysis shows that this is not the case, it suffices again to add only one of (b, B_0) and (b, B_1), and Hopcroft's algorithm adds again the smaller of the two.

These considerations lead to the following pseudocode for Hopcroft's algorithm, where $(b, \min\{B_0, B_1\})$ denotes the smaller of (b, B_0) and (b, B_1):

```
*Hopcroft*($A$)
**Input:** DFA $A = (Q, \Sigma, \delta, q_0, F)$
**Output:** The language partition $P_\ell$.
 1  **if** $F = \emptyset$ or $Q \setminus F = \emptyset$ **then return** $\{Q\}$
 2  **else** $P \leftarrow \{F, Q \setminus F\}$
 3  $\mathcal{W} \leftarrow \{\ (a, \min\{F, Q \setminus F\})\ \mid a \in \Sigma\}$
 4  **while** $\mathcal{W} \neq \emptyset$ **do**
 5      pick $(a, B')$ from $\mathcal{W}$
 6      **for all** $B \in P$ split by $(a, B')$ **do**
 7          replace $B$ by $B_0$ and $B_1$ in $P$
 8          **for all** $b \in \Sigma$ **do**
 9              **if** $(b, B) \in \mathcal{W}$ **then** replace $(b, B)$ by $(b, B_0)$ and $(b, B_1)$ in $\mathcal{W}$
10              **else** add $(b, \min\{B_0, B_1\})$ to $\mathcal{W}$
11  **return** $P$
```

We sketch an argument showing that the **while** loop is executed at most $\mathcal{O}(mn \log n)$ times, where $m = |\Sigma|$ and $n = |Q|$. Fix a state $q \in Q$ and a letter $a \in \Sigma$. It is easy to see that at every moment during the execution of *Hopcroft* the workset contains at most one splitter (a, B) such that $q \in B$ (in particular, if (a, B) is in the workset and B is split at line 9, then q goes to either B_0 or to

B_1). We call this splitter (if present) the a-q-splitter, and define its size as the size of the block B. So during the execution of the algorithm there are alternating phases in which the workset contains one or zero a-q-splitters, respectively. Let us call them one-phases and zero-phases, respectively. It is easy to see that during a one-phase the size of the a-q-splitter (defined as the number of states in the block) can only decrease (at line 9). Moreover, if at the end of a one-phase the a-q-splitter has size k, then, because of line 10, at the beginning of the next one-phase it has size at most $k/2$. So the number of a-q-splitters added to the workset throughout the execution of the algorithm is $\mathcal{O}(\log n)$, and therefore the total number of splitters added to the workset is $\mathcal{O}(mn \log n)$. So the **while** loop is executed $\mathcal{O}(mn \log n)$ times. If the algorithm is carefully implemented (which is non-trivial), then it also runs in $\mathcal{O}(mn \log n)$ time.

3.3 Reducing NFAs

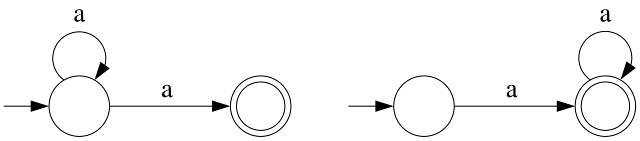


Figure 3.5: Two minimal NFAs for aa^*.

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. Moreover, computing any of the minimal NFAs equivalent to a given NFA is computationally hard. In Chapter 4 we will show that the *universality problem* for NFAs 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 equivalent to a given one is PSPACE-complete.

Theorem 3.23 *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. Recall that Definition 3.18 and the first part of Lemma 3.20 were defined for NFA. 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 3.16, 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 3.14 by

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

Definition 3.24 (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 3.16 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** $P \leftarrow \{Q\}$
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** $P$
```

Notice that line 1 of CSR(A) is different from line 1 in algorithm *LanPar*. If all states of a NFA are nonfinal then every state recognizes $\emptyset$, but if all are final we can no longer conclude that every state recognizes Σ^*, as was the case for DFAs. In fact, all states might recognize different languages.

In the case of DFAs we had Theorem 3.17, 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 3.25 *Let $A = (Q, \Sigma, \delta, Q_0, F)$ be a NFA. The partition CSR refines P_ℓ.*

Now, Lemma 3.20 and Theorem 3.25 lead to the final result:

Corollary 3.26 *Let $A = (Q, \Sigma, \delta, Q_0, F)$ be a NFA. Then $L(A/CSR) = L(A)$.*

Example 3.27 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 3.28 If A is an NFA, then A/P_ℓ might 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 $a \in \Sigma$ and for every $q_1' \in \delta(q_1, a)$, there exists $q_2' \in \delta(q_2, a)$ such that $L\left(q_1'\right) = L\left(q_2'\right)$. This can

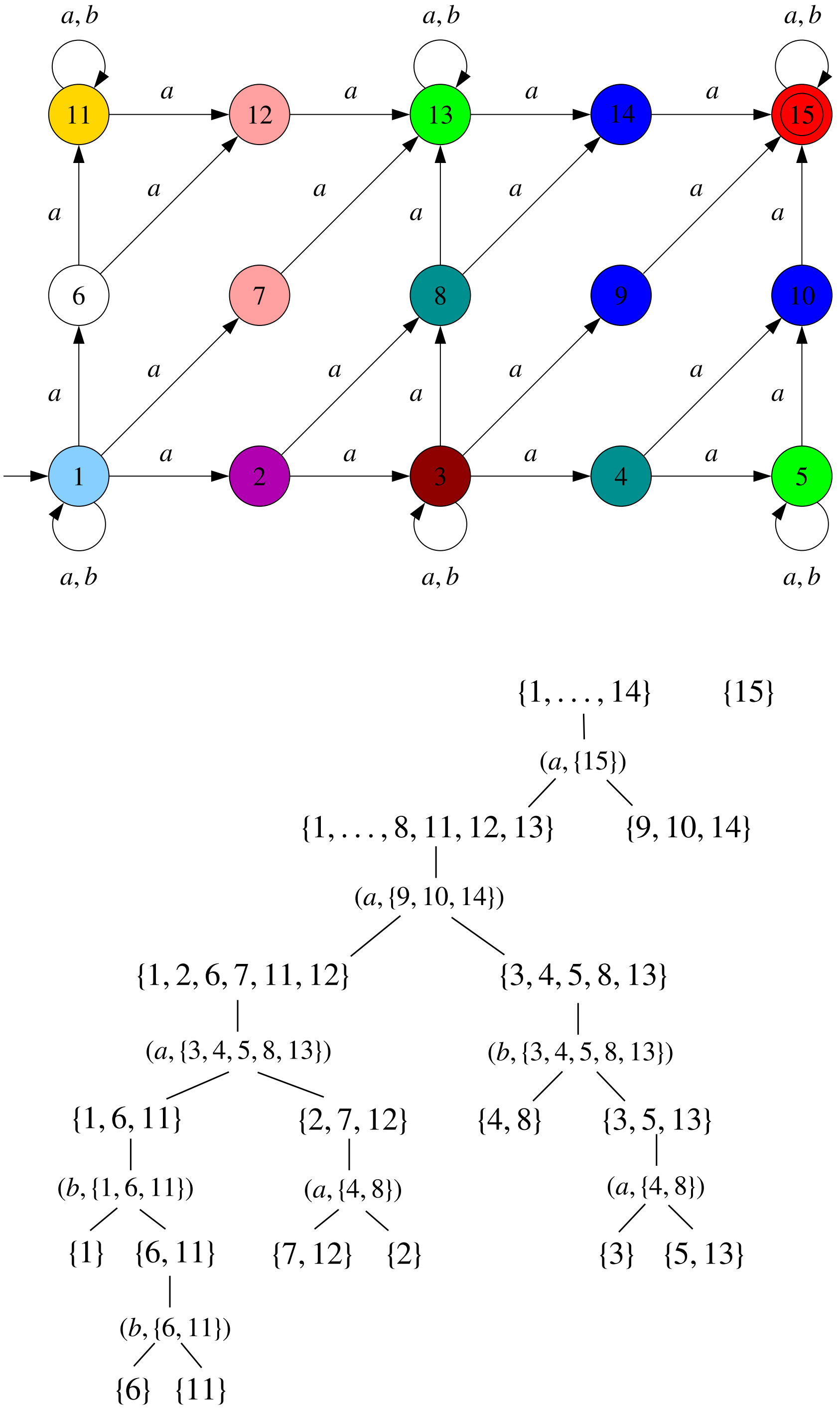


Figure 3.6: An NFA and a run of *CSR*() on it.

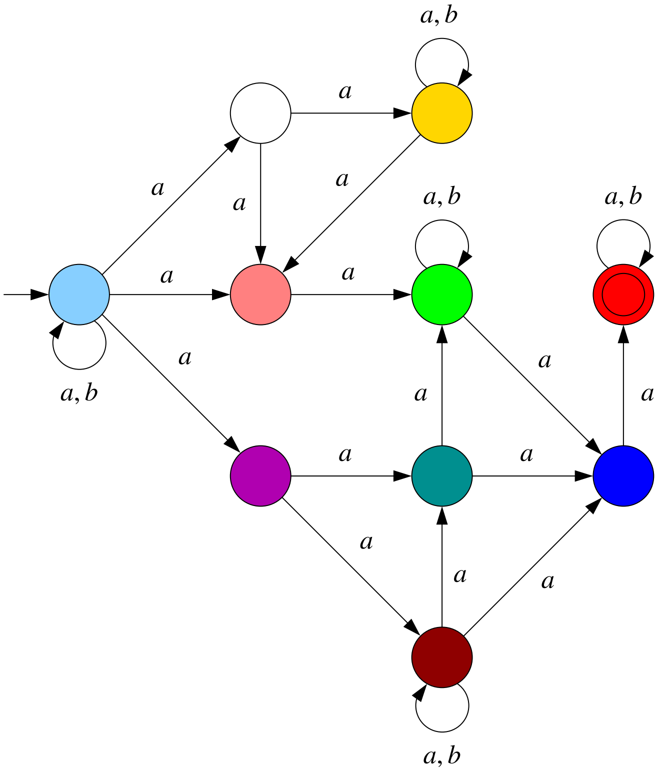


Figure 3.7: The quotient of the NFA of Figure 3.6.

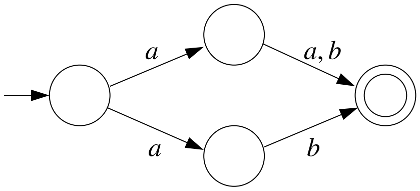


Figure 3.8: An NFA A such that A/P_ℓ is not minimal.

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. □

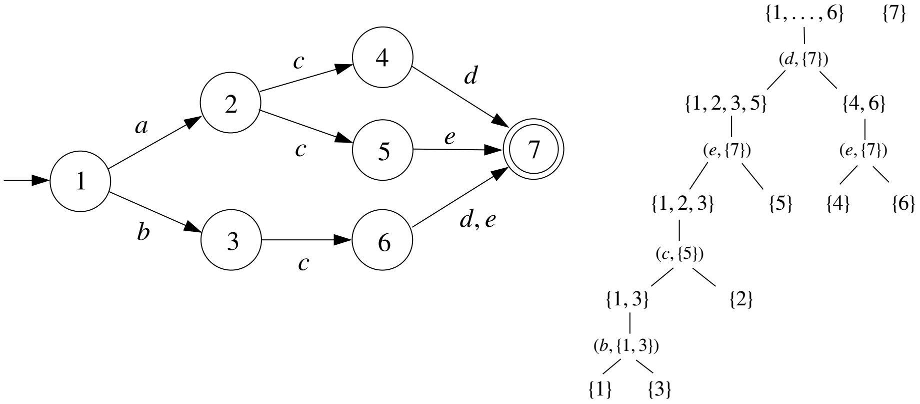


Figure 3.9: An NFA such that $CSR \neq P_\ell$.

3.4 A Characterization of the Regular Languages

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

Theorem 3.29 *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 3.10, 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 3.6, 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 $W \subseteq \Sigma^*$ such that $L^w \neq L^v$ for every two distinct words $w, v \in W$ (see also Example 3.4). In Example 3.4 we showed using this technique that the languages $\{a^n b^n \mid n \geq 0\}$ and $\{ww \mid w \in \Sigma^*\}$ have infinitely many residuals, and so that they are not regular. We give here a third example:

- $\{a^{n^2} \mid n \geq 0\}$. 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 27 Determine the residuals of the following languages over $\Sigma = \{a, b\}$: $(ab + ba)^*$, $(aa)^*$, and $\{a^n b^n c^n \mid n \geq 0\}$.

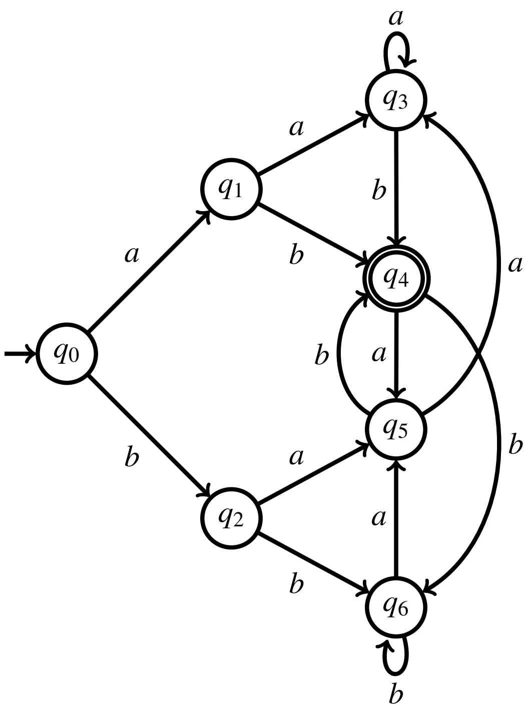
Exercise 28 Consider the most-significant-bit-first encoding (MSBF encoding) of natural numbers over the alphabet $\Sigma = \{0, 1\}$. In this exercise we assume that every number has infinitely many encodings, because all the words of $0 * w$ encode the same number as w.

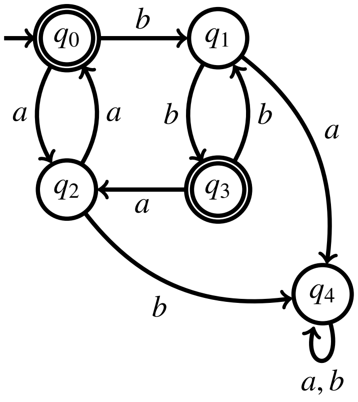
Construct the minimal DFAs accepting the following languages, where Σ^4 denotes all words of length 4.

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

(b) $\{w \mid \text{MSBF}^{-1}(w) \text{ is a prime }\} \cap \Sigma^4$.

Exercise 29 londin) Let A and B be respectively the following DFAs:





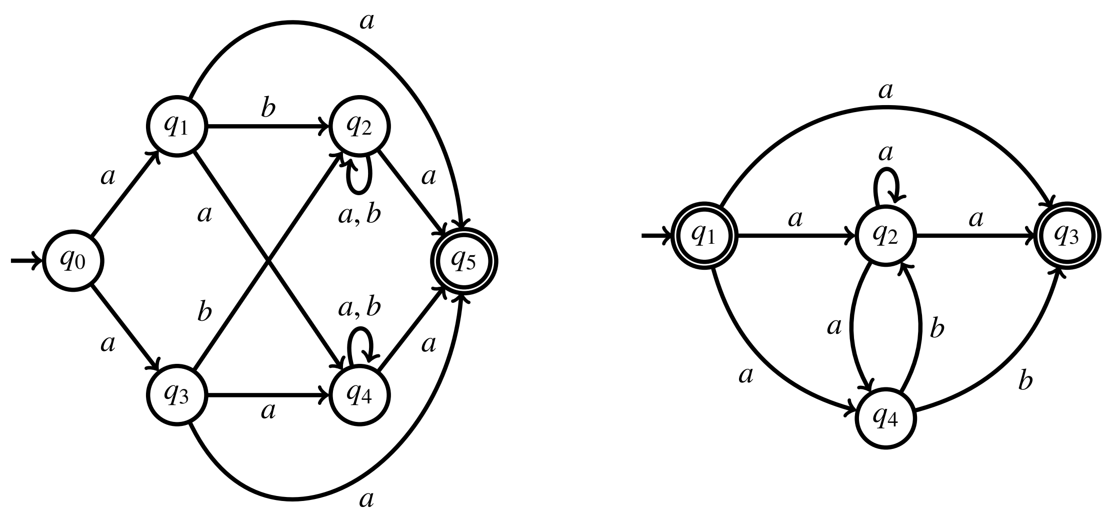
1. Compute the language partitions of A and B.

2. Construct the quotients of A and B with respect to their language partitions.

3. Give regular expressions for $L(A)$ and $L(B)$.

Exercise 30 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 31 (Blondin) Let A and B be respectively the following NFAs:



1. Compute the coarsest stable refinements (CSR) of A and B.

2. Construct the quotients of A and B with respect to their CSRs.

3. Are the obtained automata minimal?

Exercise 32 Let A_1, A_2 be two DFAs with n_1 and n_2 states, respectively. Show: if $L(A_1) \neq L(()A_2)$, then there exists a word w of length at most $n_1 + n_2 - 2$ such that $w \in (L_1 \setminus L_2) \cup (L_2 \setminus L_1)$. *Hint*: Consider the NFA obtained by putting A_1 and A_2 "side by side", and compute CSR(A). If $L(A_1) \neq L(A_2)$, then after termination the initial states q_{01} and q_{02} will be in different blocks.

Exercise 33 Consider the family of languages $L_k = \{ww \mid w \in \Sigma^k\}$, where $k \geq 2$.

(1) Construct the minimal DFA for L_2.

(2) How many states has the minimal DFA accepting L_k ?

Exercise 34 Given a language $L \subseteq \Sigma^*$ and $w \in \Sigma^*$, we denote ${}^{w}L = \{u \in \Sigma^* \mid uw \in L\}$. A language $L' \subseteq \Sigma^*$ is an *inverse residual* of L if $L' = {}^{w}L$ for some $w \in \Sigma^*$.

(1) Determine the inverse residuals of the first two languages in Exercise 27.

(2) Show that a language is regular iff it has finitely many inverse residuals.

(3) Does a language always have as many residuals as inverse residuals?

Exercise 35 A DFA $A = (Q, \Sigma, \delta, q_0, F)$ is *reversible* if no letter can enter a state from two distinct states, i.e. for every $p, q \in Q$ and $\sigma \in \Sigma$, if $\delta(p, \sigma) = \delta(q, \sigma)$, then $p = q$.

1. Give a reversible DFA that accepts $L = \{ab, ba, bb\}$.

2. Show that the minimal DFA accepting L is not reversible.

3. Is there a unique minimal reversible DFA accepting L (up to isomorphism)? Justify your answer.

Exercise 36 Given a regular expression r over Σ, let $E(r) = \varepsilon$ if $\varepsilon \in L(r)$, and $E(r) = \emptyset$ otherwise. For every $a \in \Sigma$, define the regular expression r^a inductively as follows:

- $\emptyset^a = \varepsilon^a = \emptyset$;

- $(r_1 + r_2)^a = r_1^a + r_2^a$;

- $(r_1 r_2)^a = r_1^a r_2 + E(r_1) r_2^a$;

- $(r^*)^a = r^a r^*$.

Prove $L(r^a) = (L(r))^a$.

Exercise 37 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 38 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 39 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 40 (T. Henzinger) Which of these languages over the alphabet $\{0, 1\}$ are regular?

(1) The set of words containing the same number of 0's and 1's.

(2) 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.)

(3) Same for the pair of strings 00 and 11, the pair 001 and 110, and the pair 001 and 100.

Exercise 41 (Blondin) Let Σ_1 and Σ_2 be alphabets. A *morphism* is a function $h : \Sigma_1^* \to \Sigma_2^*$ such that $h(\varepsilon) = \varepsilon$ and $h(uv) = h(u) \cdot h(v)$ for every $u, v \in \Sigma_1^*$. In particular, $h(a_1 a_2 \cdots a_n) = h(a_1)h(a_2) \cdots h(a_n)$ for every $a_1, a_2, \ldots, a_n \in \Sigma$. Hence, a morphism h is entirely determined by its image over letters.

1. Let A be an NFA over Σ_1. Give an NFA B that accepts $h(L(A)) = \{h(w) : w \in L(A)\}$.
2. Let A be an NFA over Σ_2. Give an NFA B that accepts $h^{-1}(L(A)) = \{w \in \Sigma_1^* : h(w) \in L(A)\}$.
3. Recall that $\{a^n b^n : n \in \mathbb{N}\}$ is not regular. Using this fact and the previous results, show that $L \subseteq \{a, b, c, d, e\}^*$ where
$$L = \{(ab)^m a^n e (cd)^m d^n : m, n \in \mathbb{N}\}$$
is also not regular.

Exercise 42 A word $w = a_1 \ldots a_n$ is a subword of $v = b_1 \ldots b_m$, denoted by $w \preceq v$, if there are indices $1 \leq i_1 < i_2 \ldots < i_n \leq m$ such that $a_{i_j} = b_j$ for every $j \in \{1, \ldots n\}$. 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*, resp. *downward-closed*), if for every two words $w, v \in \Sigma^*$, if $w \in L$ and $w \preceq v$, then $v \in L$, resp. if $w \in L$ and $w \succeq v$, 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 $v \in L$.

(1) Prove using Higman's lemma that every upward-closed language is regular.
Hint: Consider the minimal words of L, i.e., the words $w \in L$ such that no proper subword of w belongs to L.

(2) Prove that every downward-closed language is regular.

(3) Give regular expressions for the upward and downward closures of $\{a^n b^n \min n \geq 0\}$.

(4) Give algorithms that transform a regular expression r for a language into regular expressions $r\Uparrow$ and $r\Downarrow$ for its upward-closure and its downward-closure.

(5) Give algorithms that transform an NFA A recognizing a language into NFAs $A\Uparrow$ and $A\Downarrow$ recognizing its upward-closure and its downward-closure.

Exercise 43 (Abdulla, Bouajjani, and Jonsson) An *atomic expression* over an alphabet Σ^* is an expression of the form $\emptyset$, ε, $(a + \varepsilon)$ 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.

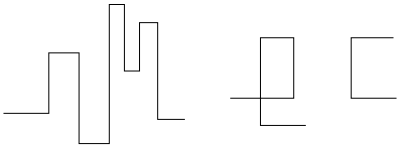
(1) Prove that the language of a simple regular expression is downward-closed (see Exercise 42).

(2) 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 44 A word of a language L is *minimal* if it is not a proper subword of another word of L. We denote the set of minimal words of L by $\min(L)$. Give a family of regular languages $\{L_n\}_{n=1}^{\infty}$ such that every L_n is recognized by a NFA with $O(n)$ states, but the smallest NFA recognizing $\min(L)$ has $O(2^n)$ states.

Exercise 45 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. For example, the set of all *staircases* can be specified as the set of lines given by the regular language $(up\, right)^*$. It is a regular language.

(a) Specify the set of all *skylines* as a regular language (i.e., formalize the intuitive notion of skyline). From the lines below, the one on the left is a skyline, while the other two are not.



(b) Show that the set of all *rectangles* is not regular.

Exercise 46 A NFA $A = (Q, \Sigma, \delta, Q_0, F)$ is em reverse-deterministic if $(q_1, a, q) \in \delta$ and $(q_2, a, q) \in$ *trans* implies $q_1 = q_2$, i.e., no state has two input transitions labelled by the same letter. Further, A is *trimmed* if every state accepts at least one word, i.e., if $L_A(q) \neq \emptyset$ for every $q \in Q$.

Let A be a reverse-deterministic, trimmed NFA with one single final state q_f. Prove that *NFAtoDFA*(A) is a minimal DFA.

Hint: Show that any two distinct states of *NFAtoDFA*(A) recognize different languages, and apply Corollary 3.12.

Exercise 47 Let *Rev*(A) be the algorithm of Exercise 10 that, given a NFA A as input, returns a trimmed NFA A^R such that $L(A^R) = (L(A))^R$, where L^R denotes the reverse of L (see Exercise 10). Recall that a NFA is trimmed if every state accepts at least one word (see Exercise 46).

Prove that for every NFA A the DFA

$$\mathit{NFAtoDFA}(\ \mathit{Rev}(\ \mathit{NFAtoDFA}(\ \mathit{Rev}(A)\ )\ )\ )$$

is the unique minimal DFA recognizing $L(A)$.

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 word w, and a state q, and a 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
if $w = \epsilon$ then return $q \in F$
else  return Member[A]( tail(w) , $\delta(q, head(w))$ )
```

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 $L(B) = \overline{L(A)}$
$Q' \leftarrow Q$; $\delta' \leftarrow \delta$; $q'_0 \leftarrow q_0$; $F' = \emptyset$
for all $q \in Q$ do
    if $q \notin F$ then add $q$ to $F'$
```

Observe that complementation of DFAs preserves minimality. By construction, each state of *CompDFA*(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 *CompDFA*(A). Apply now Corollary 3.12, 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:*

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

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 $w = a_1 a_2 \ldots a_n$ is also a "pairing" of the runs of A_1 and A_2 over w. Formally,

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

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

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

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 $[A_1, A_2]$. 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. □

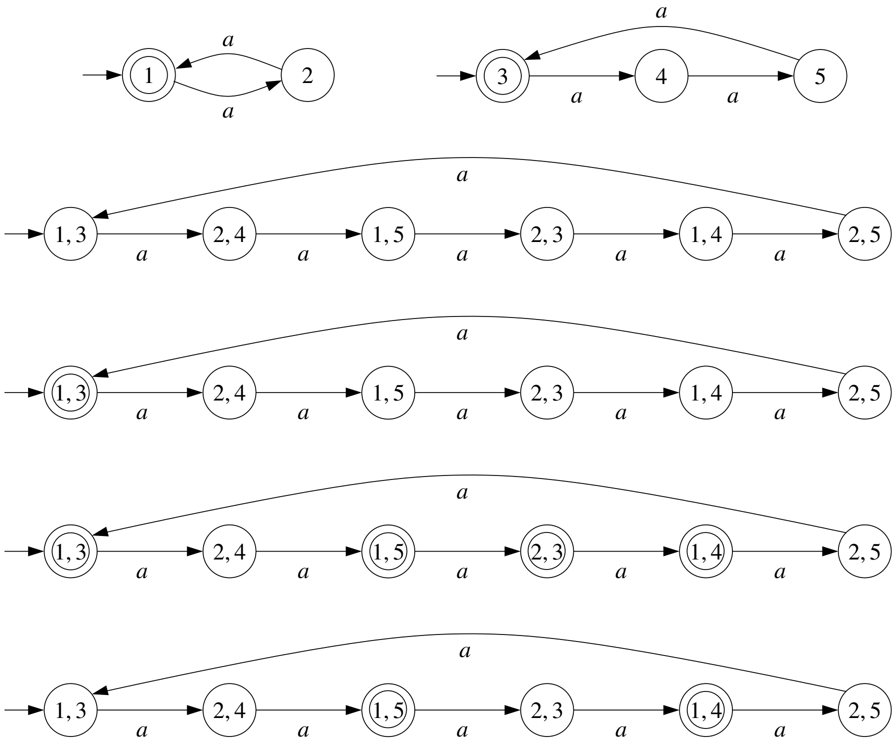


Figure 4.1: Two DFAs, their pairing, and DFAs for the intersection, union, and difference of their languages.

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

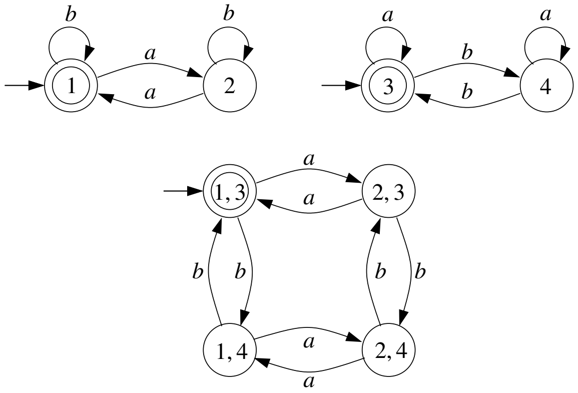


Figure 4.2: Two DFAs and a DFA for their intersection.

boolean operator $\odot$: $\{\textbf{true}, \textbf{false}\} \times \{\textbf{true}, \textbf{false}\} \rightarrow \{\textbf{true}, \textbf{false}\}$, we lift $\odot$ to a function $\widehat{\odot}$: $2^{\Sigma^*} \times 2^{\Sigma^*} \rightarrow 2^{\Sigma^*}$ 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 $L_1 \cap L_2 = L_1 \widehat{\wedge} L_2$. 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 $L(A) = L(A_1) \ \widehat{\odot} \ L(A_2)$

$Q, \delta, F \leftarrow \emptyset$
$q_0 \leftarrow [q_{01}, q_{02}]$
$W \leftarrow \{q_0\}$
while $W \neq \emptyset$ do
    pick $[q_1, q_2]$ from $W$
    add $[q_1, q_2]$ to $Q$
    if $(q_1 \in F_1) \odot (q_2 \in F_2)$ then add $[q_1, q_2]$ to $F$
    for all $a \in \Sigma$ do
        $q'_1 \leftarrow \delta_1(q_1, a)$; $q'_2 \leftarrow \delta_2(q_2, a)$
        if $[q'_1, q'_2] \notin Q$ then add $[q'_1, q'_2]$ to $W$
        add $([q_1, q_2], a, [q'_1, q'_2])$ to $\delta$
```

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 *Mult*(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 (i.e., two numbers without a common divisor), we have $\mathit{Mult}(n_1) \cap \mathit{Mult}(n_2) = \mathit{Mult}(n_1 \cdot n_2)$. Therefore, any DFA for $\mathit{Mult}(n_1 \cdot n_2)$ has at least $n_1 \cdot n_2$ states. In fact, if we denote the minimal DFA for $\mathit{Mult}(k)$ by A_k, then $\mathit{BinOp}[\wedge](A_n, A_m) = A_{n \cdot m}$.

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 $F = \emptyset$
```

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 $L_2 = L(A_2)$. We have

$$\begin{array}{ll} & L_1 \nsubseteq L_2 \\ \Leftrightarrow & L_1 \setminus L_2 \neq \emptyset \\ \Leftrightarrow & \text{at least one state } [q_1, q_2] \text{ of the DFA for } L_1 \setminus L_2 \text{ is final} \\ \Leftrightarrow & q_1 \in F_1 \text{ and } q_2 \notin F_2. \end{array}$$

□

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  $Q \leftarrow \emptyset$;
 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 ($q_2 \notin F_2$) then return false
 7      for all $a \in \Sigma$ do
 8          $q'_1 \leftarrow \delta_1(q_1, a)$; $q'_2 \leftarrow \delta_2(q_2, a)$
 9          if $[q'_1, q'_2] \notin Q$ then add $[q'_1, q'_2]$ 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** .

Let us call this algorithm *EqDFA*. An alternative procedure is to minimize A_1 and A_2, and then check if the results are isomorphic DFAs. In fact, the isomorphism check is not even necessary: one can just apply CSR to the NFA $A_1 \cup A_2 = (Q_1 \cup A_2, \Sigma, \delta_1 \cup \delta_2, \{q_{01}, q_{02}\}, F_1 \cup F_2$. It is easy to see that in this particular case CSR still computes the language partition, and so we have $L(()\,A_1) = L(()\,A_2)$ if and only if after termination the initial states of A_1 and A_2 belong to the same block.

If Hopcroft's algorithm is used for computing CSR, then the equality check can be performed in $O(n \log n)$ time, where n is the sum of the number of states of A_1 and A_2. This complexity is lower than that of *EqDFA*. However, *EqDFA* has two important advantages:

- It works on-the-fly. That is, $L(A_1) = L(A_2)$ can be tested while constructing A_1 and A_2. This allows to stop early if a difference in the languages is detected. On the contrary, minimization algorithms cannot minimize a DFA while constructing it. All states and transitions must be known before the algorithm can start.

- It is easy to modify *EqDFA* so that it returns a witness when $L(A_1) \neq L(A_2)$, that is, a word in the symmetric difference of $L(A_1)$ and $L(A_2)$. This is more difficult to achieve with the minimization algorithm. Moreover, to the best of our knowledge it cancels the complexity advantage. This may seem surprising, because, as shown in Exercise **??**, the shortest element in the symmetric difference of $L(A_1)$ and $L(A_2)$ has length n_1+n_2-2, where n_1 and n_2 are the numbers of states of A_1 and A_2, respectively. However, this element is computed by tracking for each pair of states the shortest word in the symmetric difference of the languages they recognize. Since there are $\mathcal{O}(n_1 \cdot n_2)$ pairs, this takes $\mathcal{O}(n_1 \cdot n_2)$ time. There could be a more efficient way to compute the witness, but we do not know of any.

4.2 Implementation on NFAs

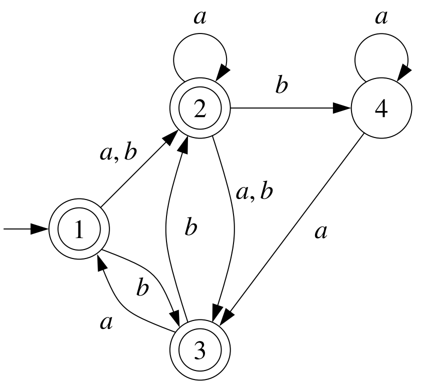
For NFAs we make the same assumptions on the complexity of basic operations as for DFAs. For DFAs, however, we had the assumption that, given a state q and a letter $a \in \Sigma$, we can find in constant time the unique state $\delta(q, a)$. This assumption no longer makes sense for NFA, since $\delta(q, a)$ 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
$W \leftarrow Q_0$;
while $w \neq \epsilon$ do
    $U \leftarrow \emptyset$
    for all $q \in W$ do
        add $\delta(q, head(w))$ to $U$
    $W \leftarrow U$
    $w \leftarrow tail(w)$
return $(W \cap F \neq \emptyset)$
```

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 $L\left(\overline{A}\right) = \overline{L(A)}$

$\overline{A} \leftarrow$ 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

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

are 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 $L(A_1 \cap A_2) = L(A_1) \cap L(A_2)$
$Q, \delta, F \leftarrow \emptyset$; $Q_0 \leftarrow Q_{01} \times Q_{02}$
$W \leftarrow Q_0$
while $W \neq \emptyset$ do
  pick $[q_1, q_2]$ from $W$
  add $[q_1, q_2]$ to $Q$
  if $(q_1 \in F_1)$ and $(q_2 \in F_2)$ then add $[q_1, q_2]$ to $F$
  for all $a \in \Sigma$ do
    for all $q_1' \in \delta_1(q_1, a), q_2' \in \delta_2(q_2, a)$ do
      if $[q_1', q_2'] \notin Q$ then add $[q_1', q_2']$ to $W$
      add $([q_1, q_2], a, [q_1', q_2'])$ to $\delta$
```

Notice that we overload the symbol ∩, 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, ∩ 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 $(q_1, a, q_1') \in \delta_1, (q_2, a, q_2') \in \delta_2$ of transitions, 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 67. 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, this is unnecessary. To obtain such a NFA, it suffices to put A_1 and A_2 "side by side": take the union of its states, transitions, initial, and final states (where we assume that these sets are disjoint):

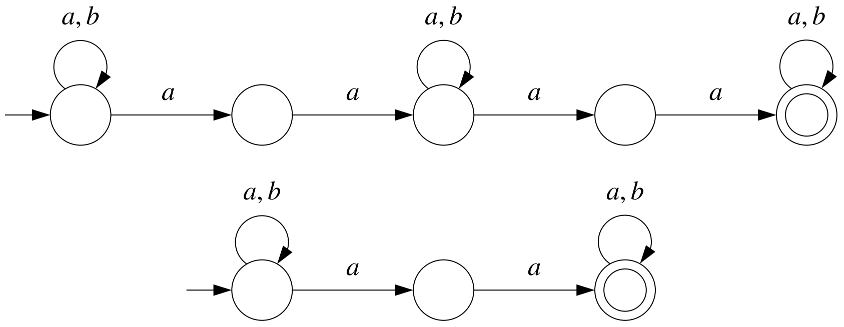


Figure 4.4: Two NFAs

```
$UnionNFA(A_1, A_2)$
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 $A_1 \cup A_2 = (Q, \Sigma, \delta, Q_0, F)$ with $L(A_1 \cup A_2) = L(A_1) \cup L(A_2)$

$Q \leftarrow Q_1 \cup Q_2$
$\delta \leftarrow \delta_1 \cup \delta_2$
$Q_0 \leftarrow Q_{01} \cup Q_{02}$
$F \leftarrow F_1 \cup F_2$
```

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 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 not 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 $B = \mathit{NFAtoDFA}(A)$ 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 $(\Sigma \cup \{\#\})^*$ (we assume $\# \notin \Sigma$) 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 $\mathit{NFAtoDFA}(A)$ 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 4.5 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. □

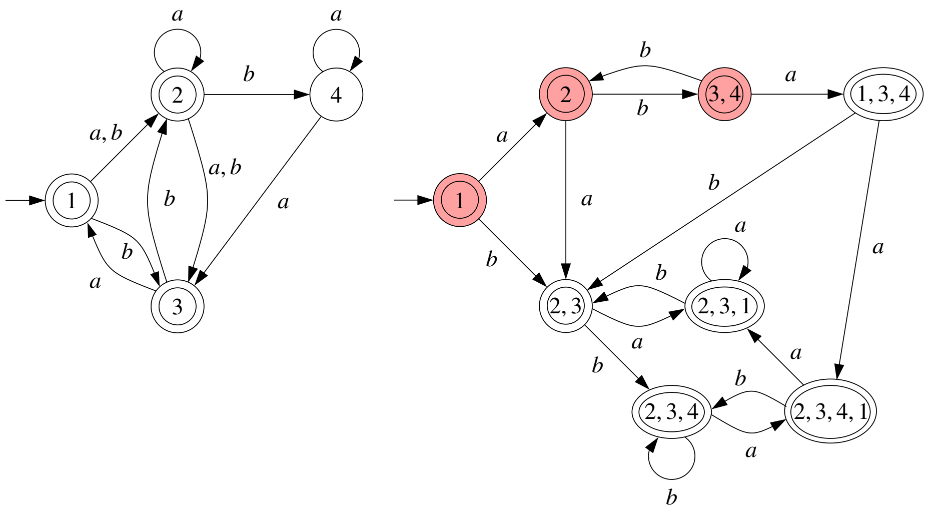


Figure 4.5: An NFA, and the result of converting it into a DFA, with the minimal states shaded.

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 workset.

```
*UnivNFA*($A$)
**Input:** NFA $A = (Q, \Sigma, \delta, Q_0, F)$
**Output:** **true** if $L(A) = \Sigma^*$, **false** otherwise
 1  $\mathcal{Q} \leftarrow \emptyset$;
 2  $\mathcal{W} \leftarrow \{ Q_0 \}$
 3  while $\mathcal{W} \neq \emptyset$ do
 4     pick $Q'$ from $\mathcal{W}$
 5     if $Q' \cap F = \emptyset$ then return false
 6     add $Q'$ to $\mathcal{Q}$
 7     for all $a \in \Sigma$ do
 8        $Q'' \leftarrow \bigcup_{q \in Q'} \delta(q, a)$
 9        if $\mathcal{W} \cup \mathcal{Q}$ contains no $Q''' \subseteq Q''$ then add $Q''$ to $\mathcal{W}$
10  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 $\pi = Q_1 \xrightarrow{a_1} Q_2 \ldots Q_{n-1} \xrightarrow{a_n} Q_n$ 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 $Q_2 \ldots Q_{n-1} \xrightarrow{a_n} Q_n$ is a shorter path satisfying the same properties), and so Q_2 is never added to the workset. On the other hand, since $Q_1 \in \mathcal{Q}_t$, the state Q_1 is eventually added to and picked from the workset. 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 workset in line 8. So at that moment either the workset 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 $\pi' = Q_2' \xrightarrow{a_2} Q_3' \ldots Q_{n-1}' \xrightarrow{a_n} Q_n'$ for some states $Q_3', \ldots, Q_n'$. Since $Q_2' \subset Q_2$ we have $Q_2' \subset Q_2, Q_3' \subseteq Q_3, \ldots, Q_n' \subseteq Q_n$ (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, the inclusion $L(A_1) \subseteq L(A_2)$ holds if and only if every state $[q_1, q_2]$ of $[A_1, A_2]$ having $q_1 \in F_1$ also has $q_2 \in F_2$. 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 $[A_1, B_2]$ having $q_1 \in F_1$ also has $Q_2 \cap F_2 \neq \emptyset$.*

Proof: Since A_2 and B_2 recognize the same language,

$$\begin{array}{ll} & L(A_1) \subseteq L(A_2) \\ \Leftrightarrow & L(A_1) \;\cap\; \overline{L(A_2)} \;=\; \emptyset \\ \Leftrightarrow & L(A_1) \cap \overline{L(B_2)} = \emptyset \\ \Leftrightarrow & [A_1, B_2] \text{ has a state } [q_1, Q_2] \text{ such that } q_1 \in F_1 \text{ and } Q_2 \cap F_2 = \emptyset \\ \Leftrightarrow & [A_1, B_2] \text{ has a \textit{minimal} state } [q_1, Q_2] \text{ such that } q_1 \in F_1 \text{ and } Q_2 \cap F_2 = \emptyset \end{array}$$

□

So we get the following algorithm to check inclusion:

```
InclNFA($A_1, A_2$)
Input: NFAs $A_1 = (Q_1, \Sigma, \delta_1, Q_{01}, F_1)$, $A_2 = (Q_2, \Sigma, \delta_2, Q_{02}, F_2)$
Output: true if $L(A_1) \subseteq L(A_2)$, false otherwise

$Q \leftarrow \emptyset$;
$W \leftarrow \{ [q_{01}, Q_{02}] \mid q_{01} \in Q_{01} \}$
while $W \neq \emptyset$ do
   pick $[q_1, Q_2']$ from $W$
   if ($q_1 \in F_1$) and ($Q_2' \cap F_2 = \emptyset$) then return false
   add $[q_1, Q_2']$ to $Q$
   for all $a \in \Sigma, q_1' \in \delta_1(q_1, a)$ do
      $Q_2'' \leftarrow \bigcup_{q_2 \in Q_2'} \delta_2(q_2, a)$
      if $\mathcal{W} \cup Q$ contains no $[q_1'', Q_2''']$ s.t. $q_1'' = q_1'$ and $Q_2''' \subseteq Q_2''$ then
         add $[q_1', Q_2'']$ to $\mathcal{W}$
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 workset 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 workset 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 48 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.

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 49 Prove or disprove: the minimal DFAs recognizing a language L and its complement $\overline{L}$ have the same number of states.

Exercise 50 Give a regular expression for the words over $\{0, 1\}$ that do not contain 010 as subword.

Exercise 51

- 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 52 Let $A = (Q, \Sigma, \delta, Q_0, F)$ be an NFA. Show that with the universal accepting condition of Exercise 16 the automaton $A' = (Q, \Sigma, \delta, q_0, Q \setminus F)$ recognizes the complement of the language recognized by A.

Exercise 53 Recall the alternating automata introduced in Exercise 19.

(a) 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 $\overline{A} = (Q_2, Q_1, \Sigma, \delta, q_0, Q \setminus F)$ 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.

(b) 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 *UnionNFA*.

(c) Show that the emptiness problem for alternating automata is PSPACE-complete.
Hint: Using Exercise 51, prove that the emptiness problem for alternating automata is PSPACE-complete.

Exercise 54 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 16.

Exercise 55 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 56 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 57 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 58 Given regular languages L_1, L_2 over an alphabet Σ, the *left quotient* of L_1 by L_2 is the language

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

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

1. Given NFA A_1, A_2, construct an NFA A such that $L(A) = L(A_1) \diagdown L(A_2)$.

2. Do the same for the *right quotient* of L_1 by L_2, defined as $L_1 \diagup L_2 := \{u \in \Sigma^* \mid \exists v \in L_2 : \; uv \in L_1\}$.

3. Determine the inclusion relations between the following languages: L_1, $(L_1 \diagup L_2)L_2$, and $(L_1 L_2) \diagup L_2$

Exercise 59 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.

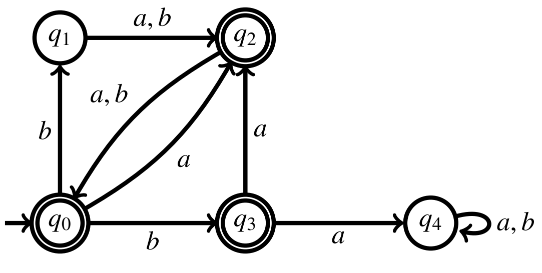
2. Construct the minimal DFA for L.

Exercise 60

- 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 61 Use the algorithm *UnivNFA* to test whether the following NFA is universal.



Exercise 62

1. Build B_p and C_p for the word pattern $p = mammamia$.

2. How many transitions are taken when reading $t = mami$ in B_p and C_p?

3. Let $n > 0$. Find a text $t \in \{a,b\}^*$ and a word pattern $p \in \{a,b\}^*$ such that testing whether p occurs in t takes n transitions in B_p and $2n-1$ transitions in C_p.

Exercise 63 Let Σ be an alphabet, and define the *shuffle operator* $\| \;:\; \Sigma^* \times \Sigma^* \to 2^{\Sigma^*}$ as follows, where $a, b \in \Sigma$ and $w, v \in \Sigma^*$:

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

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 64 The *perfect shuffle* of two languages $L, L' \in \Sigma^*$ is defined as:

$$L \,\widetilde{|||}\, L' = \{w \in \Sigma^* : \exists a_1, \ldots, a_n, b_1, \ldots, b_n \in \Sigma \text{ s.t. } \begin{array}{l} a_1 \cdots a_n \in L \text{ and} \\ b_1 \cdots b_n \in L' \text{ and} \\ w = a_1 b_1 \cdots a_n b_n \;\} \;. \end{array}$$

Give an algorithm that takes two DFAs A and B in input, and that returns a DFA accepting $L(A) \,\widetilde{|||}\, L(B)$.

Exercise 65 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 $h(w_1 w_2) = h(w_1)h(w_2) foreveryw_1, w_2 \in \Sigma_1^*$. 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 $\{0^n 1^n \mid n \in \mathbb{N}\}$ is not regular. Use the preceding results to show that $\{(01^k 2)^n 3^n \mid k, n \in \mathbb{N}\}$ is also not regular.

Exercise 66 Given alphabets Σ and Δ, a *substitution* is a map $f\colon \Sigma \to 2^{\Delta^*}$ 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 $\Sigma = \{\texttt{Name}, \texttt{Tel}, \texttt{:}, \texttt{\#}\}$, let $\Delta = \{A, \ldots, Z, 0, 1, \ldots, 9, :, \#\}$, and let f be the substitution f given by

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

1. Draw a DFA recognizing $L = \texttt{Name:Tel(\#Telephone)}^*$.

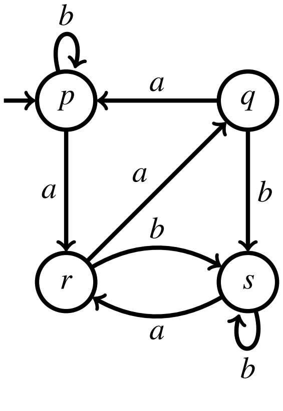
2. Sketch an NFA-reg recognizing $f(L)$.

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 67 Given two NFAs A_1 and A_2, let $B = \mathit{NFAtoDFA}(\mathit{IntersNFA}(A_1, A_2))$ and $C = \mathit{IntersDFA}(\mathit{NFAtoDFA}(A_1)$
Show that B and C are isomorphic, and so in particular have thesame number of states.
(A superficial analysis gives that for NFAs with n_1 and n_2 states B and C have $O(2^{n_1 \cdot n_2})$ and $O(2^{n_1+n_2})$ states, respectively, wrongly suggesting that C might be more compact than B.)

Exercise 68 (Blondin) Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA. A word $w \in \Sigma^*$ is a *synchronizing word* of A if reading w from any state of A leads to a common state, i.e. if there exists $q \in Q$ such that for every $p \in Q$, $p \xrightarrow{w} q$. A DFA is *synchronizing* if it has a synchronizing word.

1. Show that the following DFA is synchronizing:



2. Give a DFA that is not synchronizing.

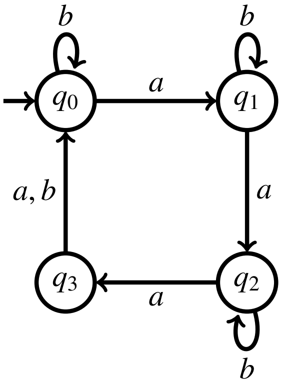
3. Give an exponential time algorithm to decide whether a DFA is synchronizing. (Hint: use the powerset construction).

4. Show that a DFA $A = (Q, \Sigma, \delta, q_0, F)$ is synchronizing if, and only if, for every $p, q \in Q$, there exist $w \in \Sigma^*$ and $r \in Q$ such that $p \xrightarrow{w} r$ and $q \xrightarrow{w} r$.

5. Give a polynomial time algorithm to test whether a DFA is synchronizing. (Hint: use 4).

6. Show that 4 implies that every synchronizing DFA with n states has a synchronizing word of length at most $(n^2 - 1)(n - 1)$. (Hint: you might need to reason in terms of the product construction.)

7. Show that the upper bound obtained in 6 is not tight by finding a synchronizing word of length $(4 - 1)^2$ for the following DFA:



Chapter 5

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 5.1 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.

5.1 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 2.14 on page 33, and then removing the ϵ transitions by means of *NFAϵtoNFA* on page 30. Let us call the resulting algorithm *RegtoNFA*. Once A is constructed, we simulate A on t as in *MemNFA*$[A](q_0,t)$ on page 78. 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 $t = a_1 \ldots a_n \in \Sigma^+$, pattern $p \in \Sigma^*$
Output: the first occurrence of $p$ in $t$, or $\perp$ if no such occurrence exists.
$A \leftarrow RegtoNFA(\Sigma^* p)$
$S \leftarrow Q_0$
for all $k = 0$ to $n - 1$ do
    if $S \cap F \neq \emptyset$ then return $k$
    $S \leftarrow \delta(S, a_{k+1})$
return $\perp$
```

For a given fixed alphabet Σ, a rough estimate of the complexity of *PatternMatchingNFA* for a word of length n and a pattern of length m can be obtained as follows. *RegtoNFA* is the concatenation of *RegtoNFAϵ* and *NFAϵtoNFA*. For a pattern of size m, *RegtoNFAϵ* takes $\mathcal{O}(m)$ time, and outputs a NFAϵ with $\mathcal{O}(m)$ states and $\mathcal{O}(m)$ transitions. So, when applied to this automaton, *NFAϵtoNFA* takes $\mathcal{O}(m^2)$ time, and outputs a NFA with $\mathcal{O}(m)$ states and $\mathcal{O}(m^2)$ transitions (see page 32 for the complexity of *NFAϵtoNFA*). The loop is executed at most n times, and, for an automaton with $\mathcal{O}(m)$ states, each line of the loop's body takes at most $\mathcal{O}(m^2)$ time. So the loop runs in $\mathcal{O}(nm^2)$ time. The overall runtime is thus $\mathcal{O}(nm^2)$.

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

```
PatternMatchingDFA($t$, $p$)
Input: text $t = a_1 \ldots a_n \in \Sigma^+$, pattern $p$
Output: the first occurrence of $p$ in $t$, or $\perp$ if no such occurrence exists.
$A \leftarrow NFAtoDFA(RegtoNFA(\Sigma^* p))$
$q \leftarrow q_0$
for all $k = 0$ to $n - 1$ do
    if $q \in F$ then return $k$
    $q \leftarrow \delta(q, a_{k+1})$
return $\perp$
```

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) runs in $\mathcal{O}(m^2)$ time, but it outputs a NFA with only $\mathcal{O}(m)$ states, and so the call to *NFAtoDFA* (see Table **??**) 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 $\mathcal{O}(n) + 2^{\mathcal{O}(m)}$.

5.2 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 $t = a_1 \ldots a_n$ and an arbitrary but fixed word pattern $p = b_1 \ldots b_m$. Instead of taking a fixed alphabet, we assume that the alphabet is implicitely defined by the text and the pattern, and so that the alphabet has size $\mathcal{O}(n + m)$.

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 a faster algorithm with time complexity $\mathcal{O}(m + n)$. Notice that in many applications n is very large, and so, even for a relatively small m the difference between nm and $m + n$ can be significant. Moreover, the constant hidden in the $\mathcal{O}$-notation is now *independent* of the size of the alphabet. This is important for applications where the alphabet is large (like chinese texts).

Figure 5.1(a) shows the obvious NFA A_p recognizing $\Sigma^* p$ for the case $p = nano$. In general, $A_p = (Q, \Sigma, \delta, \{q_0\}, F)$, where $Q = \{0, 1, \ldots, m\}$, $q_0 = 0$, $F = \{m\}$, and

$$\delta = \{(i, b_{i+1}, i + 1) \mid i \in [0, m - 1]\} \cup \{(0, a, 0) \mid a \in \Sigma\} .$$

Clearly, A_p can reach state k whenever the word read so far ends with $b_0 \ldots b_k$. We define the *hit* letter for each state of A_p. Intuitively, it is the letter that makes A_p "progress" towards reading p.

Definition 5.2 *We say that $a \in \Sigma$ is a* hit *for state i of A_p if $\delta(i, a) = \{i + 1\}$; otherwise a is a* miss *for state i.*

Figure 5.1(b) shows the DFA B_p obtained by applying *NFAtoDFA* to A_p. It has as many states as A_p, and there is a natural correspondence between the states of A_p and B_p: each state of A_p is the largest element of *exactly one* state of B_p. For instance, 3 is the largest element of $\{3, 1, 0\}$, and 4 is the largest element of $\{4, 0\}$.

Definition 5.3 *The* head *of a state $S \subseteq \{0, \ldots, m\}$ of B_p, denoted by $h(S)$, is the largest element of S. The* tail *of S, denoted by $t(S)$ is the set $t(S) = S \setminus \{h(S)\}$. The hit for a state S of B_p is defined as the hit of the state $h(S)$ in A_p.*

If we label a state with head k by the string $b_1 \ldots b_k$, as shown in Figure 5.1(c), then we see that B_p keeps track of how close it is to finding *nano*. For instance:

- if B_p is in state n and reads an a (a hit for this state), then it "makes progress", and moves to state na;
- if B_p is in state nan and reads an a (a miss for this state), then it is "thrown back" to state na. Not to ϵ, because if the next two letters are n and o, then B_p should accept!

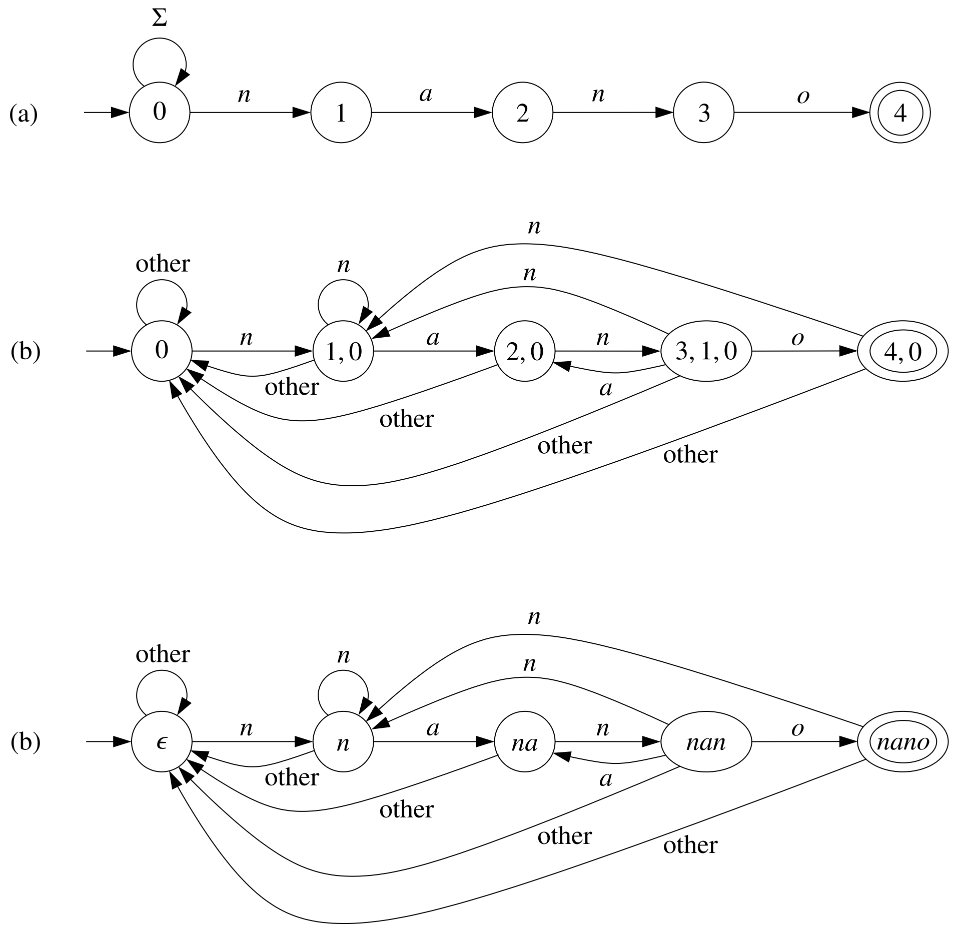


Figure 5.1: NFA A_p and DFA B_p for $p = nano$

B_p has another property that will be very important later on: for each state S of B_p (with the exception of $S = \{0\}$) the tail of S is again a state of B_p. For instance, the tail of $\{3, 1, 0\}$ is $\{1, 0\}$, which is also a state of B_p. We show that this property and the ones above hold in general, and not only in the special case $p = nano$. Formally, we prove the following invariant of *NFAtoDFA* (which is shown again in Table 5.2 for convenience) when applied to a word pattern p.

Proposition 5.4 *Let $p = b_1 \ldots b_m$, and let S_k be the k-th set picked from the workset during the execution of NFAtoDFA(A_p). Then*

(1) $h(S_k) = k$ (and therefore $k \leq m$), and

(2) either $k = 0$ and $t(S_k) = \emptyset$, or $k > 0$ and $t(S_k) \in \mathcal{Q}$.

```
   NFAtoDFA(A)
   Input: NFA $A = (Q, \Sigma, \delta, Q_0, F)$
   Output: DFA $B = (\mathcal{Q}, \Sigma, \Delta, \mathcal{Q}_0, \mathcal{F})$ with $L(B) = L(A)$
 1 $\mathcal{Q}, \Delta, \mathcal{F} \leftarrow \emptyset$; $\mathcal{Q}_0 \leftarrow \{q_0\}$
 2 $\mathcal{W} = \{\mathcal{Q}_0\}$
 3 while $\mathcal{W} \neq \emptyset$ do
 4   pick $S$ from $\mathcal{W}$
 5   add $S$ to $\mathcal{Q}$
 6   if $S \cap F \neq \emptyset$ then add $S$ to $\mathcal{F}$
 7   for all $a \in \Sigma$ do
 8     $S' \leftarrow \delta(S, a)$
 9     if $S' \notin \mathcal{Q}$ then add $S'$ to $\mathcal{W}$
10     add $(S, a, S')$ to $\Delta$
```

Table 5.1: *NFAtoDFA*(A)

Proof: We prove that (1) and (2) hold for all $0 \leq k \leq m-1$, and moreover that right before the k-th iteration of the while loop the workset only contains S_k (we call this (3)).

Clearly, (1), (2), and (3) hold for $k = 0$. For $k > 0$, consider the k-th iteration of the while loop. By induction hypothesis, we have $h(S_k) = k$ by (1), $t(S_k) = S_l$ for some $l < k$ by (2), and by (3) we know that before the k-th iteration the workset only contains S_k. So the algorithm picks S_k from the workset and examines the sets $\delta(S_k, a)$ for every action a, where we use the notation $\delta(S_k, a) = \bigcup_{q \in S_k} \delta(q, a)$.

We consider two cases: a is a hit for S_k, and a is a miss for S_k.

If a is a miss for S_k, then by definition it is also a miss for its head $h(S_k) = k$. So we have $\delta(k, a) = \emptyset$, and hence $\delta(S_k, a) = \delta(t(S_k), a) = \delta(S_l, a)$. So $\delta(S_k, a)$ was already explored by the algorithm during the l-th iteration of the loop, and $\delta(S_k, a)$ is not added to the workset at line 9.

If a is a hit for S_k, then $\delta(k, a) = \{k+1\}$. Since $\delta(S_k, a) = \delta(h(S_k), a) \cup \delta(t(S_k), a)$, we get $\delta(S_k, a) = \{k+1\} \cup \delta(S_l, a)$. Since state $k+1$ has not been explored before, the set $\{k+1\} \cup \delta(S_l, a)$ becomes the $(k+1)$-st state added to the workset, i.e., $S_{k+1} = \{k+1\} \cup \delta(S_l, a)$. So we get $h(S_{k+1}) = k+1$, and, since $t(S_{k+1}) = t(S_l)$, also $t(S_{k+1}) \in \mathcal{Q}$; in other words, S_{k+1} satisfies (1) and (2). Moreover, S_{k+1} is the only state added to the workset during the k-th iteration of the loop, and so we also have (3).

Finally, consider the m-th iteration of the loop. Since no letter is a hit for state m, the algorithm does not add any new state to the workset, and so at the end of the iteration the workset is empty and the algorithm terminates. □

By Proposition 5.4, B_p has at most $m+1$ states for a pattern of length m. So *NFAtoDFA* does not incur in any exponential blowup for word patterns. Even more: any DFA for $\Sigma^* p$ must have at least $m+1$ states, because for any two distinct prefixes p_1, p_2 of p the residuals $(\Sigma^* p)^{p_1}$ and $(\Sigma^* p)^{p_2}$ are also distinct. So we get

Corollary 5.5 *B_p is the minimal DFA recognizing $\Sigma^* p$.*

B_p has $m+1$ states and $m\,|\Sigma|$ transitions. Transitions of B_p labeled by letters that do not appear in p always lead to state 0, and so they do not need to be explicitely stored. The remaining $\mathcal{O}(m)$ transitions for each state can be constructed and stored using $\mathcal{O}(m^2)$ space and time, leading to a $\mathcal{O}(n+m^2)$ algorithm. To achieve $\mathcal{O}(n+m)$ time we introduce an even more compact data structure: the *lazy DFA for* $\Sigma^* p$, that, as we shall see, can be constructed in $\mathcal{O}(m)$ space and time.

5.2.1 Lazy DFAs

Recall that a DFA can be seen as the control unit of a machine that reads input from a tape divided into cells by means of a reading head At each step, the machine reads the content of the cell occupied by the reading head, updates the current state according to the transition function, and advances the head one cell to the right. The machine accepts a word if the state reached after reading it completely is final.

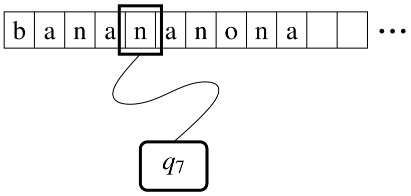


Figure 5.2: Tape with reading head.

In *lazy* DFAs the machine may advance the head one cell to the right *or keep it on the same cell*. Which of the two takes place is a function of the current state and the current letter read by the head. Formally, a lazy DFA only differs from a 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 a fourtuple (q, a, q', d), where $d \in \{R, N\}$ is the move of the head. Intuitively, a transition (q, a, q', N) means that state q is lazy and *delegates* processing the letter a to the state q'.

A lazy DFA C_p for $\Sigma^* p$. Recall that every state S_k of B_p has a hit letter and all other letters are misses. In particular, if a is a miss, then $\delta_B(S_k, a) = \delta(t(S_k), a)$, and so

> when it reads a miss, B_p moves from S_k to the same state it would move to, if it were in state $t(S)$.

Using this, we construct a lazy DFA C_p with the same states as B_p and transition function $\delta_C(S_k, a)$ given by:

- If a is a hit for S_k, then C_p behaves as B_p: $\delta_C(S_k, a) = (S_{k+1}, R)$.

- If a is a miss for S_k, then S_k "delegates" to $t(S_k)$: $\delta_C(S_k, a) = (t(S_k), N)$.
  However, if $k = 0$ then $t(S_k)$ is not a state, and C_p moves as B_p: $\delta_C(S_0, a) = (S_0, R)$.

Notice that in the case of a miss C_p always delegates to the same state, independently of the letter being read, and so we can "summarize" the transitions for all misses into a single transition $\delta_C(S_k, miss) = (t(S_k), N)$. Figure 5.3 shows the DFA and the lazy DFA for $p = nano$. (We write just k instead of S_k in the states.) Consider the behaviours of B_p and C_p from state S_3 if they read a n. B_p moves to S_1 (what it would do if it were in state S_1); instead, C_p delegates to S_1, which delegates to S_0, which moves S_1: that is, the move of B_p is simulated in C_p by a chain of delegations, followed by a move of the head to the right (in the worst case the chain of delegations reaches S_0, who cannot delegate to anybody). The final destination is the same in both cases.

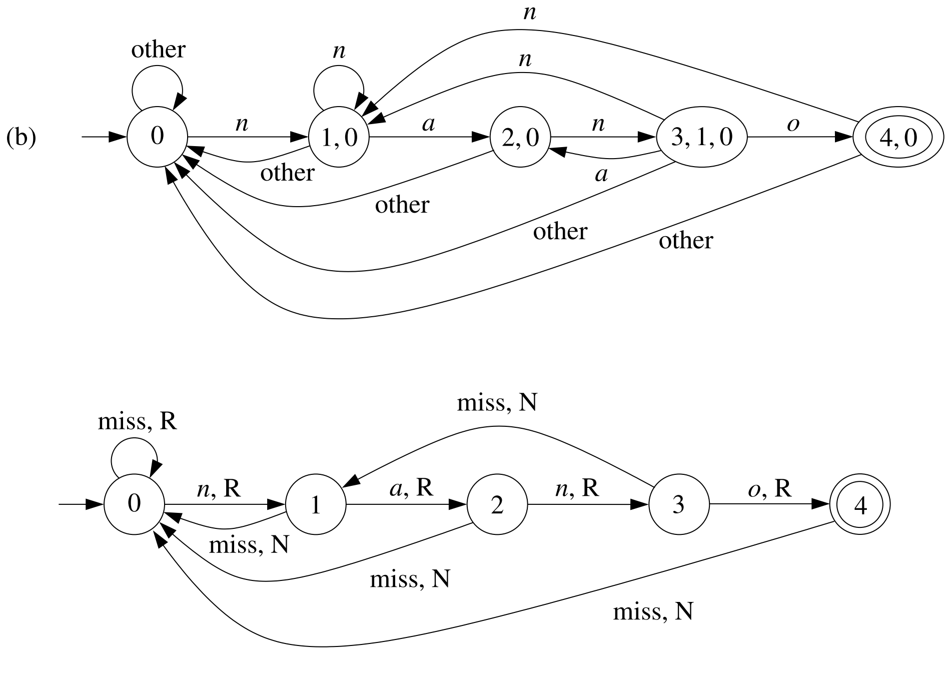


Figure 5.3: DFA and lazy DFA for $p = nano$

Notice that C_p may require more steps than B_p to read the text. But the number of steps is at most $2n$. For every letter the automaton C_p does a number of N-steps, followed by one R-step. Call this step sequence a *macrostep*, and let S_{j_i} be the state reached after the i-th macrostep, with $j_0 = 0$. Since the i-th macrostep leads from $S_{j_{i-1}}$ to S_{j_i}, and N-steps never move forward along the spine, the number of steps of the i-th macrostep is bounded by $j_{i-1} - j_i + 2$. So the total number of steps is bounded by

$$\sum_{i=1}^{n}(j_{i-1} - j_i + 2) = j_0 - j_n + 2n \leq 0 - m + 2n = 2n \; .$$

Computing C_p in $\mathcal{O}(m)$ time. Let $Miss(i)$ be the head of the target state of the miss transition for S_i. For instance, for $p = nano$ we have $Miss(3) = 1$ and $Miss(i) = 0$ for $i = 0, 1, 2, 4$. Clearly, if we can compute all of $Miss(0), \ldots, Miss(m)$ *together* in $\mathcal{O}(m)$ time, then we can construct C_p in $\mathcal{O}(m)$ time.

Consider the auxiliary function $miss(S_i)$ which returns the target state of the miss transition, instead of its head (i.e., $miss(S_i) = S_j$ iff $Miss(i) = j$). We obtain some equations for *miss*, which can then be easily transformed into an algorithm for *Miss*. By definition, for every $i > 0$ in the case of a miss the state S_i delegates to $t(S_i)$, i.e., $miss(S_i) = t(S_i)$. Since $t(S_1) = \{0\} = S_0$, this already gives $miss(S_1) = S_0$. For $i > 1$ we have $S_i = \{i\} \cup t(S_i)$, and so

$$t(S_i) = t(\delta_B(S_{i-1}, b_i)) = t(\delta(i-1, b_i) \cup \delta(t(S_{i-1}, b_i)) = t(\{i\} \cup \delta(t(S_{i-1}), b_i)) = \delta_B(t(S_{i-1}), b_i) \; .$$

Hence we get

$$miss(S_i) = \delta_B(miss(S_{i-1}), b_i)$$

Moreover, we have

$$\delta_B(S_j, b) = \begin{cases} S_{j+1} & \text{if } b = b_{j+1} \text{ (hit)} \\ S_0 & \text{if } b \neq b_{j+1} \text{ (miss) and } j = 0 \\ \delta_B(t(S_j), b) & \text{if } b \neq b_{j+1} \text{ (miss) and } j \neq 0 \end{cases}$$

Putting these two last equations together, and recalling that $miss(S_0) = S_0$, we finally obtain

$$miss(S_i) = \begin{cases} S_0 & \text{if } i = 0 \text{ or } i = 1 \\ \delta_B(miss(S_{i-1}), b_i) & \text{if } i > 1 \end{cases} \tag{5.1}$$

$$\delta_B(S_j, b) = \begin{cases} S_{j+1} & \text{if } b = b_{j+1} \text{ (hit)} \\ S_0 & \text{if } b \neq b_{j+1} \text{ (miss) and } j = 0 \\ \delta_B(miss(S_j), b) & \text{if } b \neq b_{j+1} \text{ (miss) and } j \neq 0 \end{cases} \tag{5.2}$$

Equations 5.1 and 5.2 lead to the algorithms shown in Table 5.2. $CompMiss(p)$ computes $Miss(i)$ for each index $i \in \{0, \ldots, m\}$. $CompMiss(p)$ calls $DeltaB(j, b)$, shown on the right of the Table, which returns the head of the state $\delta_B(S_j, b)$, i.e., $DeltaB(j, b)= k$ iff $\delta_B(S_j, b) = S_k$.
It remains to show that all calls to *DeltaB* during the execution of $Miss(p)$ take *together* $\mathcal{O}(m)$ time. Let n_i be the number of iterations of the while loop at line 1 during the call $DeltaB(Miss(i-1), b_i)$. We prove $\sum_{i=2}^{m} n_i \leq m$. To this end, observe that $n_i \leq Miss(i-1) - (Miss(i) - 1)$, because initially j is set to $Miss(i-1)$, each iteration of the loop decreases j by at least 1 (line 2), and the return value assigned to $Miss(i)$ is at most the final value of j plus 1. So we get

$$\sum_{i=2}^{m} n_i \leq \sum_{i=2}^{m} (Miss(i-1) - Miss(i) + 1) \leq Miss(1) - Miss(m) + m \leq Miss(1) + m = m$$

```
$CompMiss(p)$
Input: pattern $p = b_1 \cdots b_m$.
Output: heads of targets of miss transitions.
$Miss(0) \leftarrow 0$; $Miss(1) \leftarrow 0$
for $i \leftarrow 2, \ldots, m$ do
    $Miss(i) \leftarrow DeltaB(\, Miss(i-1)\, ,\, b_i\, )$
```

```
$DeltaB(j, b)$
Input: head $j \in \{0, \ldots, m\}$, letter $b$.
Output: head of the state $\delta_B(S_j, b)$.
while $b \neq b_{j+1}$ and $j \neq 0$ do  $j \leftarrow CompMiss(j)$
if $b = b_{j+1}$ then return $j + 1$
else return $0$
```

Table 5.2: Algorithm *CopmpMiss*(p)

Exercises

Exercise 69 Design an algorithm that solves the following problem for an alphabet Σ: Given a text $t \in \Sigma^*$ and a regular expression p over Σ, find a shortest prefix $w_1 \in \Sigma^*$ of t such that some prefix $w_1 w_2$ of w satisfies $w_2 \in L(p)$. Discuss the complexity of your solution.

Exercise 70 (Blondin)

1. Build the automata B_p and C_p for the word pattern $p = mammamia$.

2. How many transitions are taken when reading $t = mami$ in B_p and C_p?

3. Let $n > 0$. Find a text $t \in \{a, b\}^*$ and a word pattern $p \in \{a, b\}^*$ such that testing whether p occurs in t takes n transitions in B_p and $2n - 1$ transitions in C_p.

Exercise 71 We have shown that lazy DFAs for a word pattern may need more than n steps to read a text of length n, but not more than $2n + m$, where m is the length of the pattern. Find a text t and a word pattern p such that the run of B_p on t takes at most n steps and the run of C_p takes at least $2n - 1$ steps.
Hint: a simple pattern of the form a^k for some $k \geq 0$ is sufficient.

Exercise 72 (Blondin) Two-way DFAs are an extension of lazy automata where the reading head is also allowed to move left. Formally, a *two-way DFA (2DFA)* is a tuple $A = (Q, \Sigma, \delta, q_0, F)$ where $\delta : Q \times (\Sigma \cup \{\vdash, \dashv\}) \to Q \times \{L, S, R\}$. Given a word $w \in \Sigma^*$, A starts in q_0 with its reading tape initialized with $\vdash w \dashv$, and its reading head pointing on $\vdash$. When reading a letter, A moves the head according to δ (*L*eft, *S*tationnary, *R*ight). Moving left on $\vdash$ or right on $\dashv$ does not move the reading head. A accepts w if, and only if, it reaches $\dashv$ in a state of F.

1. Let $n \in \mathbb{N}$. Give a 2DFA that accepts $(a + b)^* a (a + b)^n$.

2. Give a 2DFA that does not terminate on any input.

3. Describe an algorithm to test whether a given 2DFA A accepts a given word w.

4. Let $A_1, A_2, \ldots, A_n$ be DFAs over a common alphabet. Give a 2DFA B such that

$$L(B) = L(A_1) \cap L(A_2) \cap \cdots \cap L(A_n) .$$

Chapter 6

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 $Id_X = \{(x, x) \mid x \in X\}$.

6.1 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 $x \in U$ 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 s_x 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 6.1 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 6.2 *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*

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

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.

6.2 Transducers and Regular Relations

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

Definition 6.3 *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 6.4 *Let T be a transducer over Σ. Given words $w_1 = a_1a_2 \dots a_n$ and $w_2 = b_1b_2 \dots b_n$, 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 6.2

Definition 6.5 *Let T be a transducer.*

- *T accepts a pair $(x, y) \in U \times U$ if it accepts all encodings of (x, y).*
- *T rejects a pair $(x, y) \in U \times U$ if it accepts no encoding of (x, y).*
- *T recognizes a relation $R \subseteq U \times U$ 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 6.6 The *Collatz function* is the function $f \colon \mathbb{N} \to \mathbb{N}$ 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 6.1 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 $LSBF(22) = L(011010^*)$. □

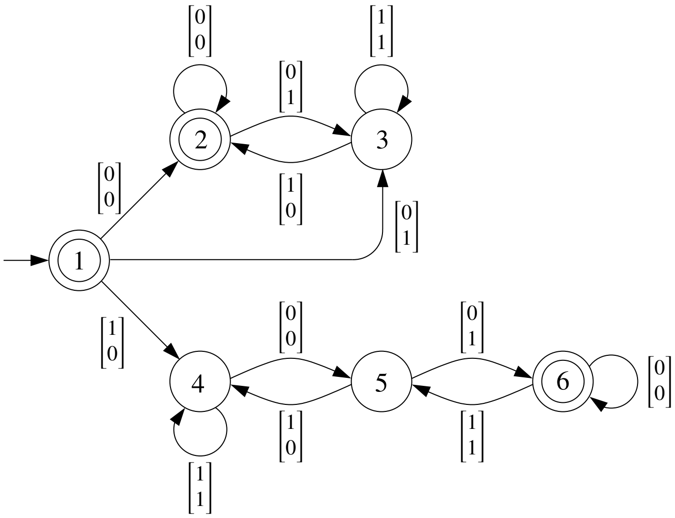

1
2
3
4
5
6
$\begin{bmatrix}0\\0\end{bmatrix}$
$\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}0\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}$
$\begin{bmatrix}1\\0\end{bmatrix}$
$\begin{bmatrix}0\\1\end{bmatrix}$
$\begin{bmatrix}1\\0\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}$
$\begin{bmatrix}0\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}$
$\begin{bmatrix}1\\0\end{bmatrix}$
$\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}1\\1\end{bmatrix}$

Figure 6.1: A transducer for Collatz's function.

Why "transducer"? In Engineering, a transducer is a device that converts signals in one form of energy into signals in a different form. Two examples of transducers are microphones and loudspeakers. We can look at a transducer T over an alphabet Σ as a device that transforms an input word into an output word. If we choose Σ as the union of an input and an output alphabet, and ensure that in every transition $q \xrightarrow{(a,b)} q'$ the letters a and b are an input and an output letter, respectively, then the transducer transforms a word over the input alphabet into a word over the output alphabet. (Observe that the same word can be transformed into different ones.)

When looking at transducers from this point of view, it is customary to write a pair $(a, b) \in \Sigma \times \Sigma$ as a/b, and read it as "the transducer reads an a and writes a b". In some exercises we use this notation. However, in section 6.4 we extend the definition of a transducer, and consider transducers that recognize relations of arbitrary arity. For such transducers, the metaphore of a converter is less appealing: while in a binary relation it is natural and canonical to interpret the first and second components of a pair as "input" and "output", there is no such canonical interpretation for a relation of arity 3 or higher. In particular, there is no canonical extension of the a/b notation. For this reason, while we keep the name "transducer" for historical reasons, we use the notation $q \xrightarrow{(a_1,\dots,a_n)} q'$ for transitions, or the column notation, as in Example 6.6.

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 6.1 is deterministic in this sense, when an appropriate sink state is added.

There is another possibility to define determinism of transducers, which corresponds to the converter interpretation $(a, b) \mapsto a/b$ described in the previous paragraph. If the letter a/b is interpreted as "the transducer receives the input a and produces the output b", then it is natural to call a transducer 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.

6.3 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 none 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.

6.3.1 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** $\delta'$
5  $F' \leftarrow PadClosure((Q', \Sigma, \delta', Q'_0, F''), \#)$
```

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  $W \leftarrow F$; $F' \leftarrow \emptyset$;
2  while $W \neq \emptyset$ do
3     pick $q$ from $W$
4     add $q$ to $F'$
5     for all $(q', \#, q) \in \delta$ do
6        if $q' \notin F'$ 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 6.7 Figure 6.2 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

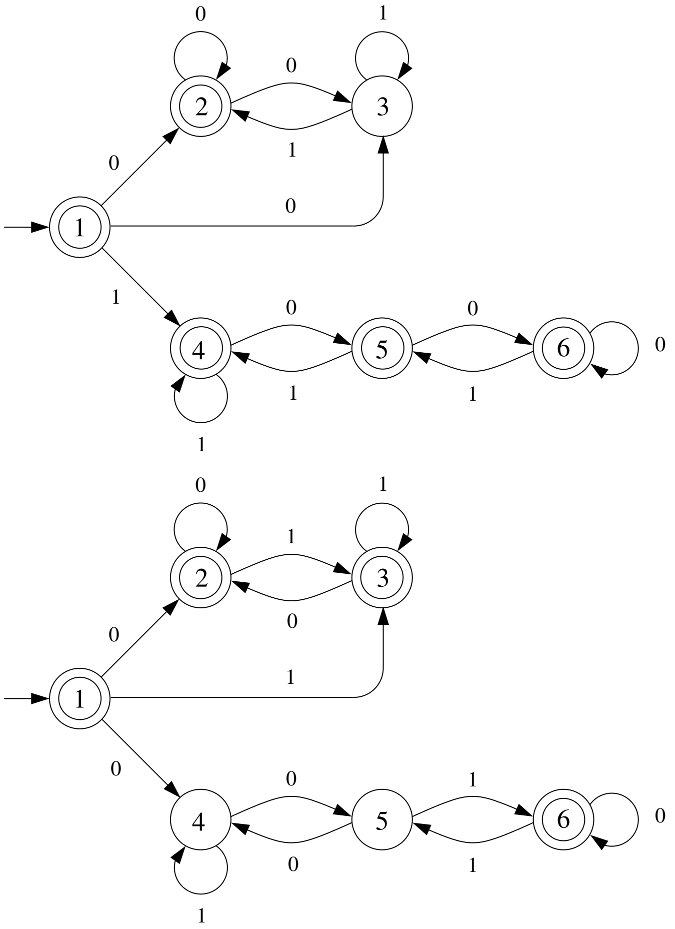


Figure 6.2: Projections of the transducer for the Collatz function onto the first and second components.

can easily check that this is indeed the case. Observe that both projections are nondeterministic, although the transducer is deterministic. □

6.3.2 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 $q \xrightarrow{(a,c)} r$ 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

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

iff T_1 and T_2 have runs

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

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  $Q, \delta, F' \leftarrow \emptyset$;  $Q_0 \leftarrow Q_{01} \times Q_{02}$
 2  $W \leftarrow Q_0$
 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 $\delta$
 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 6.8 Recall that the transducer of Figure 6.1, shown again at the top of Figure 6.3, 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 6.3 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 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 $(q_1, c, q'_1) \in \delta_1, (q_2, (c, b), q'_2) \in \delta_2$ do
8        add $([q_1, q_2], b, [q'_1, q'_2])$ to $\delta$
```

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 $(q_1, (a, c), q'_1) \in \delta_1, (q_2, c, q'_2) \in \delta_2$ do
8        add $\delta$ to $([q_1, q_2], a, [q'_1, q'_2])$
```

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 6.9 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:

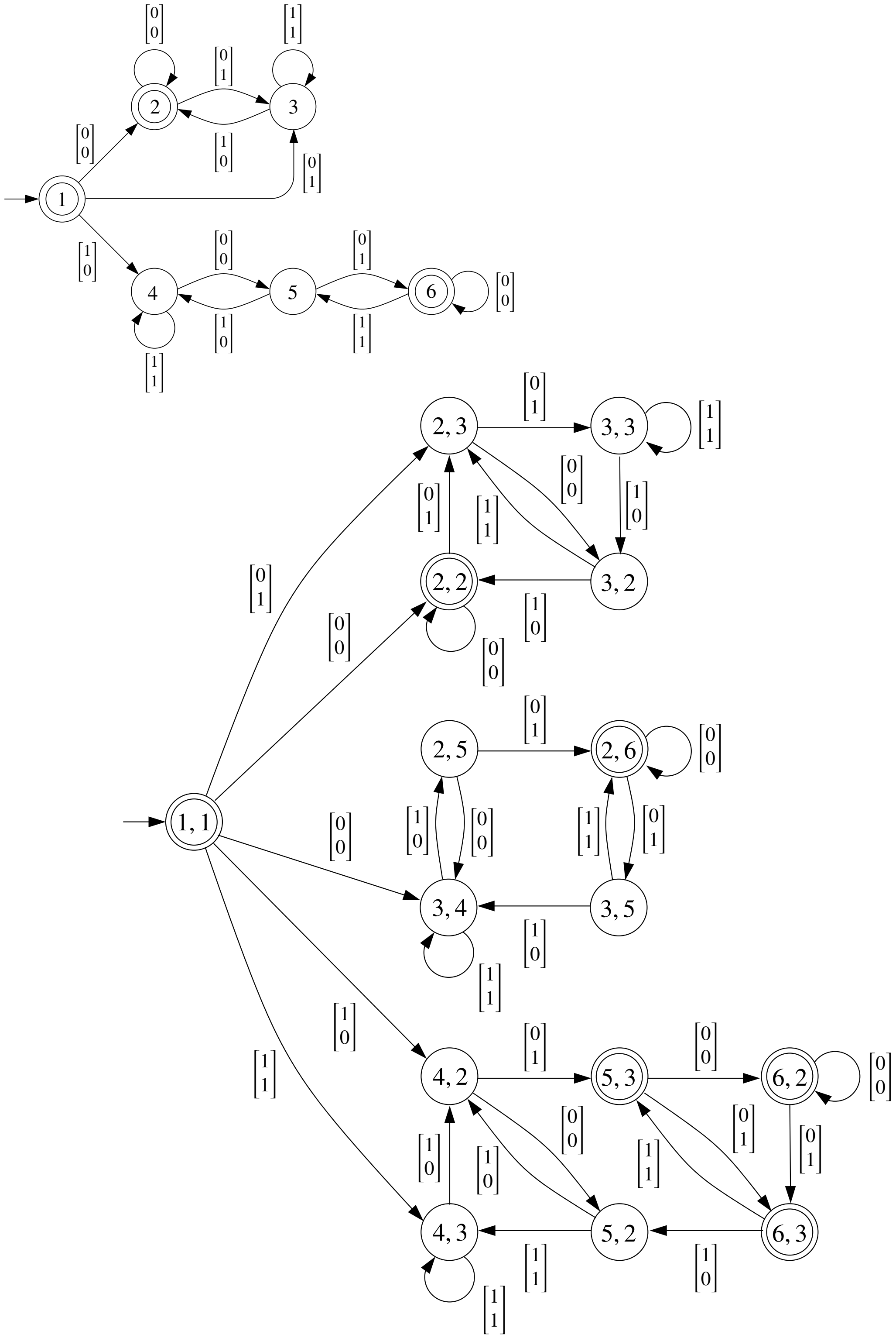
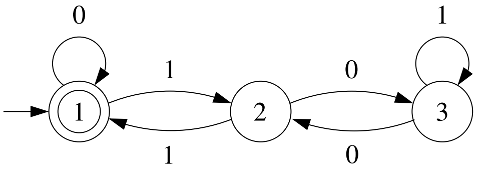


Figure 6.3: A transducer for $f(f(n))$.



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 6.4. 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.

The NFA of Figure 6.4 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! □

6.4 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 $(q, (a_{i_1}, \ldots, a_{i_n}), q')$.

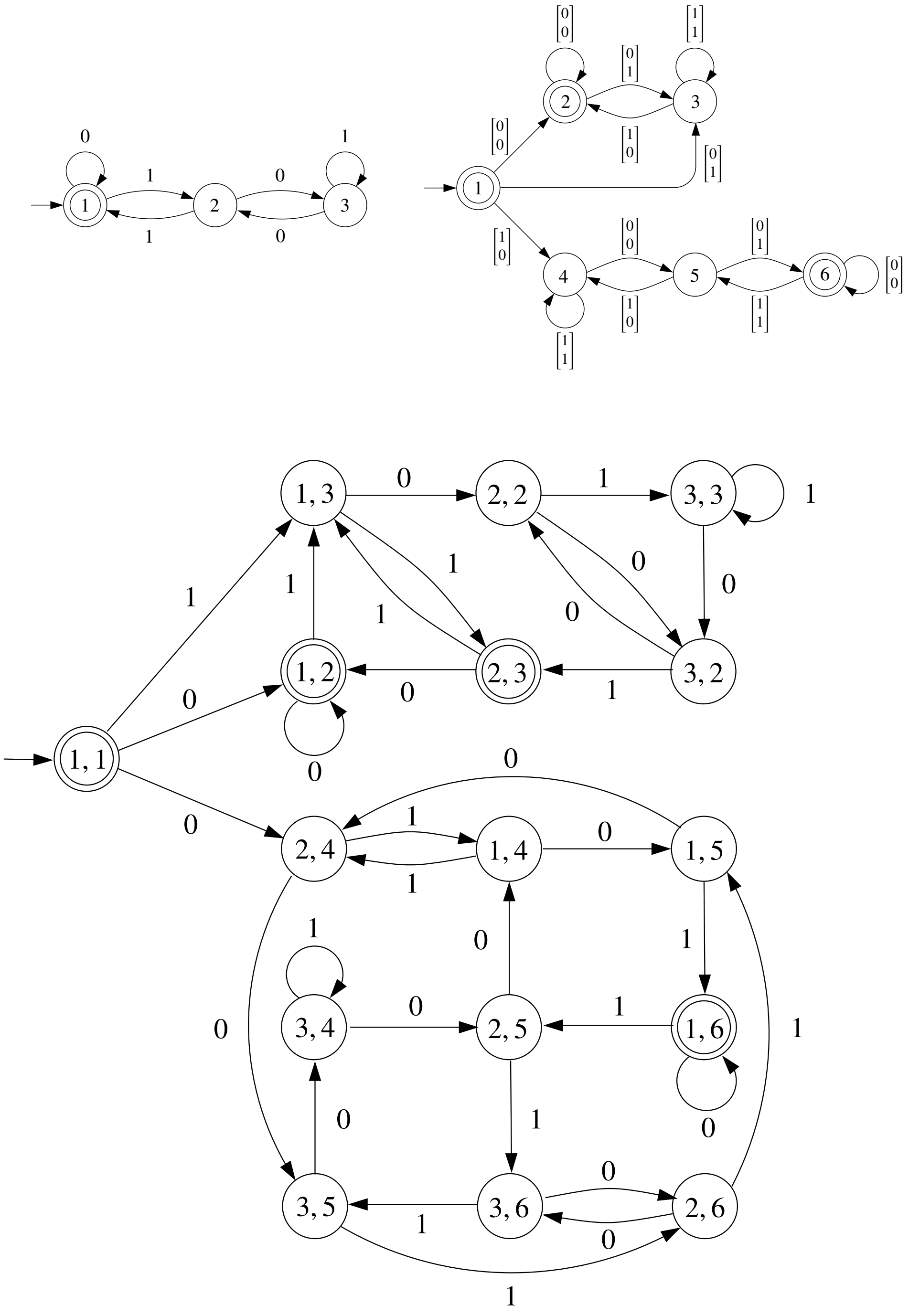


Figure 6.4: Computing $f(n)$ for all multiples of 3.

- 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 $R_1 \circ_{I_1,I_2} R_2 = \{\vec{x}_{K_1 \setminus I_1}\ \vec{x}_{K_2 \setminus I_2} \mid \exists \vec{x} \in R_1, \vec{y} \in R_2 \colon \quad \vec{x}_{I_1} = \vec{y}_{I_2}\}$

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 $\vec{a}_{I_1} = \vec{b}I_2$, 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 73 Let val : $\{0, 1\}^* \to \mathbb{N}$ be such that val(w) is the number represented by w with the "least significant bit first" encoding.

1. Give a transducer that doubles numbers, i.e. a transducer accepting
$$L_1 = \{[x, y] \in (\{0, 1\} \times \{0, 1\})^* : \mathrm{val}(y) = 2 \cdot \mathrm{val}(x)\}.$$

2. Give an algorithm that takes $k \in \mathbb{N}$ as input, and that produces a transducer A_k accepting
$$L_k = \left\{[x, y] \in (\{0, 1\} \times \{0, 1\})^* : \mathrm{val}(y) = 2^k \cdot \mathrm{val}(x)\right\}.$$
(Hint: use (a) and consider operations seen in class.)

3. Give a transducer for the addition of two numbers, i.e. a transducer accepting
$$\{[x, y, z] \in (\{0, 1\} \times \{0, 1\} \times \{0, 1\})^* : \mathrm{val}(z) = \mathrm{val}(x) + \mathrm{val}(y)\}.$$

4. For every $k \in \mathbb{N}_{>0}$, let
$$X_k = \{[x, y] \in (\{0, 1\} \times \{0, 1\})^* : \mathrm{val}(y) = k \cdot \mathrm{val}(x)\}.$$
Suppose you are given transducers A and B accepting respectively X_a and X_b for some $a, b \in \mathbb{N}_{>0}$. Sketch an algorithm that builds a transducer C accepting X_{a+b}. (Hint: use (b) and (c).)

5. Let $k \in \mathbb{N}_{>0}$. Using (b) and (d), how can you build a transducer accepting X_k?

6. Show that the following language has infinitely many residuals, and hence that it is not regular:
$$\left\{[x, y] \in (\{0,1\} \times \{0,1\})^* : \mathrm{val}(y) = \mathrm{val}(x)^2\right\}.$$

Exercise 74 Let $U = \mathbb{N}$ be the universe of natural numbers, and consider the MSBF encoding. Give transducers for the sets of pairs $(n, m) \in \mathbb{N}^2$ such that (a) $m = n + 1$, (b) $m = \lfloor n/2 \rfloor$, (c) $n/4 \leq m \leq 4n$. How do the constructions change for the LSBF encoding?

Exercise 75 Let U be some universe of objects, and fix an encoding of U over Σ^*. Prove or disprove: if a relation $R \subseteq U \times U$ is regular, then the language

$$L_R = \{w_x w_y \mid (w_x, w_y) \text{ encodes a pair } (x, y) \in R\}$$

is regular.

Exercise 76 Let A be an NFA over the alphabet Σ.

(a) Show how to construct a transducer T over the alphabet $\Sigma \times \Sigma$ such that $(w, v) \in L(T)$ iff $wv \in L(A)$ and $|w| = |v|$.

(b) 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 77 In phone dials letters are mapped into digits as follows:

$$\begin{array}{cccc} \text{ABC} \mapsto 2 & \text{DEF} \mapsto 3 & \text{GHI} \mapsto 4 & \text{JKL} \mapsto 5 \\ \text{MNO} \mapsto 6 & \text{PQRS} \mapsto 7 & \text{TUV} \mapsto 8 & \text{WXYZ} \mapsto 9 \end{array}$$

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 (for simplicity, we assume that it contains neither 1 nor 0), 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. Given a number n, show how to construct a NFA recognizing all the words that are mapped to n.

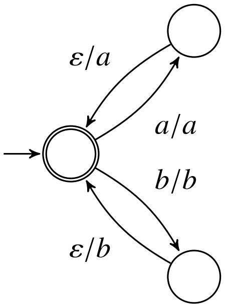
Exercise 78 As we have seen, the application of the **Post**, **Pre** operations to transducers requires to compute the padding closure in order to guarantee that the resulting automaton accepts either all or none of the encodings of a object. The padding closure has been defined for encodings where 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 79 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 $(a_1 \dots a_n, b_1 \dots b_n)$ of the same length. In many applications this is limiting.

An *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

$$q_0 \xrightarrow{(a_1,b_1)} q_1 \xrightarrow{(a_2,b_2)} \cdots \xrightarrow{(a_n,b_n)} q_n \text{ with } a_i, b_i \in \Sigma \cup \{\varepsilon\}$$

such that $w = a_1 \dots a_n$ and $w' = b_1 \dots b_n$. Note that $|w| \leq n$ and $|w'| \leq n$. The relation accepted by the ε-transducer T is denoted by $L(T)$. The figure below shows a transducer over the alphabet $\{a, b\}$ that, intuitively, duplicates the letters of a word, e.g., on input *aba* it outputs *aabbaa*. In the figure we use the notation a/b.



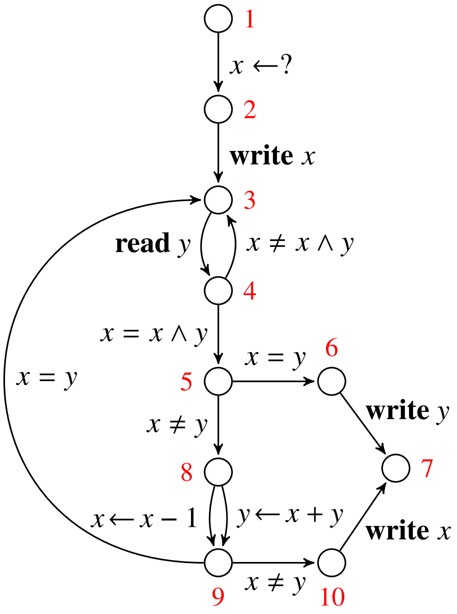
Give an algorithm $Post^{\varepsilon}(A, T)$ that, given a NFA A and an ε-transducer T, both over the same alphabet Σ, returns a NFA recognizing the language

$$post_{T_\varepsilon}(A) = \{w \mid \exists\, w' \in L(A) \text{ such that } (w', w) \in L(T)\}$$

Hint: View ε as an additional alphabet letter.

Exercise 80 Transducers can be used to capture the behaviour of simple programs. The figure below shows a program P and its control-flow diagram (the instruction **end** finishes the execution of the program).

```
bool $x, y$ init 0
$x \leftarrow ?$
write x
while $true$ do
    read y until $y = x \wedge y$
    if $x = y$ then write y end
    $x \leftarrow x - 1$ or $y \leftarrow x + y$
    if $x \neq y$ then write x end
```



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 I/O-relation of P is the set of pairs $(w_I, w_O) \in \{0, 1\} * \times \{0, 1\}^*$ such that there is an execution of P during which P reads w_I and writes w_O.

Let $[i, x, y]$ denote the configuration of P in which P is at node i of the control-flow diagram, and the values of x and y are x and y, respectively. The initial configuration of P is $[1, 0, 0]$. By executing the first instruction P moves nondeterministically to one of the configurations $[2, 0, 0]$ and $[2, 1, 0]$; no input symbol is read and no output symbol is written. Similarly, by executing its second instruction, the program P moves from $[2, 1, 0]$ to $[3, 1, 0]$ while reading nothing and writing 1.

(a) Give an ε-transducer recognizing the I/O-relation of P.

(b) Can an overflow error occur? (That is, can a configuration be reached in which the value of X or y is not 0 or 1?)

(c) Can node 10 of the control-flow graph be reached?

(d) What are the possible values of x upon termination, i.e. upon reaching **end**?

(e) Is there an execution during which P reads 101 and writes 01?

(f) 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 when P is fed inputs from I, 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) belongs to the I/O-relation of P.

Exercise 81 In Exercise 79 we have shown how to compute pre- and post-images of relations described by ε-transducers. In this exercise we show that, unfortunately, and unlike standard transducers, ε-transducers are not closed under intersection.

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

(b) Show that no ε-transducer recognizes $R_1 \cap R_2$.

Exercise 82 (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". Sketch ε-transducers for the following operations, each of which is informally defined by means of two examples. In each example, when the transducer reads the string on the left 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 7

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 6 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 6, 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*. For instance, if the universe consists of the numbers in the range $[0..2^{32} - 1]$, its objects can be encoded by words over $\{0, 1\}$ of length 32. Since all encodings have the same length, padding is not required to represent tuples of objects, and we can assume that each object is encoded by exactly one word. As in Chapter 4, we also assume that each word encodes some object. Operations on objects correspond to operations on languages, but complementation requires some care. If $X \subset U$ is encoded as a language $L \subseteq \Sigma^k$ for some number $k \geq 0$, then the complement set $U \setminus X$ is not encoded by $\overline{L}$ (which contains words of any length) but by $\overline{L} \cap \Sigma^k$.

7.1 Fixed-length Languages and the Master Automaton

Definition 7.1 *A language $L \subseteq \Sigma^*$ has* length $n \geq 0$ *if every word of L has length n. If L has length n for some $n \geq 0$, then we say that L is a* fixed-length *language, or that it* has fixed-length.

Some remarks are in order:

- According to this definition, the empty language has length n for *every* $n \geq 0$ (the assertion "every word of L has length n" is vacuously true). This is useful, because then the complement of a language of length n has also length n.

- There are exactly two languages of length 0: the empty language $\emptyset$, and the language $\{\epsilon\}$ containing only the empty word.

- Every fixed-length language contains only finitely many words, and so it is regular.

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.

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 7.2 *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 7.3 Figure 7.1 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 7.4 *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:

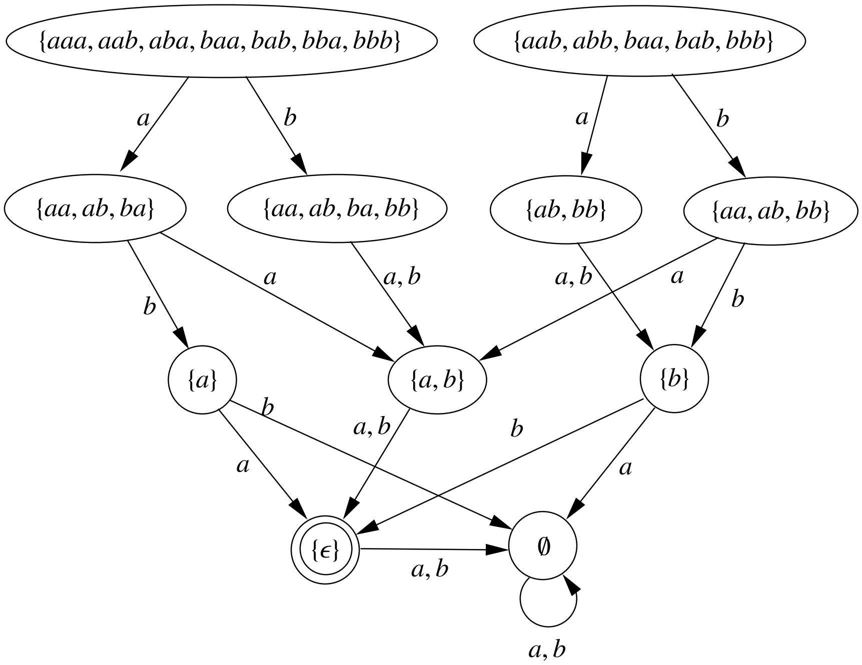


Figure 7.1: A fragment of the master automaton for the alphabet $\{a, b\}$

Proposition 7.5 *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 7.4, distinct states of A_L recognize distinct languages. By Corollary 3.12 (a DFA is minimal if and only if distinct states recognized different languages) A_L is minimal. □

7.2 A Data Structure for Fixed-length Languages

Proposition 7.5 allows us 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 7.6 *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 $F_\mathcal{L} = F_M$.*

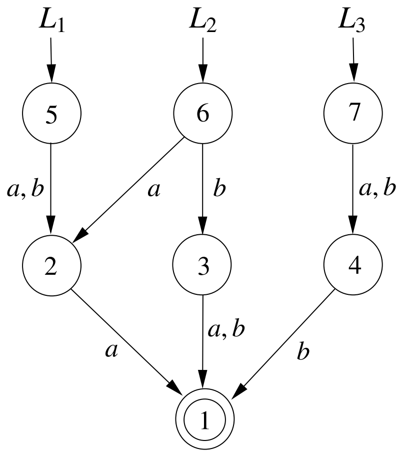


Figure 7.2: The multi-DFA for $\{L_1, L_2, L_3\}$ with $L_1 = \{aa, ba\}$, $L_2 = \{aa, ba, bb\}$, and $L_3 = \{ab, bb\}$.

Example 7.7 Figure 7.2 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 $L_3 = \{ab, bb\}$. For clarity 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. Along the chapter we denote the state identifiers of the state for the languages $\emptyset$ and $\{\epsilon\}$ by $q_\emptyset$ and q_ϵ, respectively.

The multi-DFA is represented by a table containing a node for each state, with the exception of the nodes $q_\emptyset$ and q_ϵ. The table for the multi-DFA of Figure 7.2, where state identifiers are numbers, is

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

The procedure *make*(s)**.** The algorithms on multi-DFAs use a procedure *make*(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. If s is the tuple all whose components are $q_\emptyset$, then *make*(s) returns $q_\emptyset$. The procedure assumes that all the states of the tuple s (with the exception of $q_\emptyset$ and q_ϵ) appear in T.[1] For instance, if T is the

[1] Notice that the procedure makes use of the fact that no two states of the table have the same successor tuple.

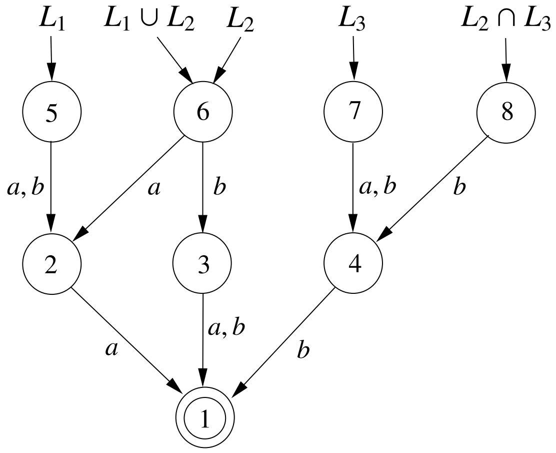


Figure 7.3: The multi-DFA for $\{L_1, L_2, L_3, L_1 \cup L_2, L_2 \cap L_3\}$

table above, then *make*$(2, 2)$ returns 5, but *make*$(3, 2)$ adds a new row, say $8, 3, 2$, and returns 8.

7.3 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 $n - 1$. This allows to design recursive algorithms that directly compute the result when the inputs are languages of length 0, and reduce the problem of computing the result for languages of length $n \geq 1$ 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 union and intersection. 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 7.2. 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 7.3, 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*, we present an algorithm that returns the state of the master automaton recognizing $L_1 \cap L_2$ (the algorithm for $L_1 \cup L_2$ is analogous). The properties

- if $L_1 = \emptyset$ or $L_2 = \emptyset$, then $L_1 \cap L_2 = \emptyset$;
- if $L_1 = \{\epsilon\}$ and $L_2 = \{\epsilon\}$, then $L_1 \cap L_2 = \{\epsilon\}$;
- if $L_1, L_2 \notin \{\emptyset, \{\epsilon\}\}$, 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*(q_1, q_2) shown in Table 7.1. Assume the states q_1, q_2 recognize the languages L_1, L_2 of the same length. We say that q_1, q_2 have the same length. The algorithm returns the state identifier $q_{L_1 \cap L_2}$. 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 $q_1 = q_\epsilon = q_2$, the algorithm returns q_ϵ. This deals with all the cases in which $q_1, q_2 \in \{q_\emptyset, q_\epsilon\}$ (and some more, which does no harm). If $q_1, q_2 \notin \{q_\emptyset, q_\epsilon\}$, 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*$(r_1, \ldots, r_n)$ (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*$(q_1^{a_i}, q_2^{a_i})$.

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*(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($q_1, q_2$)
Input: states $q_1, q_2$ of the same length
Output: state recognizing $L(q_1) \cap L(q_2)$
1  if $G(q_1, q_2)$ is not empty then return $G(q_1, q_2)$
2  if $q_1 = q_\emptyset$ or $q_2 = q_\emptyset$ then return $q_\emptyset$
3  else if $q_1 = q_\varepsilon$ and $q_2 = q_\varepsilon$ then return $q_\epsilon$
4  else /* $q_1, q_2 \notin \{q_\emptyset, q_\varepsilon\}$ */
5     for all $i = 1, \ldots, m$ do $r_i \leftarrow inter(q_1^{a_i}, q_2^{a_i})$
6     $G(q_1, q_2) \leftarrow$ make$(r_1, \ldots, r_m)$
7     return $G(q_1, q_2)$
```

Table 7.1: Algorithm *inter*

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*(q_1, q_2) and *inter*(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*(q, q) always returns q, no need to compute anything either.

Example 7.8 Consider the multi-DFA at the top of Figure 7.4, but without the blue states. State 0, accepting the empty language, is again not shown. The tree at the bottom of the figure graphically describes the run of *inter*(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 stands for a recursive call to *inter* with arguments q and q' that returns q''. For instance, the node 2,4 ↦ 2 indicates that *inter* is called with arguments 2 and 4 and the call returns state 2. Let us see why is this so. 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 *inter*(2, 4) returns a state with 1 as a-successor and 0 as b-successor. Since this state already exists (it is state 2), *inter*(2, 4) returns 2. On the other hand, *inter*(9,10) creates and returns a new state: its two "children calls" return 5 and 6, and so a new state with state 5 and 6 as a- and b-successors must be created.

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

Fixed-length complement. Recall that if a set $X \subseteq U$ is encoded by a language L of length n, then the set $U \setminus X$ is encoded by the *fixed-length complement* $\Sigma^n \setminus L$, which we denote by $\overline{L}^{n}$. Since the empty language has all lengths, we have e.g. $\overline{\emptyset}^{2} = \Sigma^2$, but $\overline{\emptyset}^{3} = \Sigma^3$ and $\overline{\emptyset}^{0} = \Sigma^0 = \{\epsilon\}$.

Given the state of the master automaton recognizing L, we compute the state recognizing $\overline{L}^{n}$ with the help of these properties:

- If L has length 0 and $L = \emptyset$ then $\overline{L}^{0} = \{\epsilon\}$.
- If L has length 0 and $L = \{\epsilon\}$, then $\overline{L}^{0} = \emptyset$.
- If L has length $n \geq 1$, then $\left(\overline{L}^{n}\right)^a = \overline{L^a}^{(n-1)}$.
  (Observe that $w \in \left(\overline{L}\right)^a$ iff $aw \notin L$ iff $w \notin L^a$ iff $w \in \overline{L^a}$.)

We obtain the algorithm of Table 7.9. If the master automaton has n states reachable from q, then the operation has complexity $\mathcal{O}(n)$.

Example 7.9 Consider again the multi-DFA at the top of Figure 7.5 without the blue states. The tree of recursive calls at the bottom of the figure graphically describes the run of *comp*(4, 12) (that is, we compute the node for the complement of the language recognized from state 12, which has length 4). For instance, *comp*(1,2) generates two recursive calls, first *comp*(0,1) (the a-successor of 2), and then *comp*(0,0). The calls returs 0 and 1, respectively, and so *comp*(1,2) returns 3. Observe

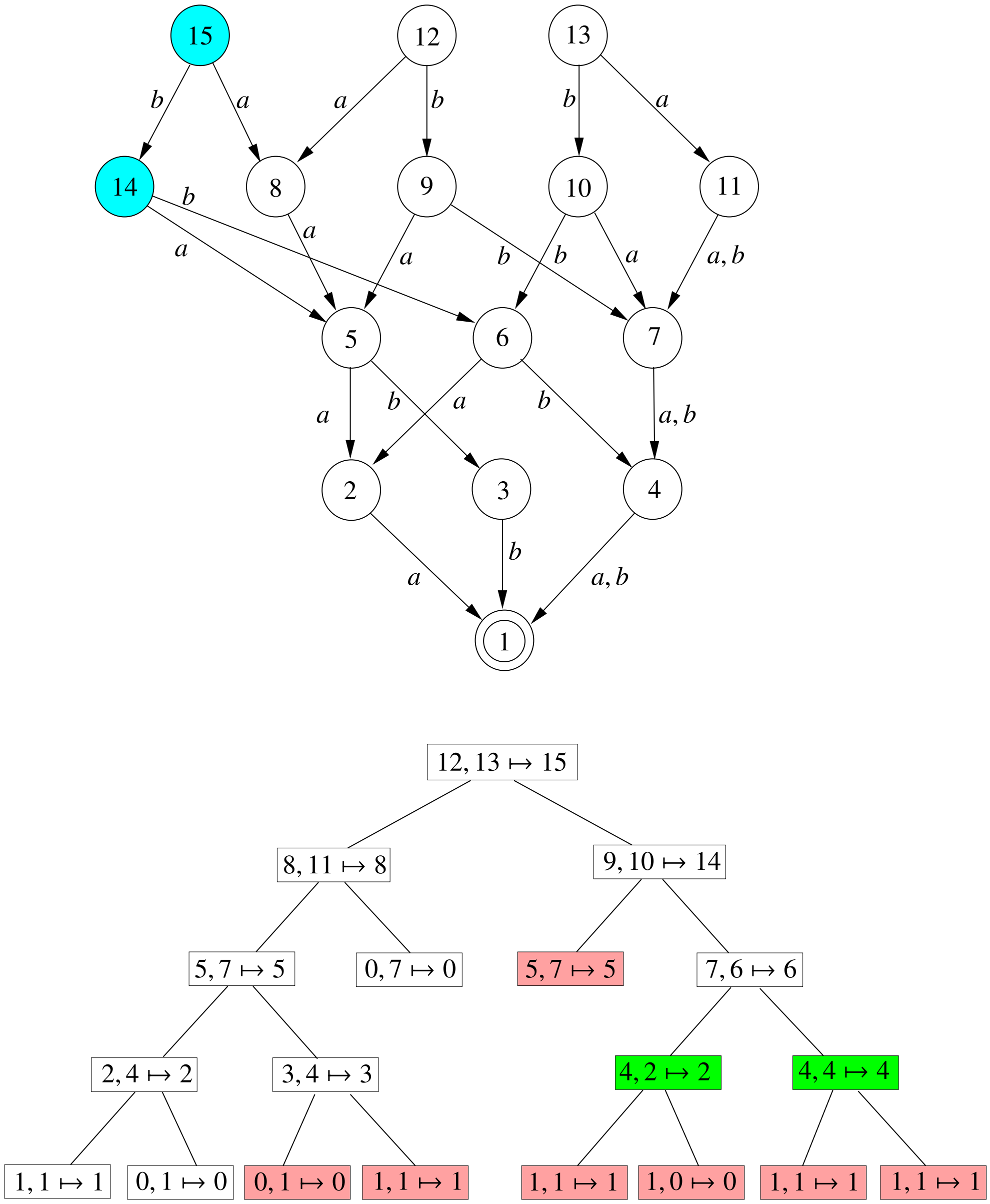


Figure 7.4: An execution of *inter*.

```
$comp(n, q)$
**Input:** length $n$, state $q$ of length $n$
**Output:** state recognizing $\overline{L(q)}^{n}$
if $G(n, q)$ is not empty then return $G(n, q)$
if $n = 0$ and $q = q_\emptyset$ then return $q_\epsilon$
else if $n = 0$ and $q = q_\epsilon$ then return $q_\emptyset$
else  /* $n \geq 1$ */
    for all $i = 1, \ldots, m$ do $r_i \leftarrow comp(n-1, q^{a_i})$
    $G(n, q) \leftarrow$ make$(r_1, \ldots, r_m)$
    return $G(n, q)$
```

Table 7.2: Algorithm *comp*

how the call *comp*(2,0) returns 7, the state accepting $\{a, b\}^2$. Pink nodes correspond again to calls for which *comp* just looks up the result in G. Green nodes correspond to calls whose result is directly computed by a more efficient version of *comp* that applies the following rule: if $comp(i, j)$ returns k, then $comp(i, k)$ returns j. □

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 emptiness can be checked in constant time..

Fixed-length universality. 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, we use the properties

- 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 $a \in \Sigma$;

that lead to the algorithm of Table 7.3. For a better algorithm see Exercise 85.

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.

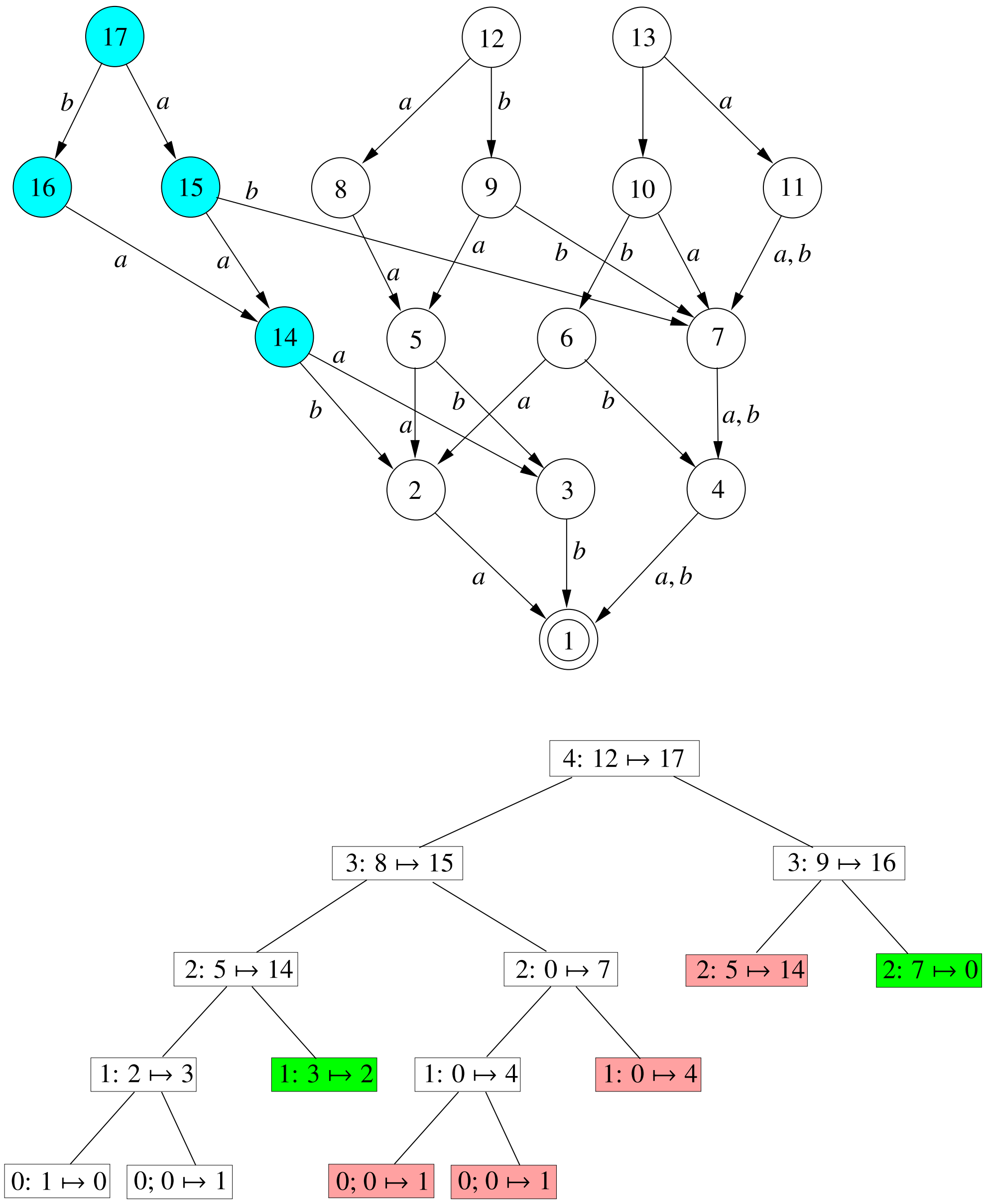


Figure 7.5: An execution of *comp*.

```
$univ(q)$
Input: state $q$
Output: true if $L(q)$ is fixed-length universal,
        false otherwise
if $G(q)$ is not empty then return $G(q)$
if $q = q_\emptyset$ then return false
else if $q = q_\epsilon$ then return true
else  $/ * \; q \neq q_\emptyset$ and $q \neq q_\epsilon \; * /$
    $G(q) \leftarrow$ and$(univ(q^{a_1}), \ldots, univ(q^{a_m}))$
    return $G(q)$
```

Table 7.3: Algorithm *univ*

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 a constant time algorithm. This solution, however, assumes that the two input nodes come from the same table. 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 algorithm *eq*2 of Table 7.4, which assumes that q_i belongs to a table T_i, and that both tables assign state identifiers $q_{\emptyset 1}$ and $q_{\emptyset 2}$ to the empty language.

```
$eq2(q_1, q_2)$
Input: states $q_1$, $q_2$ of different tables, of the same length
Output: true if $L(q_1) = L(q_2)$, false otherwise
    if $G(q_1, q_2)$ is not empty then return $G(q_1, q_2)$
    if $q_1 = q_{\emptyset 1}$ and $q_2 = q_{\emptyset 2}$ then $G(q_1, q_2) \leftarrow$ true
    else if $q_1 = q_{\emptyset 1}$ and $q_2 \neq q_{\emptyset 2}$ then $G(q_1, q_2) \leftarrow$ false
    else if $q_1 \neq q_{\emptyset 1}$ and $q_2 = q_{\emptyset 2}$ then $G(q_1, q_2) \leftarrow$ false
    else  $/ * \; q_1 \neq q_{\emptyset 1}$ and $q_2 \neq q_{\emptyset 2} \; * /$
        $G(q_1, q_2) \leftarrow$ and(eq$(q_1^{a_1}, q_2^{a_1}), \ldots,$ eq$(q_1^{a_m}, q_2^{a_m})$)
    return $G(q_1, q_2)$
```

Table 7.4: Algorithm *eq*2

7.4 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 7.5 returns the state of the master automaton recognizing L. 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 recognizing $L(S)$. 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 the states of S recognize fixed-length languages, 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 $L(S) = \{\epsilon\}$);
- if $S \neq \emptyset$ and $S \cap F = \emptyset$, then $L(S) = \displaystyle\bigcup_{i=1}^{n} a_i \cdot L(S_i)$, where $S_i = \delta(S, a_i)$.

These properties lead to the recursive algorithm of Table 7.5. The procedure *state*(S) uses a table G of results, initially empty. When *state*(S) is computed for the first time, the result is memoized in $G(S)$, and any subsequent call directly reads the result from G. The algorithm has exponential complexity, because, in the worst case, it may call *state*(S) for every set $S \subseteq Q$. To show that an exponential blowup is unavoidable, consider the family $\{L_n\}_{n \geq 0}$ of languages, where $L_n = \{ww' \mid w, w' \in \{0,1\}^n \text{ and } w \neq w'\}$. While L_n can be recognized be an NFAs of size $\mathcal{O}(n^2)$, its minimal DFA has $\mathcal{O}(2^n)$ states: for every $u, v \in \Sigma^n$ if $u \neq v$ then $L_n^u \neq L_v$, because $v \in L_n^u$ but $v \notin L_n^v$.

Example 7.10 Figure 7.6 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. □

7.5 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 projection, join, *pre*, and *post* operations can be however implemented more efficiently as in Chapter 6.

We start with an observation on encodings. In Chapter 6 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.

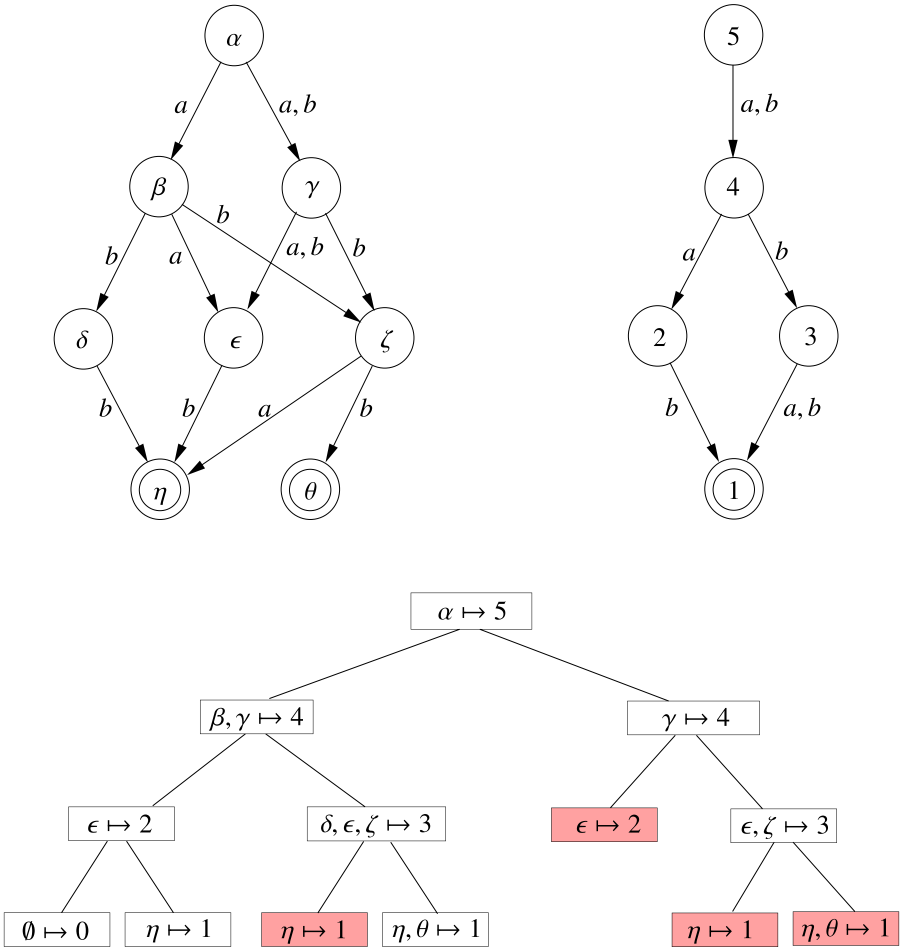


Figure 7.6: Run of *det&min* on an NFA for a fixed-length language

```
*det&min*($A$)
**Input:** NFA $A = (Q, \Sigma, \delta, Q_0, F)$
**Output:** master state recognizing $L(A)$
    **return** *state*($Q_0$)

*state*($S$)
**Input:** set $S \subseteq Q$ recognizing languages of the same length
**Output:** state recognizing $L(S)$
    **if** $G(S)$ is not empty **then return** $G(S)$
    **else if** $S = \emptyset$ **then return** $q_\emptyset$
    **else if** $S \cap F \neq \emptyset$ **then return** $q_\epsilon$
    **else** $/ * S \neq \emptyset$ and $S \cap F = \emptyset * /$
        **for all** $i = 1, \ldots, m$ **do** $S_i \leftarrow \delta(S, a_i)$
        $G(S) \leftarrow make(state(S_1), \ldots, state(S_m))$;
        **return** $G(S)$
```

Table 7.5: Algorithm *det&min*.

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.

The basic definitions on fixed-length languages extend easily to fixed-length relations. A word relation $R \subseteq \Sigma^* \times \Sigma^*$ has length $n \geq 0$ if for all pairs $(w_1, w_2) \in R$ 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 has fixed-length

Recall that a transducer T accepts a pair $(w_1, w_2) \in \Sigma^* \times \Sigma^*$ if $w_1 = a_1 \ldots a_n$, $w_2 = b_1 \ldots b_n$, and T accepts the word $(a_1, b_1) \ldots (a_n, b_n) \in \Sigma^* \times \Sigma^*$. A fixed-length transducer accepts a relation $R \subseteq X \times X$ if it recognizes the word relation $\{(w_x, w_y) \mid (x, y) \in R\}$.

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 $R^{[a,b]}$. 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, $F_M = \{(\epsilon, \epsilon)\}$, and $\delta_M \colon Q_M \times (\Sigma \times \Sigma) \to Q_M$ is given by $\delta_M(R, [a, b]) = R^{[a,b]}$ for every $q \in Q_M$ and $a, b \in \Sigma$. As in the language case, the minimal deterministic transducer recognizing a fixed-length relation R is the fragment of the master transducer containing the states reachable from 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 projection The implementation of the projection operation of Chapter 6 may yield a nondterministic transducer, even if the initial transducer is deterministic. So we need a different implementation. We observe that projection can be reduced to *pre* or *post*: the projection of a binary relation R onto its first component is equal to $pre_R(\Sigma^*)$, and the projection onto the second component to $post_R(\Sigma^*)$. So we defer dealing with projection until the implementation of *pre* and *post* have been discussed.

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

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

which lead to the algorithm of Figure 7.7, where *union* is defined similarly to *inter*. The complexity is exponential in the worst case: if $t(n)$ denotes the worst-case complexity for two states of length n, then we have $t(n) = \mathcal{O}(t(n-1)^2)$, because *union* has quadratic worst-case complexity. This exponential blowup is unavoidable. We prove it later for the projection operation (see Example 7.11), which is a special case of *pre* and *post*, which in turn can be seen as variants of join.

```
join(r_1, r_2)
Input: states $r_1, r_2$ of transducer table of the same length
Output: state recognizing $L(r_1) \circ L(r_2)$
if $G(r_1, r_2)$ is not empty then return $G(r_1, r_2)$
if $r_1 = q_\emptyset$ or $r_2 = q_\emptyset$ then return $q_\emptyset$
else if $r_1 = q_\epsilon$ and $r_2 = q_\epsilon$ then return $q_\epsilon$
else  / * $q_\emptyset \neq r_1 \neq q_\epsilon$ and $q_\emptyset \neq r_2 \neq q_\epsilon$ * /
    for all $(a_i, a_j) \in \Sigma \times \Sigma$ do
        $r_{i,j} \leftarrow union\left(join\left(r_1^{[a_i,a_1]}, r_2^{[a_1,a_j]}\right), \ldots, join\left(r_1^{[a_i,a_m]}, r_2^{[a_m,a_j]}\right)\right)$
    $G(r_1, r_2) = make(r_{1,1}, \ldots, \ldots, r_{m,m})$
    return $G(r_1, r_2)$
```

Figure 7.7: Algorithm *join*

Fixed-length *pre* and *post*. Recall that in the fixed-length case we do not need any padding symbol. Then, given a fixed-length language L and a relation R, $pre_R(L)$ admits an inductive definition that we now derive. We have the following properties:

- if $R = \emptyset$ or $L = \emptyset$, then $pre_R(L) = \emptyset$;
- if $R = \{\, [\epsilon, \epsilon] \,\}$ and $L = \{\epsilon\}$, then $pre_R(L) = \{\epsilon\}$;
- if $\emptyset \neq R \neq \{[\epsilon, \epsilon]\}$ and $\emptyset \neq L \neq \{\epsilon\}$, then $pre_R(L) = \displaystyle\bigcup_{a,b \in \Sigma} a \cdot pre_{R^{[a,b]}}(L^b)$,

  where $R^{[a,b]} = \{w \in (\Sigma \times \Sigma)^* \mid [a, b]w \in R\}$.

The first two properties are obvious. For the last one, observe that all pairs of R have length at least one, and so every word of $pre_R(L)$ also has length at least one. Now, given an arbitrary word $aw_1 \in \Sigma\Sigma^*$, we have

$$\begin{array}{cl} & aw_1 \in pre_R(L) \\ \Leftrightarrow & \exists bw_2 \in L \colon [aw_1, bw_2] \in R \\ \Leftrightarrow & \exists b \in \Sigma \; \exists w_2 \in L^b \colon [w_1, w_2] \in R^{[a,b]} \\ \Leftrightarrow & \exists b \in \Sigma \colon w_1 \in pre_{R^{[a,b]}}(L^b) \\ \Leftrightarrow & aw_1 \in \displaystyle\bigcup_{b \in \Sigma} a \cdot pre_{R^{[a,b]}}(L^b) \end{array}$$

and so $pre_R(L) = \displaystyle\bigcup_{a,b \in \Sigma} a \cdot pre_{R^{[a,b]}}(L^b)$ These properties lead to the recursive algorithm of Table 7.5, which accepts as inputs a state of the transducer table for a relation R and a state of the automaton table for a language L, and returns the state of the automaton table recognizing $pre_R(L)$. The transducer table is not changed by the algorithm.

```
$pre(r, q)$
**Input:** state $r$ of transducer table and state $q$ of automaton table,
of the same length
**Output:** state recognizing $pre_{L(r)}(L(q))$
if $G(r, q)$ is not empty then return $G(r, q)$
if $r = r_\emptyset$ or $q = q_\emptyset$ then return $q_\emptyset$
else if $r = r_\epsilon$ and $q = q_\epsilon$ then return $q_\epsilon$
else
   for all $a_i \in \Sigma$ do
      $q'_i \leftarrow union\left(pre\left(r^{[a_i,a_1]}, q^{a_1}\right), \ldots, pre\left(r^{[a_i,a_m]}, q^{a_m}\right)\right)$
   $G(q, r) \leftarrow make(q'_1, \ldots, q'_m)$
   return $G(q, r)$
```

Table 7.6: Algorithm *pre*.

As promised, we can now give an implementation of the operation that projects a relation R onto its first component. It suffices to give a dedicated algorithm for $pre_R(\Sigma^*)$, shown in Table 7.5.

```
$pro_1(r)$
Input: state $r$ of transducer table
Output: state recognizing $proj_1(L(r))$
    if $G(r)$ is not empty then return $G(r)$
    if $r = r_\emptyset$ then return $q_\emptyset$
    else if $r = r_\epsilon$ then return $q_\epsilon$
    else
      for all $a_i \in \Sigma$ do
        $q'_i \leftarrow union\left(pro_1\left(r^{[a_i,a_1]}\right), \ldots, pro_1\left(r^{[a_i,a_m]}\right)\right)$
      $G(r) \leftarrow make(q'_1, \ldots, q'_m)$
      return $G(r)$
```

Table 7.7: Algorithm pro_1.

Algorithm pro_1 has exponential worst-case complexity. As in the case of *join*, the reason is the quadratic blowup introduced by *union* when the recursion depth increases by one. The next example shows that projection is inherently exponential.

Example 7.11 Consider the relation $R \subseteq \Sigma^{2n} \times \Sigma^{2n}$ given by

$$R = \{(w_1 x w_2 y w_3,\ 0^{|w_1|}10^{|w_2|}10^{|w_3|}) \mid x \neq y, |w_2| = n \text{ and } |w_1 w_3| = n - 2\} .$$

That is, R contains all pairs of words of length $2n$ whose first word has a position $i \leq n$ such that the letters at positions i and $i + n$ are distinct, and whose second word contains only 0's but for two 1's at the same two positions. It is easy to see that the minimal deterministic transducer for R has $\mathcal{O}(n^2)$ states (intuitively, it memorizes the letter x above the first 1, reads $n - 1$ letters of the form $(z, 0)$, and then reads $(y, 1)$, where $y \neq x$). On the other hand, we have

$$proj_1(R) = \{ww' \mid w, w' \in \Sigma^n \text{ and } w \neq w'\} ,$$

whose minimal DFA, as shown when discussing *det&min*, has $\mathcal{O}(2^n)$ states. So any algorithm for projection has $\Omega(2^{\sqrt{n}})$ complexity. □

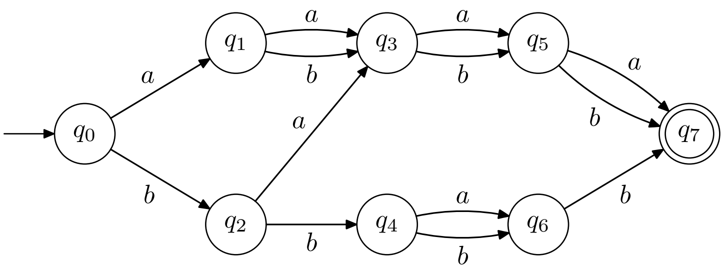
Slight modifications of this example show that *join*, *pre*, and *post* are inherently exponential as well.

7.6 Decision Diagrams

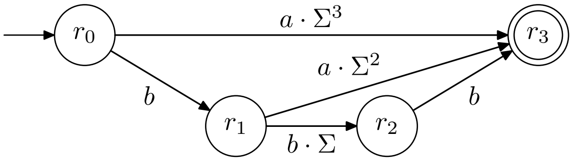
Binary Decision Diagrams, BDDs for short, are a very popular data structure for the representation and manipulation of boolean functions. In this section we show that they can be seen as minimal automata of a certain kind.

Given a boolean function $f(x_1, \dots, x_n) : \{0,1\}^n \to \{0,1\}$, 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 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:

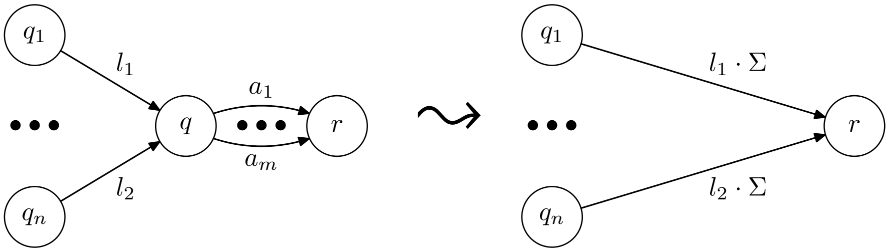


Its language can be described as follows: after reading an a, accept any word of length three; after reading ba, accept any word of length 2; after reading bb, accept any two-letter word whose last letter is a b. Following this description, the language can also be more compactly described by an automaton with regular expressions as transitions:



We call such an automaton a *decision diagram* (DD). The intuition behind this name is that, if we view states as points at which a decision is made, namely which should be the next state, then states q_1, q_3, q_4, q_6 do not correspond to any real decision; whatever the next letter, the next state is the same. As we shall see, the states of minimal DD will always correspond to "real" decisions.

Section 7.6.1 shows that the minimal DD for a fixed-length language 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 minimal DD by "reversing" the rule. This already allows us to use DDs as a data structure for fixed-length languages, but only through conversion to minimal DFAs: to compute an operation using minimal DDs, expand them to minimal DFAs, conduct the operation, and convert the result back. Section

7.6.2 shows how to do better by directly defining the operations on minimal DDs, bypassing the minimal DFAs.

7.6.1 Decision Diagrams and Kernels

A *decision diagram* (DD) is an automaton $A = (Q, \Sigma, \delta, Q_0, F)$ whose transitions are labelled by regular expressions of the form

$$a\Sigma^n = a\,\underbrace{\Sigma\Sigma\Sigma\ldots\Sigma\Sigma}_{n}$$

and satisfies the following *determinacy condition*: for every $q \in Q$ and $a \in \Sigma$ there is exactly one $k \in \mathbb{N}$ such that $\delta(q, a\Sigma^k) \neq \emptyset$, and for this k there is a state q' such that $\delta(q, a\Sigma^k) = \{q'\}$. Observe that DFAs are special DDs in which $k = 0$ for every state and every letter.

We introduce the notion of kernel, and kernel of a fixed-length language.

Definition 7.12 *A fixed-length language L over an alphabet Σ is a* kernel *if $L = \emptyset$, $L = \epsilon$, or there are $a, b \in \Sigma$ such that $L^a \neq L^b$. The* kernel *of a fixed-length language L, denoted by $\langle L \rangle$, is the unique kernel satisfying $L = \Sigma^k \langle L \rangle$ for some $k \geq 0$.*

Observe that the number k is also unique for every language but $\emptyset$. Indeed, for the empty language we have $\langle \emptyset \rangle = \emptyset$ and so $\emptyset = \Sigma^k \langle \emptyset \rangle$ for every $k \geq 0$.

Example 7.13 Let $\Sigma = \{a, b, c\}$. $L_1 = \{aab, abb, bab, cab\}$ is a kernel because $L_1^a = \{ab, bb\} \neq \{ab\} = L_1^b$, and $\langle L_1 \rangle = L_1$; the language $L_2 = \{aa, ba\}$ is also a kernel because $L_2^a = \{a\} \neq \emptyset = L_2^c$. However, if we change the alphabet to $\Sigma' = \{a, b\}$ then L_2 is no longer a kernel, and we have $\langle L_2 \rangle = \{a\}$. For the language $L_3 = \{aa, ab, ba, bb\}$ over Σ' we have $L_3 = (\Sigma')^2$, and so $k = 2$ and $\langle L_3 \rangle = \{\epsilon\}$. □

The mapping that assigns to ever nonempty, fixed-length language L the pair $(k, \langle L \rangle)$ is a bijection. In other words, L is completely determined by k and $\langle L \rangle$. So a representation of kernels can be extended to a representation of all fixed-length languages. Let us now see how to represent kernels.

The *master decision diagram* (we call it just "the master") has the set of all kernels as states, the kernel $\{\epsilon\}$ as unique final state, and a transition $(K, a\Sigma^k, \langle K^a \rangle)$ for every kernel K and $a \in \Sigma$, where k is equal to the length of K^a minus the length of $\langle K^a \rangle$. (For $K = \emptyset$, which has all lengths, we take $k = 0$.)

Example 7.14 Figure 7.8 shows a fragment of the master for the alphabet $\{a, b\}$ (compare with Figure 7.1). The languages $\{a, b\}$, $\{aa, ab, ba, bb\}$, and $\{ab, bb\}$ of Figure 7.1 are not kernels, and so they are not states of the master either. □

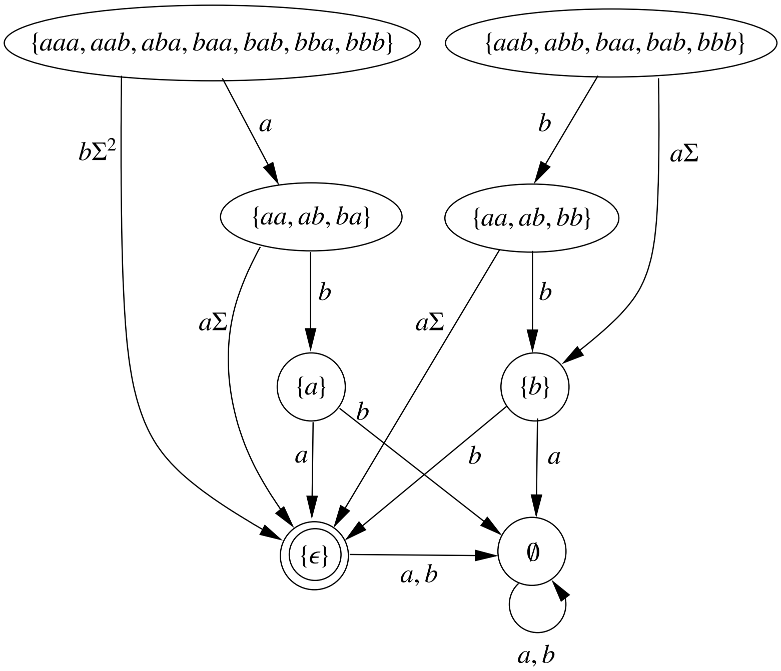

{aaa, aab, aba, baa, bab, bba, bbb}
{aab, abb, baa, bab, bbb}
bΣ²
a
b
aΣ
{aa, ab, ba}
{aa, ab, bb}
aΣ
b
aΣ
b
{a}
{b}
b
a
b
a
{ϵ}
∅
a, b
a, b

Figure 7.8: A fragment of the master decision diagram

The DD A_K for a kernel K is the fragment of the master containing the states reachable from K. It is easy to see that A_K recognizes K. A DD is minimal if no other DD for the same language has fewer states. Observe that, since every DFA is also a DD, the minimal DD for a language has at most as many states as its minimal DD.

The following proposition shows that the minimal DD of a kernel has very similar properties to the minimal DFAs of a regular language. In particular, A_K is always a minimal DD for the kernel K. However, because of a technical detail, it is not the unique minimal DD: The label of the transitions of the master leading to $\emptyset$ can be changed from a to $a\Sigma^k$ for any $k \geq 0$, and from b to $b\Sigma^k$ for any $k \geq 0$, without changing the language. To recover unicity, we redefine minimality: A DD is *minimal* no other DD for the same language has fewer states, and every transition leading to a state from which no word is accepted is labeled by a or b.

Proposition 7.15 *(1) Let A be a DD such that $L(A)$ is a kernel. A is minimal if and only if (i) every state of A recognizes a kernel, and (ii) distinct states of A recognize distinct kernels.*

(2) For every $K \neq \emptyset$, A_K is the unique minimal DD recognizing K.

(3) The result of exhaustively applying the reduction rule to the minimal DFA recognizing a fixed-length language L is the minimal DD recognizing $\langle L \rangle$.

Proof: (1 $\Rightarrow$): For (i), assume A contains a state q such that $L(q)$ is not a kernel. We prove that A is not minimal. Since $L(A)$ is a kernel, q is neither initial nor final. Let k be the smallest number such that A contains a transition $(q, a\Sigma^k, q')$ for some letter a and some state q'. Then $L(q)^a = \Sigma^k L(q')$, and, since $L(q)$ is not a kernel, $L(q)^a = L(q)^b$ for every $b \in \Sigma$. So we have $L(q) = \bigcup_{a\in\Sigma} a\,\Sigma^k L(q') = \Sigma^{k+1}\, L(q')$. Now we perform the following two operations: first, we replace every transition $(q'', b\Sigma^l, q)$ of A by a transition $(q'', b\Sigma^{l+k+1}, q')$; then, we remove q and any other state no longer reachable from the initial state (recall that q is neither initial nor final). The resulting DD recognizes the same language as A, and has at least one state less. So A is not minimal.

For (ii), observe that the quotienting operation can be defined for DDs as for DFAs, and so we can merge states that recognize the same kernel without changing the language. If two distinct states of A recognize the same kernel then the quotient has fewer states than A, and so A is not minimal.

(1 $\Leftarrow$): We show that two DDs A and A' that satisfy (i) and (ii) and recognize the same language are isomorphic, which, together with (1 $\Rightarrow$), proves that they are minimal. It suffices to prove that if two states q, q' of A and A' satisfy $L(q) = L(q')$, then for every $a \in \Sigma$ the (unique) transitions $(q, a\Sigma^k, r)$ and $(q', a\Sigma^{k'}, r')$ satisfy $k = k'$ and $L(r) = L(r')$. Let $L(q) = K = L(q')$. By (1 $\Rightarrow$), both $L(r)$ and $L(r')$ are kernels. But then we necessarily have $L(r) = \langle K^a \rangle = L(q')$, because the only solution to the equation $K = a\Sigma^l K'$, where l and K' are unknowns and K' must be a kernel, is $K' = \langle K^a \rangle$.

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

(3) Let B be a DD obtained by exhaustively applying the reduction rule to A. By (1), it suffices to prove that B satisfies (i) and (ii). For (ii) observe that, since every state of A recognizes a different language, so does every state of B (the reduction rule preserves the recognized languages). For (i), assume that some state q does not recognize a kernel. Without loss of generality, we can choose $L(q)$ of minimal length, and therefore the target states of all outgoing transitions of q recognize kernels. It follows that all of them necessarily recognize $\langle L(q) \rangle$. Since B contains at most one state recognizing $\langle L(q) \rangle$, all outgoing transitions of q have the same target, and so the reduction rule can be applied to q, contradicting the hypothesis that it has been applied exhaustively. □

7.6.2 Operations on Kernels

We use *multi-DDs* to represent sets of fixed-length languages of the same length. A set $\mathcal{L} = \{L_1, \ldots, L_m\}$ is represented by the states of the master recognizing $\langle L_1 \rangle, \ldots, \langle L_m \rangle$ *and* by the common length of $L_1, \ldots, L_m$. Observe that the states and the length completely determine $\mathcal{L}$.

Example 7.16 Figure 7.9 shows the multi-DD for the set $\{L_1, L_2, L_3\}$ of Example 7.7. Recall that $L_1 = \{aa, ba\}$, $L_2 = \{aa, ba, bb\}$, and $L_3 = \{ab, bb\}$. The multi-DD is the result of applying the reduction rule to the multi-automaton of Figure 7.2. We represent the set by the multi-DD and the

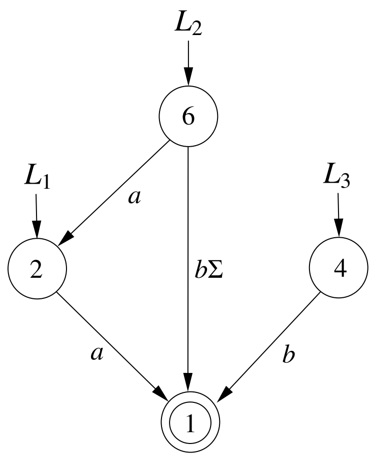


Figure 7.9: The multi-zDFA for $\{L_1, L_2, L_3\}$ with $L_1 = \{aa\}$, $L_2 = \{aa, bb\}$, and $L_3 = \{aa, ab\}$.

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, $\langle L_2 \rangle$ has a different length than $\langle L_1 \rangle$ and $\langle L_3 \rangle$. □

Multi-DDs 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. The table for the multi-DD of Figure 7.9 is:

Ident.	Length	a-succ	b-succ
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 states 2 and 1, we cannot infer which expressions label the transitions 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 are a and $b\Sigma$.

The procedure kmake(l, s). All algorithms call a procedure kmake(l, s) with the following specification. Let K_i be the kernel recognized by the i-th component of s. Then kmake(l, s) returns the kernode for $\langle L \rangle$, where L is the unique language of length l such that $\langle L^{a_i} \rangle = K_i$ for every $a_i \in \Sigma$.

If $K_i \neq K_j$ for some i, j, then kmake(l, s) behaves like make(s): if the current table already contains a kernode $\langle q, l, s \rangle$, then kmake(l, s) returns q; and, if no such kernode exists, then kmake(l, s) creates a new kernode $\langle q, l, s \rangle$ with a fresh identifier q, and returns q.

If $K_1, \ldots, K_m$ are all equal to some kernel K, then we have $L = \bigcup_{i=1}^{m} a_i \Sigma^k K$ for some k, and therefore $\langle L \rangle = \langle \Sigma^{l+1} K \rangle = K$. So `kmake`$(l, t)$ returns the kernode for K. For instance, if T is the table above, then `kmake`$(3, (2, 2))$ returns 3, while `make`$(2, 2)$ creates a new node having 2 as a-successor and b-successor.

Algorithms. The algorithms for operations on kernels are modifications of the algorithms of the previous section. We show how to modify the algorithms for intersection, complement, and for simultaneous determinization and minimization. In the previous section, the state of the master automaton for a language L was the language L itself, and was obtained by recursively computing the states for $L^{a_1}, \ldots, L^{a_m}$ and then applying `make`. Now, the state of the master DD for L is $\langle L \rangle$, and can be obtained by recursively computing states for $\langle L^{a_1} \rangle, \ldots, \langle L^{a_m} \rangle$ and applying `kmake`.

Fixed-length intersection. Given kernels K_1, K_2 of languages L_1, L_2, we compute the state recognizing $K_1 \sqcap K_2 \overset{def}{=} \langle L_1 \cap L_2 \rangle$. [2] We have the obvious property

- if $K_1 = \emptyset$ or $K_2 = \emptyset$, then $K_1 \sqcap K_2 = \emptyset$.

Assume now $K_1 \neq \emptyset \neq K_2$. If the lengths of K_1 and K_2 are l_1, l_2, then since $\langle \Sigma^k L \rangle = \langle L \rangle$ holds for every k, L we have

$$K_1 \sqcap K_2 = \begin{cases} \langle \Sigma^{l_2 - l_1} K_1 \cap K_2 \rangle & \text{if } l_1 < l_2 \\ \langle K_1 \cap \Sigma^{l_1 - l_2} K_2 \rangle & \text{if } l_1 > l_2 \\ \langle K_1 \cap K_2 \rangle & \text{if } l_1 = l_2 \end{cases}$$

which allows us to obtain the state for $K_1 \sqcap K_2$ by computing states for

$$\langle\, (\Sigma^{l_1 - l_2} K_1 \cap K_2)^a \,\rangle \quad , \quad \langle\, (K_1 \cap \Sigma^{l_2 - l_1} K_2)^a \,\rangle \quad \text{or} \quad \langle\, (K_1 \cap K_2)^a \,\rangle$$

for every $a \in \Sigma$, and applying `kmake`. These states can be computed recursively by means of:

- if $l_1 < l_2$ then $\langle\, (\Sigma^{l_2 - l_1} K_1 \cap K_2)^a \,\rangle = \langle\, \Sigma^{l_2 - l_1 - 1} K_1 \cap K_2^a \,\rangle = K_1 \sqcap \langle K_2^a \rangle$;
- if $l_1 > l_2$ then $\langle\, (K_1 \cap \Sigma^{l_1 - l_2} K_2)^a \,\rangle = \langle\, K_1^a \cap \Sigma^{l_1 - l_2 - 1} K_2 \,\rangle = \langle K_1^a \rangle \sqcap K_2$;
- if $l_1 = l_2$ then $\langle\, (K_1 \cap K_2)^a \,\rangle = \langle\, K_1^a \cap K_2^a \,\rangle = \langle K_1^a \rangle \sqcap \langle K_2^a \rangle$;

which leads to the algorithm of Table 7.8.

Example 7.17 Example 7.8 shows a run of *inter* on the two languages represented by the multi-DFA at the top of Figure 7.4. The multi-DD for the same languages is shown at the top of Figure 7.10, 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. □

[2] $\sqcap$ is well defined because $\langle L_1 \rangle = \langle L_1' \rangle$ and $\langle L_2 \rangle = \langle L_2' \rangle$ implies $\langle L_1 \cap L_2 \rangle = \langle L_1' \cap L_2' \rangle$.

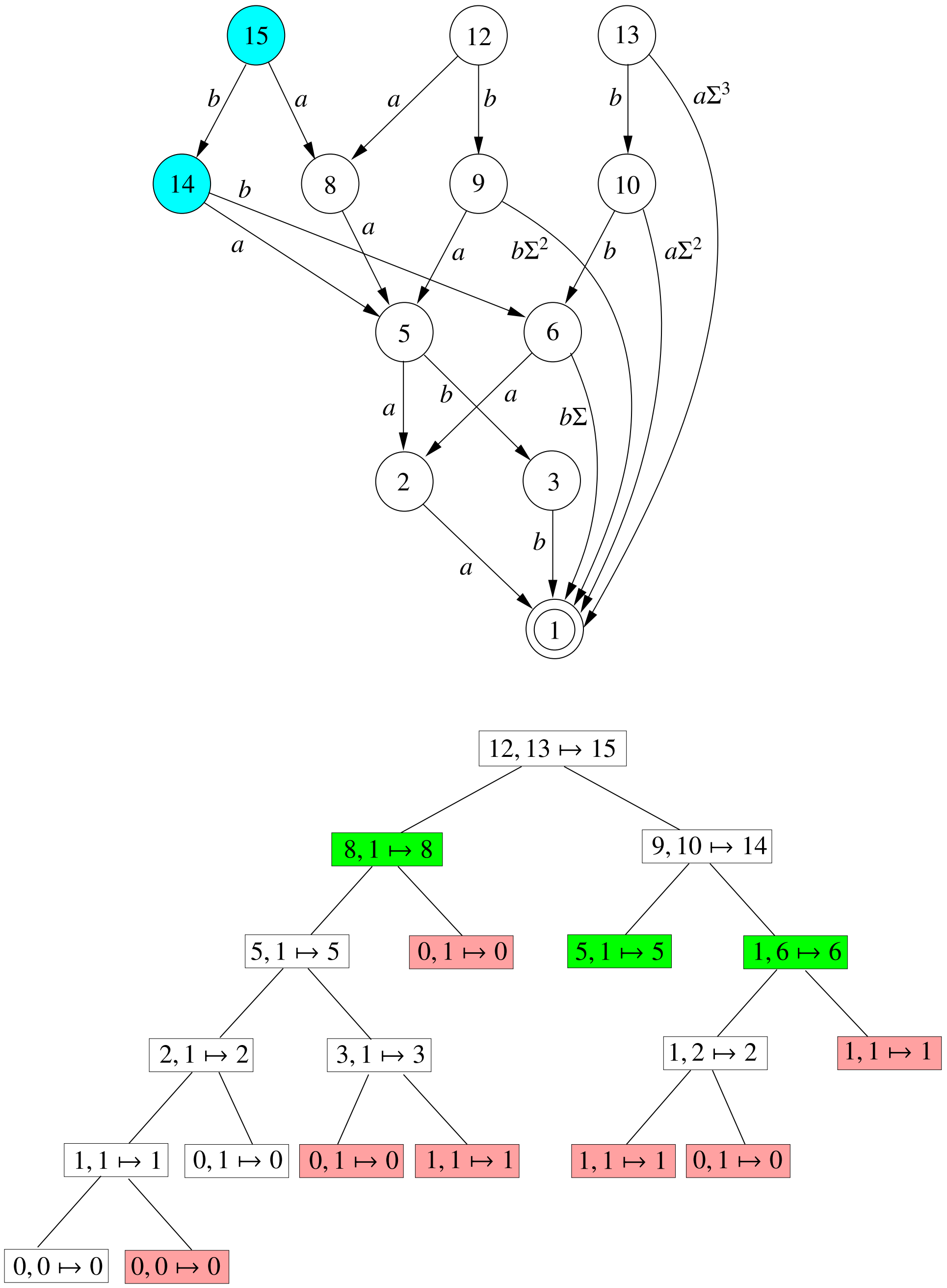


Figure 7.10: An execution of *kinter*.

```
*kinter*($q_1, q_2$)
**Input:** states $q_1, q_2$ recognizing $\langle L_1 \rangle, \langle L_2 \rangle$
**Output:** state recognizing $\langle L_1 \cap L_2 \rangle$
if $G(q_1, q_2)$ is not empty then return $G(q_1, q_2)$
if $q_1 = q_\emptyset$ or $q_2 = q_\emptyset$ then return $q_\emptyset$
if $q_1 \neq q_\emptyset$ and $q_2 \neq q_\emptyset$ then
  if $l_1 < l_2$ /* lengths of the kernodes for $q_1, q_2$ */ then
    for all $i = 1, \ldots, m$ do $r_i \leftarrow kinter(q_1, q_2^{a_i})$
    $G(q_1, q_2) \leftarrow$ kmake($l_2, r_1, \ldots, r_m$)
  else if $l_1 \quad l_2$ then
    for all $i = 1, \ldots, m$ do $r_i \leftarrow kinter(q_1^{a_i}, q_2)$
    $G(q_1, q_2) \leftarrow$ kmake($l_1, r_1, \ldots, r_m$)
  else /* $l_1 = l_2$ */
    for all $i = 1, \ldots, m$ do $r_i \leftarrow kinter(q_1^{a_i}, q_2^{a_i})$
    $G(q_1, q_2) \leftarrow$ kmake($l_1, r_1, \ldots, r_m$)
return $G(q_1, q_2)$
```

Table 7.8: Algorithm *kinter*

The algorithm can be improved by observing that two further properties hold:

- if $K_1 = \{\epsilon\}$ then $L_1 \cap L_2 = L_1$, and so $K_1 \sqcap K_2 = K_1$, and if $K_2 = \{\epsilon\}$ then $L_1 \cap L_2 = L_2$, and so $K_1 \sqcap K_2 = K_2$.

These properties imply that $kinter(q_\epsilon, q) = q = kinter(q, q_\epsilon)$ 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 7.10 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.

Fixed-length complement. Given the kernel K of a fixed-language L of length n, we wish to compute the master state recognizing $\langle \overline{L}^{\,n} \rangle$, where n is the length of L. The subscript n is only necessary because $\emptyset$ has all possible lengths, and so $\overline{\emptyset}^{\,n} = \Sigma^n \neq \Sigma^m = \overline{L}^{\,m}$ for $n \neq m$. Now we have $\langle \overline{\emptyset}^{\,n} \rangle = \{\epsilon\}$ for every $n \geq 0$, and so the subscript is not needed anymore. We define the operator $\widehat{\ }$ on kernels by $\widehat{K} = \langle \overline{L} \rangle$.[3] We obtain the state for $\widehat{K}$ by recursively computing states for $\langle \widehat{K^a} \rangle$ by means of the properties

- if $K = \emptyset$ then $\widehat{K} = \{\epsilon\}$, and if $K = \{\epsilon\}$, then $\widehat{K} = \emptyset$;
- if $\emptyset \neq K \neq \{\epsilon\}$ then $\langle \widehat{K^a} \rangle = \widehat{K^a}$;

which lead to the algorithm of Table 7.9.

[3] The operator is well defined because $\langle L \rangle = \langle L' \rangle$ implies $\langle \overline{L} \rangle = \langle \overline{L'} \rangle$.

```
$kcomp(q)$
Input: state $q$ recognizing a kernel $K$
Output: state recognizing $\widehat{K}$
if $G(q)$ is not empty then return $G(q)$
if $q = q_\emptyset$ then return $q_\epsilon$
else if $q = q_\epsilon$ then return $q_\emptyset$
else
    for all $i = 1, \ldots, m$ do $r_i \leftarrow kcomp(q^{a_i})$
    $G(q) \leftarrow \mathtt{kmake}(r_1, \ldots, r_m)$
    return $G(q)$
```

Table 7.9: Algorithm *kcomp*

Determinization and Minimization.

The algorithm *kdet&min* that converts a NFA recognizing a fixed-language L into the minimal DD recognizing $\langle L \rangle$ differs from *det&min* essentially in one letter: it uses *kmake* instead of *make*. It is shown in Table 7.10.

```
$kdet\&min(A)$
Input: NFA $A = (Q, \Sigma, \delta, Q_0, F)$
Output: state of a multi-DFA recognizing $L(A)$
    return $state[A](Q_0)$

$kstate(S)$
Input: set $S$ of states of length $l$
Output: state recognizing $L(R)$
    if $G(S)$ is not empty then return $G(S)$
    else if $S = \emptyset$ then return $q_\emptyset$
    else if $S \cap F \neq \emptyset$ then return $q_\epsilon$
    else  $/* S \neq \emptyset$ and $S \cap F = \emptyset */$
        for all $i = 1, \ldots, m$ do $S_i \leftarrow \delta(S, a_i)$
        $G(S) \leftarrow kmake(l, kstate(S_1), \ldots, kstate(S_m))$;
        return $G(S)$
```

Table 7.10: The algorithm *kdet&min*(A).

Example 7.18 Figure 7.11 shows again the NFA of Figure 7.6, and the minimal DD 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*$(\{\alpha\})$. □

Exercises

Exercise 83 Prove that the minimal DFA for a language of length 4 over a two-letter alphabet has at most 12 states, and give a language for which the minimal DFA has exactly 12 states.

Exercise 84 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 85 The algorithm for fixed-length universality in Table 7.3 has a best-case runtime equal to the length of the input state q. Give an improved algorithm that only needs $\mathcal{O}(|\Sigma|)$ time for inputs q such that $L(q)$ is not fixed-size universal.

Exercise 86 Let $\Sigma = \{0, 1\}$. Consider the boolean function $f\colon \Sigma^6 \to \Sigma$ defined by

$$f(x_1, x_2, \ldots, x_6) = (x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee (x_5 \wedge x_6)$$

(a) Construct the minimal DFA recognizing $\{x_1 \cdots x_6 \in \Sigma^6 \mid f(x_1, \ldots, x_6) = 1\}$.
(For instance, the DFA accepts 111000 because $f(1, 1, 1, 0, 0, 0) = 1$, but not 101010, because $f(1, 0, 1, 0, 1, 0) = 0$.)

(b) Show that the minimal DFA recognizing $\{x_1x_3x_5x_2x_4x_6 \mid f(x_1, \ldots x_6) = 1\}$ has at least 15 states.
(Notice the different order! Now the DFA accepts neither 111000, because $f(1, 0, 1, 0, 1, 0) = 0$, nor 101010, because $f(1, 0, 0, 1, 1, 0) = 0$.)

(c) More generally, consider the function

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

and the languages $\{x_1x_2 \ldots x_{2n-1}x_{2n} \mid f(x_1, \ldots, x_{2n}) = 1\}$ and $\{x_1x_3 \ldots x_{2n-1}x_2x_4 \ldots x_{2n} \mid f(x_1, \ldots, x_{2n}) = 1\}$.
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 87 Let val : $\{0, 1\}^* \to \mathbb{N}$ be such that val(w) is the number represented by w with the "least significant bit first" encoding.

1. Give a transducer that doubles numbers, i.e. a transducer accepting

$$L_1 = \{[x, y] \in (\{0, 1\} \times \{0, 1\})^* : \text{val}(y) = 2 \cdot \text{val}(x)\}.$$

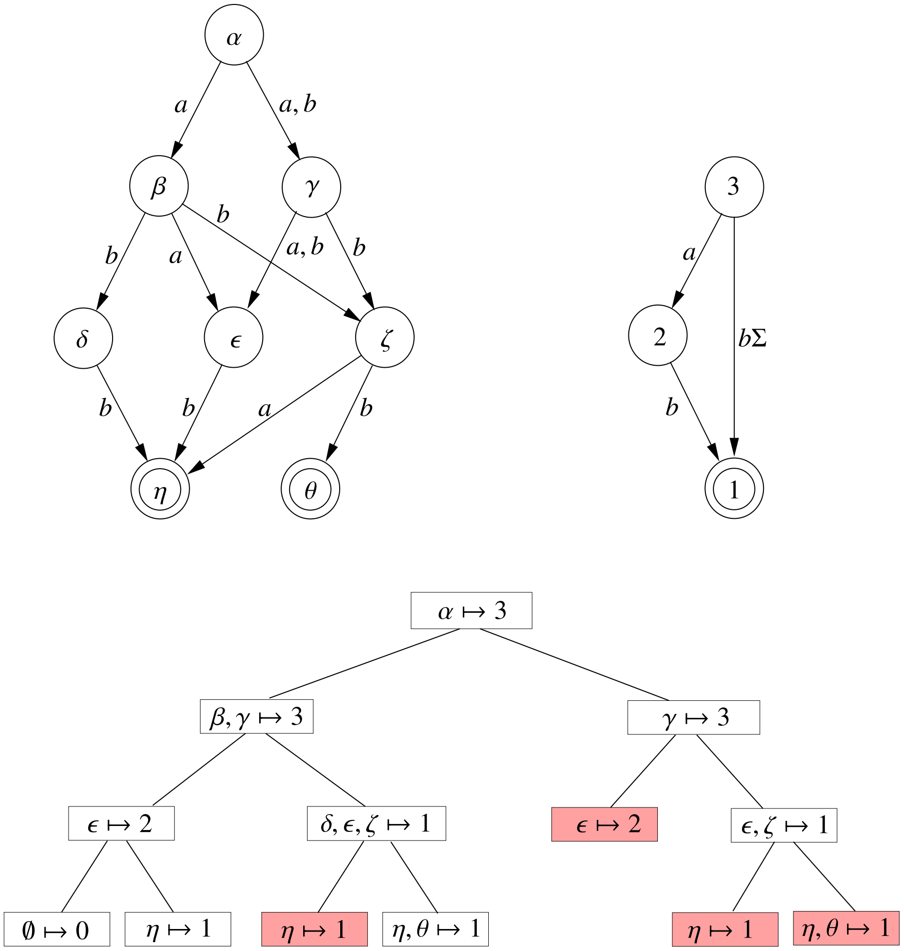

α
a
a, b
β
γ
b
b
a
a, b
b
δ
ϵ
ζ
b
b
a
b
η
θ
3
a
2
bΣ
b
1
α ↦ 3
β, γ ↦ 3
γ ↦ 3
ϵ ↦ 2
δ, ϵ, ζ ↦ 1
ϵ ↦ 2
ϵ, ζ ↦ 1
∅ ↦ 0
η ↦ 1
η ↦ 1
η, θ ↦ 1
η ↦ 1
η, θ ↦ 1

Figure 7.11:

2. Give an algorithm that takes $k \in \mathbb{N}$ as input, and that produces a transducer A_k accepting

$$L_k = \left\{[x, y] \in (\{0, 1\} \times \{0, 1\})^* : \text{val}(y) = 2^k \cdot \text{val}(x)\right\}.$$

(Hint: use (a) and consider operations seen in class.)

3. Give a transducer for the addition of two numbers, i.e. a transducer accepting

$$\{[x, y, z] \in (\{0, 1\} \times \{0, 1\} \times \{0, 1\})^* : \text{val}(z) = \text{val}(x) + \text{val}(y)\}.$$

4. For every $k \in \mathbb{N}_{>0}$, let

$$X_k = \{[x, y] \in (\{0, 1\} \times \{0, 1\})^* : \text{val}(y) = k \cdot \text{val}(x)\}.$$

Suppose you are given transducers A and B accepting respectively X_a and X_b for some $a, b \in \mathbb{N}_{>0}$. Sketch an algorithm that builds a transducer C accepting X_{a+b}. (Hint: use (b) and (c).)

5. Let $k \in \mathbb{N}_{>0}$. Using (b) and (d), how can you build a transducer accepting X_k?

6. Show that the following language has infinitely many residuals, and hence that it is not regular:

$$\left\{[x, y] \in (\{0, 1\} \times \{0, 1\})^* : \text{val}(y) = \text{val}(x)^2\right\}.$$

Exercise 88 Let $L_1 = \{abb, bba, bbb\}$ and $L_2 = \{aba, bbb\}$.

1. Suppose you are given a fixed-length language L described explicitely by a set instead of an automaton. Give an algorithm that ouputs the state q of the master automaton for L.

2. Use the previous algorithm to build the states of the master automaton for L_1 and L_2.

3. Compute the state of the master automaton representing $L_1 \cup L_2$.

4. Identify the kernels $\langle L_1 \rangle$, $\langle L_2 \rangle$, and $\langle L_1 \cup L_2 \rangle$.

Exercise 89 1. Give an algorithm to compute $L(p) \cdot L(q)$ given states p and q of the master automaton.

2. Give an algorithm to compute both the length and size of $L(q)$ given a state q of the master automaton.

3. The length and size of $L(q)$ could be obtained in constant time if they were simply stored in the master automaton table. Give a new implementation of `make` for this representation.

Exercise 90 Let $k \in \mathbb{N}_{>0}$. Let flip : $\{0, 1\}^k \to \{0, 1\}^k$ be the function that inverts the bits of its input, e.g. flip(010) = 101. Let val : $\{0, 1\}^k \to \mathbb{N}$ be such that val(w) is the number represented by w with the "least significant bit first" encoding.

1. Describe the minimal transducer that accepts

$$L_k = \left\{ [x, y] \in (\{0,1\} \times \{0,1\})^k : \mathrm{val}(y) = \mathrm{val}(\mathrm{flip}(x)) + 1 \bmod 2^k \right\} .$$

2. Build the state r of the master transducer for L_3, and the state q of the master automaton for $\{010, 110\}$.

3. Adapt the algorithm *pre* seen in class to compute *post*(r, q).

Exercise 91 Given a boolean formula over variables $x_1, \ldots, x_n$, we define the *language of* ϕ, denoted by $L(\phi)$, as follows:

$$L(\phi) = \{a_1 a_2 \cdots a_n \mid \text{ the assignment } x_1 \mapsto a_1, \ldots, x_n \mapsto a_n \text{ satisfies } \phi\}$$

(a) Give a polynomial algorithm that takes a DFA A recognizing a language of length n as input, and returns a boolean formula ϕ such that $L(\phi) = L(A)$.

(b) Give an exponential algorithm that takes a boolean formula ϕ as input, and returns a DFA A recognizing $L(\phi)$.

Exercise 92 Recall the definition of language of a boolean formula over variables $x_1, \ldots, x_n$ given in Exercise 91. Prove that the following problem is NP-hard:

Given: A boolean formula ϕ in conjunctive normal form, a number $k \geq 1$.
Decide: Does the minimal DFA for $L(\phi)$ have at most 1 state?

Exercise 93 Given $X \subset \{0, 1, \ldots, 2^k - 1\}$, where $k \geq 1$, let A_X be the minimal DFA recognizing the *LSBF*-encodings of length k of the elements of X.

(1) 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.

(2) Let $A_X = (Q, \{0,1\}, \delta, q_0, F)$. Which is the set of numbers recognized by the automaton $A' = (Q, \{0,1\}, \delta', q_0, F)$, where $\delta'(q, b) = \delta(q, 1-b)$?

Exercise 94 Recall the definition of DFAs with negative transitions (DFA-nt's) introduced in Exercise 37, and consider the alphabet $\{0, 1\}$. Show that if only transitions labeled by 1 can be negative, then every regular language over $\{0, 1\}$ has a *unique* minimal DFA-nt.

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     $x \leftarrow 0$
4   $y \leftarrow 1 - x$
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 automaton* 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. There are two possible constructions, both very simple.

- In the first construction, the states are the reachable configurations of the program plus a new state i, which is also the initial state. All states are final. For every transition $c \to c'$ of the graph there is a transition $c \xrightarrow{c'} c'$ in the system automaton. Moreover, there is a transition $i \xrightarrow{c} c$ for every initial configuration.

  It is easy to see that this construction produces a minimal deterministic automaton. Since the label of a transition is also its target state, for any two transitions $c \xrightarrow{c'} c_1$ and $c \xrightarrow{c'} c_2$ we necesarily have $c_1 = c' = c_2$, and so the automaton is deterministic. To show that it is minimal, observe that all words accepted from state c start with c, and so the languages accepted by different states are also different (in fact, they are even disjoint).

- In the second construction, the states are the reachable configurations of the program plus a new state f. The initial states are all the initial configurations, and all states are final. For every transition $c \to c'$ of the graph there is a transition $c \xrightarrow{c} c'$ in the system automaton. Moreover, there is a transition $c \xrightarrow{c} f$ for every configuration c having no successor.

Example 8.2 Figure 8.1 shows the configuration graph of the program of Example 8.1, and the system automata produced by the two constructions above. We wish to algorithmically decide 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 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 automaton* at the top of Figure 8.2. Its intersection with the system automaton in the middle of Figure 8.1 (we could also use the one at the bottom) 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 DFA. 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 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

$$[\ell, x, y]^*\ \left([4, x, 0]\ [1, x, 1]\ +\ [4, x, 1]\ [1, x, 0] \right)\ [\ell, x, y]^*\ .$$

A property automaton for this expression can be easily constructed, and its intersection with the system automaton is again empty. So the property holds, and the assignment is indeed redundant. □

8.2 Programs as Networks of Automata

We can also model the program of Example 8.1 as a *network of communicating automata*. The key idea is to model the two variables x and y and the control flow of the program as three independent processes. The processes for x and y maintain their current value, and control flow process maintains the current value of the program counter. The execution of, say, the assignment $x \leftarrow 0$ in line 3 of the program is modeled as the execution of a joint action between the control flow process

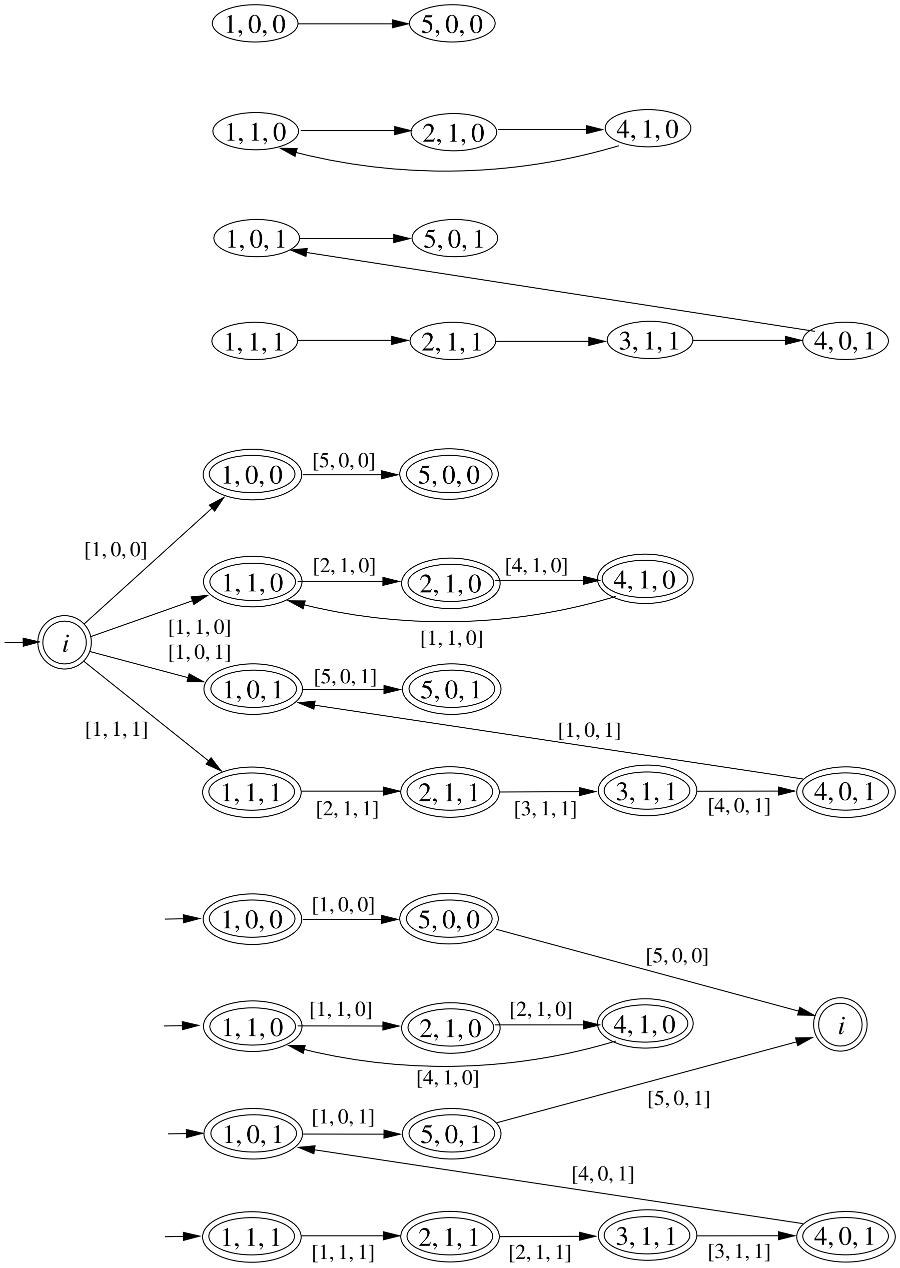


Figure 8.1: Configuration graph and system automata of the program of Example 8.1

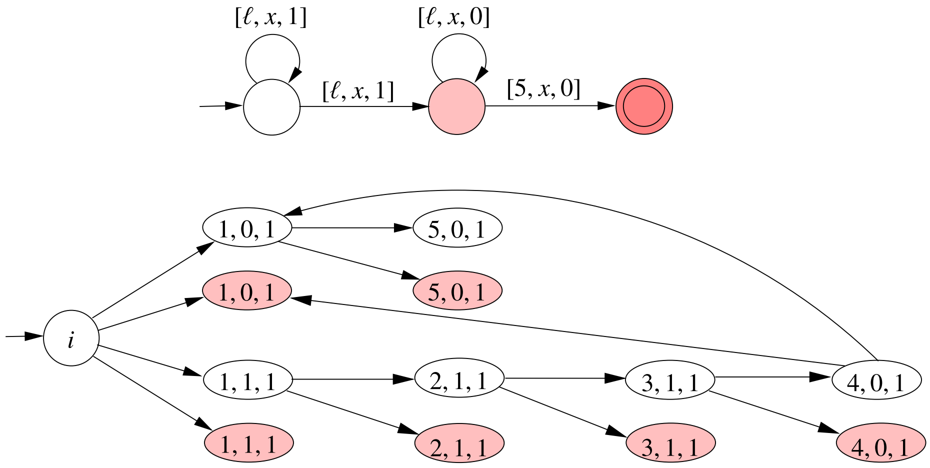


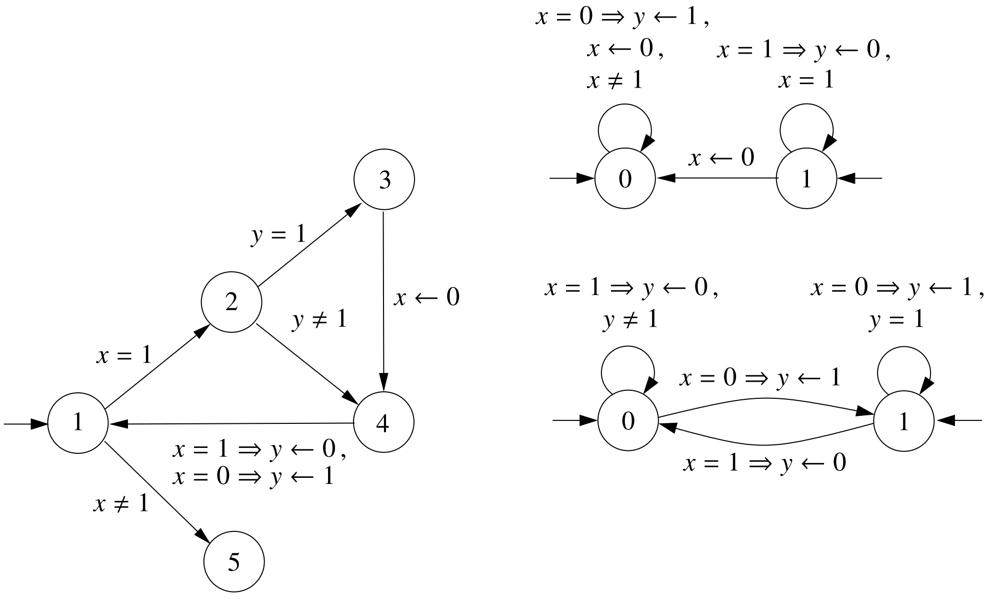
Figure 8.2: Property automaton and product automaton

and the process for variable x: the control flow process updates the current control position to 4, and simultaneously the process for x updates the current value of x to 0.

The processes for variables and control-flow are represented by finite automata where all states are final. The three automata for the program of Example 8.1 are shown in Figure 8.3. Since all states are final, we do not use the graphical representation with a double circle. The automata for x and y have two states, one for each for possible value.The control-flow automaton has 5 states, one for each control location. The alphabet of the automata for x and y correspond to the assignments or boolean conditions of the program that involve x or y, respectively. However, one single assignment may produce several alphabet letters. For instance, the assignment $y \leftarrow 1 - x$ at line 4 produces two alphabet letters, corresponding to two possible actions: if the automaton for x is currently at state 0 (that is, if x currently has value 0), then the automaton for y must move to state 1, otherwise to state 0. (The same occurs with the assignment $x \leftarrow 1 - x$.) We denote these two alphabet letters as $x = 0 \Rightarrow y \leftarrow 1$ and $x = 1 \Rightarrow y \leftarrow 0$. Observe also that the execution of $y \leftarrow 1 - x$ is modeled as a joint action of all three automata: intuitively, the action $x = 0 \Rightarrow y \leftarrow 1$ can be jointly executed only if the automaton for x is currently at state 0 and the control-flow automaton is currently at state 4.

We now give a formal definition of a network of automata. In the definition we do not require all states to be final, because, as we shall see later, a more general definition proves to be useful.

Definition 8.4 *A* network of automata *is a tuple $\mathcal{A} = \langle A_1, \ldots, A_n \rangle$ of NFAs (not necessarily over the same alphabet). Let $A_i = (Q_i, \Sigma_i, \delta_i, Q_{0i}, F_i)$ for every $i = 1, \ldots, n$. A letter of $\Sigma = \Sigma_1 \cup \cdots \cup \Sigma_n$ is called an* action. *A* configuration *of $\mathcal{A}$ is a tuple $[q_1, \ldots, q_n]$ of states, where $q_i \in Q_i$ for every $i \in \{1, \ldots, n\}$. A configuration is* initial *if $q_i \in Q_{0i}$ for every $i \in \{1, \ldots, n\}$, and* final *if $q_i \in F_i$ for every $i \in \{1, \ldots, n\}$.*



$x = 1$
$y = 1$
$y \neq 1$
$x \leftarrow 0$
$x = 1 \Rightarrow y \leftarrow 0$,
$x = 0 \Rightarrow y \leftarrow 1$
$x \neq 1$
1
2
3
4
5
$x = 0 \Rightarrow y \leftarrow 1$,
$x \leftarrow 0$,
$x \neq 1$
$x = 1 \Rightarrow y \leftarrow 0$,
$x = 1$
$x \leftarrow 0$
0
1
$x = 1 \Rightarrow y \leftarrow 0$,
$y \neq 1$
$x = 0 \Rightarrow y \leftarrow 1$,
$y = 1$
$x = 0 \Rightarrow y \leftarrow 1$
$x = 1 \Rightarrow y \leftarrow 0$
0
1

Figure 8.3: A network of three automata modeling the program of Example 8.1. All states are final, and so the double circles are drawn as simple circles for clarity.

Observe that each NFA of a network has its own alphabet Σ_i. The alphabets $\Sigma_1, \ldots, \Sigma_n$ are not necessarily pairwise disjoint, in fact usually they are not. We define when is an action enabled at a configuration, and what happens when it occurs.

Definition 8.5 *Let $\mathcal{A} = \langle A_1, \ldots, A_n \rangle$ be a network of automata, where $A_i = (Q_i, \Sigma_i, \delta_i, Q_{0i}, F_i)$. Given an action a, we say that A_i* participates *in a if $a \in \Sigma_i$. An action a is* enabled *at a configuration $[q_1, \ldots, q_n]$ if $\delta_i(q_i, a) \neq \emptyset$ for every $i \in \{1, \ldots, n\}$ such that A_i participates in a. If a is enabled, then it can* occur*, and its occurrence can lead to any element of the cartesian product $Q'_1 \times \cdots \times Q'_n$, where*

$$Q'_i = \begin{cases} \delta(q_i, a) & \text{if } A_i \text{ participates in } a \\ \{q_i\} & \text{otherwise} \end{cases}$$

We call $Q'_1 \times \cdots \times Q'_n$ the set of successor configurations *of $[q_1, \ldots, q_n]$ with respect to action a. We write $[q_1, \ldots, q_n] \xrightarrow{a} [q''_1, \ldots, q''_n]$ to denote that $[q''_1, \ldots, q''_n]$ belongs to this set.*

The notion of language accepted by a network of automata is defined in the standard way:

Definition 8.6 *A* run *of $\mathcal{A}$ on input $a_0 a_1 \ldots a_{n-1}$ is a sequence $c_0 \xrightarrow{a_0} c_1 \xrightarrow{a_1} c_2 \ldots \xrightarrow{a_{n-1}} c_n$ such that c_i is a configuration for every $0 \leq i \leq n$, the configuration c_0 is initial, and $\delta(q_i, a_i) = q_{i+1}$ for every $0 \leq i \leq n - 1$. A run is* accepting *if c_n is a final configuration. $\mathcal{A}$* accepts *$w \in \Sigma^*$ if it has an* *accepting run on input w. The* language recognized *by $\mathcal{A}$, denoted by $L(\mathcal{A})$ is the set of words of accepted by $\mathcal{A}$.*

Example 8.7 Let A_x, A_y, and A_P be the three automata of Example 8.1 for the variables x and y and the control, respectively. We have

$$\begin{array}{rcl} \Sigma_x & = & \{x = 1\,,\ x \neq 1\,,\ x \leftarrow 0\,,\ (x = 0 \Rightarrow y \leftarrow 1)\,,\ (x = 1 \Rightarrow y \leftarrow 0)\,\} \\ \Sigma_y & = & \{y = 1\,,\ y \neq 1\,,\ (x = 0 \Rightarrow y \leftarrow 1)\,,\ (x = 1 \Rightarrow y \leftarrow 0)\,\} \\ \Sigma_P & = & \Sigma_x \cup \Sigma_y \end{array}$$

The automata participating in, say, the action $x = 0$ are A_P and A_x, and all three automata participate in $(x = 1 \Rightarrow y \leftarrow 0)$. Observe that A_P participates in all actions. If we define $\mathcal{A} = \langle A_P, A_x, A_y\rangle$, then the configurations of $\mathcal{A}$ are the configurations of the program of Example 8.1. The configuration $[3, 1, 0]$ enables the action $x \leftarrow 0$, and we have $[3, 1, 0] \xrightarrow{x\leftarrow 0} [4, 0, 0]$. One of the runs of $\mathcal{A}$ is

$$[1, 1, 1] \xrightarrow{x=1} [2, 1, 1] \xrightarrow{y=1} [3, 1, 1] \xrightarrow{x\leftarrow 0} [1, 0, 1] \xrightarrow{x\neq 1} [5, 0, 1]$$

and so the word $(x = 1)\ (y = 1)\ (x \leftarrow 0)\ (x \neq 1)$ belongs to $L(\mathcal{A})$. □

8.2.1 Parallel Composition

The language of a network of automata admits a useful characterization. Given languages $L_1 \subseteq \Sigma_1^*, \ldots, L_n \subseteq \Sigma_n^*$, the *parallel composition* of $L_1, \ldots, L_n$ is the language $L_1 \parallel L_2 \parallel \cdots \parallel L_n \subseteq (\Sigma_1 \cup \cdots \cup \Sigma_n)^*$ defined as follows: $w \in L_1 \parallel \cdots \parallel L_n$ iff $proj_{\Sigma_i}(w) \in L_i$ for every $1 \leq i \leq n$.

Notice that, strictly speaking, parallel composition is an operation that depends not only on the languages $L_1, \ldots, L_n$, but also on their alphabets. Take for example $L_1 = \{a\}$ and $L_2 = \{ab\}$. If we look at them as languages over the alphabet $\{a, b\}$, then $L_1 \parallel L_2 = \emptyset$; if we look at L_1 as a language over $\{a\}$, and L_2 as a languge over $\{a, b\}$, then $L_1 \parallel L_2 = \{ab\}$. So the correct notation would be $L_1 \parallel_{\Sigma_1,\Sigma_2} L_2$, but we abuse language, and assume that when a language is defined we specify its alphabet.

Proposition 8.8 *(1) Parallel composition is associative, commutative, and idempotent. That is: $(L_1 \parallel L_2) \parallel L_3 = L_1 \parallel (L_2 \parallel L_3)$ (associativity); $L_1 \parallel L_2 = L_2 \parallel L_1$ (commutativity), and $L \parallel L = L$ (idempotence).*

(2) If $L_1, L_2 \subseteq \Sigma^*$, then $L_1 \parallel L_2 = L_1 \cap L_2$.

(3) Let $\mathcal{A} = \langle A_1, \ldots, \mathcal{A}_n\rangle$ be a network of automata. Then $L(\mathcal{A}) = L(A_1) \parallel \cdots \parallel L(A_n)$.

Proof: See Exercise 97. □

Combining (2) and (3) we obtain that two automata A_1, A_2 over the same alphabet satisfy $L(A_1 \otimes A_2) = L(A_1) \cap L(A_2)$. Intuitively, in this case every step must be jointly executed by A_1 and A_2, or, in other words, the machines move in lockstep. At the other extreme, if the input alphabets are pairwise disjoint, then, intuitively, the automata do not communicate at all, and move independently of each other.

8.2.2 Asynchonous Product

Given a network of automata $\mathcal{A} = \langle A_1, \ldots A_n \rangle$, we can compute a NFA recognizing the same language. We call it the *asynchronous product* of $\mathcal{A}$, and denote itby $A_1 \otimes \cdots \otimes A_n$. The NFA is computed by algorithm *AsyncProduct* in Table 8.1. The algorithm follows easily from Definitions 8.5 and 8.6. Starting at the initial configurations, the algorithm repeatedly picks a configuration from the workset, stores it, constructs its successors, and adds them (if not yet stored) to the workset. Line 10 is the most important one. Assume we are in the midlle of the execution of *AsyncProduct*(A_1, A_2), currently processing a configuration $[q_1, q_2]$ and an action a at line 8.

- Assume that a belongs to $\Sigma_1 \cap \Sigma_2$, and the a-transitions leaving q_1 and q_2 are $q_1 \xrightarrow{a} q_1', q_1 \xrightarrow{a} q_1''$ and $q_2 \xrightarrow{a} q_2', q_1 \xrightarrow{a} q_2''$. Then we obtain $Q_1' = \{q_1', q_1''\}$ and $Q_2' = \{q_2', q_2''\}$, and the loop at lines 11-13 adds four transitions: $[q_1, q_2] \xrightarrow{a} [q_1', q_2']$, $[q_1, q_2] \xrightarrow{a} [q_1'', q_2']$, $[q_1, q_2] \xrightarrow{a} [q_1', q_2'']$, and $[q_1, q_2] \xrightarrow{a} [q_1'', q_2'']$, which correspond to the four possible "joint a-moves" that A_1 and A_2 can execute from $[q_1, q_2]$.

- Assume now that a only belongs to Σ_1, the a-transitions leaving q_1 are as before, and, since $a \notin \Sigma_2$, there are no a-transitions leaving q_2. Then $Q_1' = \{q_1', q_1''\}$, $Q_2' = \{q_2\}$, and the loop adds transitions $[q_1, q_2] \xrightarrow{a} [q_1', q_2]$, $[q_1, q_2] \xrightarrow{a} [q_1'', q_2]$, which correspond to A_1 making a move while A_2 stays put.

- Assume finally that a belongs to $\Sigma_1 \cap \Sigma_2$, the a-transitions leaving q_1 are as before, and there are no a-transitions leaving q_2 (which is possible even if $a \in \Sigma_2$, because A_2 is a NFA). Then $Q_1' = \{q_1', q_1''\}$, $Q_2' = \emptyset$, and the loop adds no transitions. This corresponds to the fact that, since a-moves must be jointly executed by A_1 and A_2, and A_2 is not currently able to do any a-move, no joint a-move can happen.

Example 8.9 The NFA *AsyncProduct*(A_P, A_x, A_y) is shown in Figure 8.4. Its states are the reachable configurations of the program. Again, since all states are final, we draw all states as simple cycles. □

Finally, observe that we have defined the asynchronous product of $A_1 \otimes \cdots \otimes A_n$ as an automaton over the alphabet $\Sigma = \Sigma_1 \cup \cdots \cup \Sigma_n$, but the algorithm can be easily modified to return a system automaton having the set of configurations as alphabet (see Exercise 96).

```
$AsyncProduct(A_1, \ldots, A_n)$
Input: a network of automata $\mathcal{A} = \langle A_1, \ldots, A_n \rangle$, where
$A_i = (Q_i, \Sigma_i, \delta_i, Q_{0i}, F_i)$ for every $i = 1, \ldots n$.
Output: NFA $A_1 \otimes \cdots \otimes A_n = (Q, \Sigma, \delta, Q_0, F)$ recognizing $L(\mathcal{A})$.

$Q, \delta, F \leftarrow \emptyset$
$Q_0 \leftarrow Q_{01} \times \cdots \times Q_{0n}$
$W \leftarrow Q_0$
while $W \neq \emptyset$ do
    pick $[q_1, \ldots, q_n]$ from $W$
    add $[q_1, \ldots, q_n]$ to $Q$
    if $\bigwedge_{i=1}^{n} q_i \in F_i$ then add $[q_1, \ldots, q_n]$ to $F$
    for all $a \in \Sigma_1 \cup \ldots \cup \Sigma_n$ do
        for all $i \in [1..n]$ do
            if $a \in \Sigma_i$ then $Q'_i \leftarrow \delta_i(q_i, a)$ else $Q'_i = \{q_i\}$
        for all $[q'_1, \ldots, q'_n] \in Q'_1 \times \ldots \times Q'_n$ do
            if $[q'_1, \ldots, q'_n] \notin Q$ then add $[q'_1, \ldots, q'_n]$ to $W$
            add $([q_1, \ldots, q_n], a, [q'_1, \ldots, q'_n])$ to $\delta$
return $(Q, \Sigma, \delta, Q_0, F)$
```

Table 8.1: Asynchronous product of a network of automata.

8.2.3 State- and event-based properties.

Properties of the sequence of configurations visited by the program are often called *state-based properties* (program configurations are often called program states, we use program configuration because of the possible confusion between program states and automaton states). If we we wish to check such properties, we construct a system automaton as shown in Exercise 96.

Properties of the sequence of instructions executed by the program can be directly checked using the asynchronous product. For instance, consider the property: no terminating execution of the program contains an occurrence of the action $(x = 0 \Rightarrow y \leftarrow 1)$. The property can be reformulated as: no execution of program belongs to the regular language

$$\Sigma_P^* \, (x = 0 \Rightarrow y \leftarrow 1) \, \Sigma_P^* \, (x \neq 1)$$

for which we can easily find a property automaton A_φ. We can check the property by checking emptiness of an automaton for the intersection of $AsyncProduct(A_P, A_x, A_y)$ and A_φ. This verification style is often called *event-based verification* (Occurrences of program actions are often called *events*; an execution may contain many events corresponding to the same action).

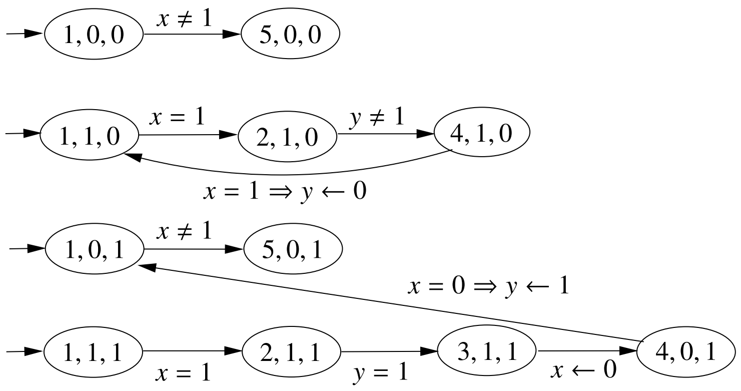


Figure 8.4: Asynchronous product of the automata of Figure 8.3.

8.3 Concurrent Programs

Networks of automata can also elegantly model *concurrent programs*, that is, programs consisting of a number of sequential programs communicating in some way. These sequential programs are often called *processes*. A popular communication mechanism between processes are *shared variables*, where a process can communicate with another by writing a value to a variable, which is then read by the other process. As an example, we consider the Lamport-Burns mutual exclusion algorithm for two processes[2]. It has the following code.

```
          repeat
$nc_0$ :      $b_0 \leftarrow 1$
$t_0$  :      while $b_1 = 1$ do skip
$c_0$  :      $b_0 \leftarrow 0$
          forever
```

```
          repeat
$nc_1$ :      $b_1 \leftarrow 1$
$t_1$  :      if $b_0 = 1$ then
$q_1$  :          $b_1 \leftarrow 0$
$q_1'$ :          while $b_0 = 1$ do skip
                  goto $nc_1$
$c_1$  :      $b_1 \leftarrow 0$
          forever
```

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 at control points c_0 and c_1 (their *critical sections*), and that the two processes will not reach a deadlock. Other properties the algorithm should satisfy are discussed later. 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

[2] L. Lamport: The mutual exclusion problem: part II-statements and solutions. *JACM* 1986

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.

A configuration of this program is a tuple $[n_{b_0}, n_{b_1}, \ell_0, \ell_1]$, where $n_{b_0}, n_{b_1} \in \{0, 1\}$, $\ell_0 \in \{nc_0, t_0, c_0\}$, and $\ell_1 \in \{nc_1, t_1, q_1, q'_1, c_1\}$. We define executions of the program by *interleaving*. We assume that, if at the current configuration both processes can do an action, then the actions will not happen at the same time, one of the two will take place before the other. However, the actions can occur in any order. So, loosely speaking, if two processes can execute two sequences of actions independently of each other (because, say, they involve disjoint sets of variables), then the sequences of actions of the two processes running in parallel are the interleaving of the sequences of the processes.

For example, at the initial configuration $[0, 0, nc_0, nc_1]$ both processes can set their variables to 1. So we assume that there are two transitions $[0, 0, nc_0, nc_1] \to [1, 0, t_0, nc_1]$ and $[0, 0, nc_0, nc_1] \to [0, 1, nc_0, t_1]$. Since the other process can still set its variable, we also have transitions $[1, 0, t_0, nc_1] \to [1, 1, t_0, t_1]$ and $[1, 0, t_0, nc_1] \to [1, 1, t_0, t_1]$

In order to model a shared-variable program as a network of automata we just model each process and each variable by an automaton. The network of automata modelling the Lamport-Burns algorithm is shown in Figure 8.5, and its asynchronous product in Figure 8.6.

8.3.1 Expressing and Checking Properties

We use the Lamport-Burns 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 can be easily checked on-the-fly while constructing the asynchronous product, and a quick inspection of Figure 8.6 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

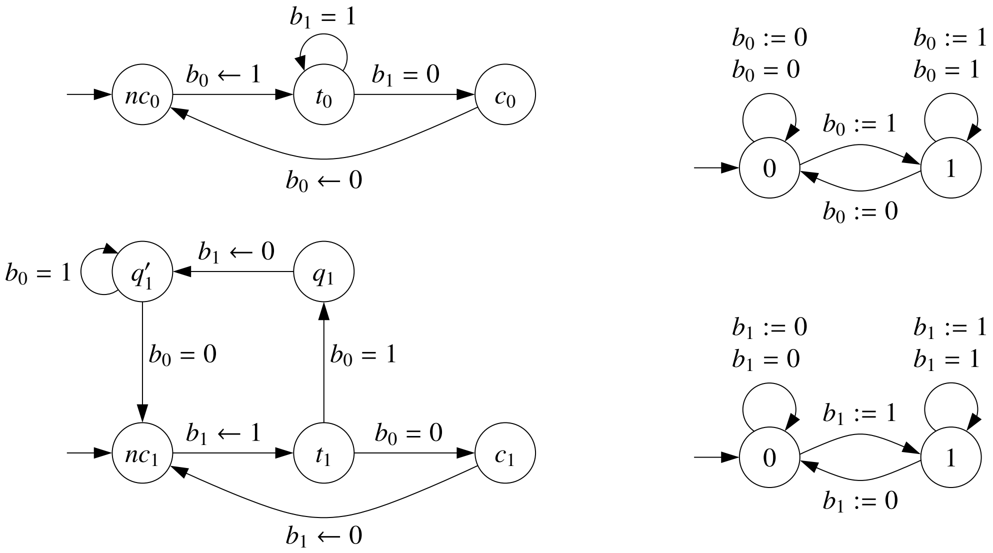


Figure 8.5: A network of four automata modeling the Lamport-Bruns mutex algorithm for two processes. The automata on the left model the control flow of the processes, and the automata on the right the two shared variables. All states are final.

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.

8.4 Coping with the State-Explosion Problem

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.

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.

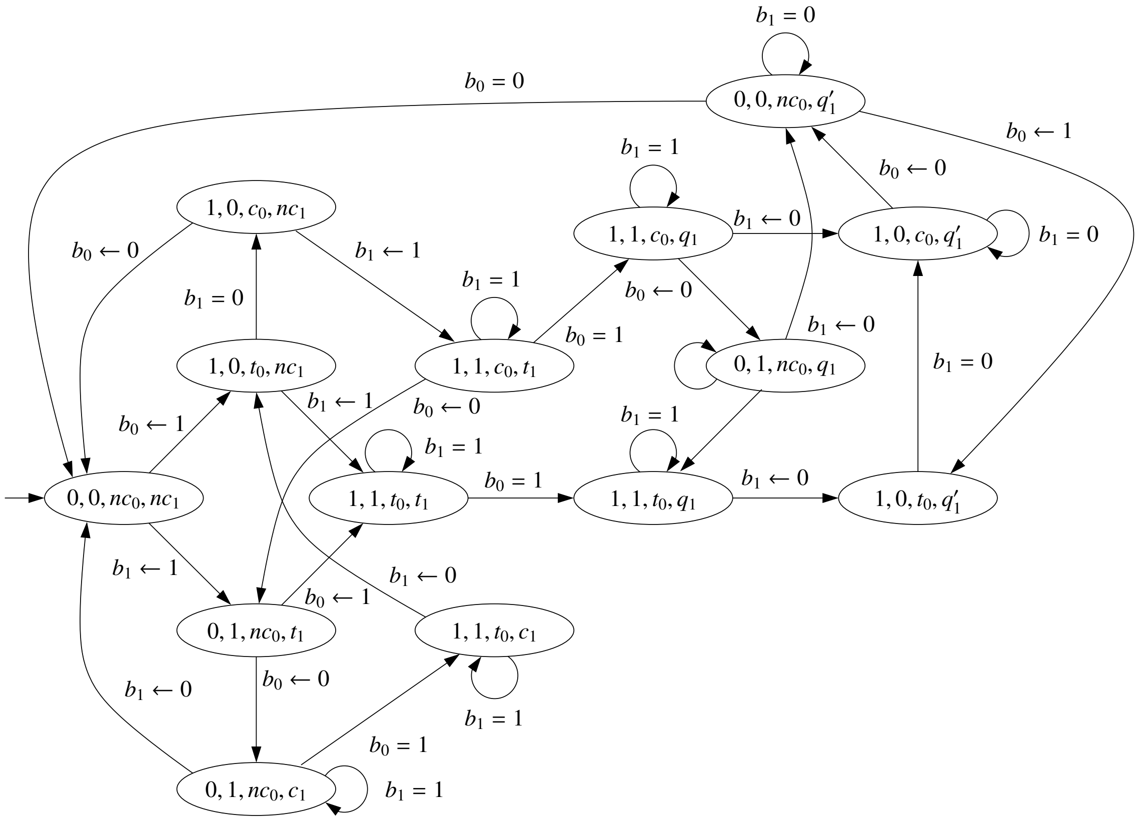


Figure 8.6: Asynchronous product of the network of Figure 8.5.

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 8.10 *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 $L(A_1 \otimes \cdots \otimes A_n \otimes V) \neq \emptyset$.

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. □

8.4.1 On-the-fly verification.

Recall that, given a program with a set E of executions, and given a regular expression describing the set of potential executions V violating a property, we have the following four-step technique to check if the program satisfis the property, i.e., to check if $E \cap V = \emptyset$: (1) transform the regular expression into an NFA A_V (V for "violations") using the algorithm of Section 2.4.1; (2) model the program as a network of automata $\langle A_1, \ldots, A_n \rangle$, and construct the NFA $A_E = \mathit{AsyncProd}(A_1, \ldots, A_n)$ recognizing E; (3) construct an NFA $A_E \cap A_V$ for $E \cap V$ using algorithm *intersNFA*; (4) check emptiness of $A_E \cap A_V$ using algorithm *empty*.

Observe that A_E may have *more* states than $A_E \cap A_V$: if a state of A_E is not reachable by any word of V, then the state does not appear in $A_E \cap A_V$. The difference in size between the NFAs can be considerable, and so it is better to *directly* construct $A_E \cap A_V$, bypassing the construction of A_E. Further, it is inefficient to first construct $A_E \cap A_V$ and then check for emptiness. It is better to check for emptines while constructing $A_E \cap A_V$, interrupting the algorithm the moment it constructs a final state. So, loosely speaking, we look for an algorithm that constructs the intersection with A_V and checks for emptiness *on the fly*, while computing A_E.

This is easily achieved by means of the observation above: 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 $L(A_1 \otimes \cdots \otimes A_{n+1}) = L(A_1 \otimes \cdots \otimes A_n)) \cap L(A_{n+1})$. So, if the alphabet of V is the union of the alphabets of $A_1, \ldots, A_n$, we have $L(\mathcal{A}) \cap L(V) = L(A_1 \otimes \cdots \otimes A_n \otimes A_v)$, and we can check emptiness by means of the algorithm *CheckViol* shown in Table 8.2.

Looking at A_V as just another component of the asynchronous product, as in Algorithm *CheckViol*, also has another small advantage. Consider again the language V

$$\Sigma_P^* \, (x = 0 \Rightarrow y \leftarrow 1) \, \Sigma_P^* \, (x \neq 1)$$

Actually, we are only interested in the subset of actions $\Sigma' = \{(x = 0 \Rightarrow y \leftarrow 1), (x \neq 1)\}$. So we can replace A_V by an automaton A'_V over Σ' recognizing only the sequence $(x = 0 \Rightarrow y \leftarrow 1)\,(x \neq 1)$. That is, this automaton participates in all occurrences of these actions, ignoring the rest. Intuitively, we can think of A'_V as an *observer* of the network $\langle A_1 \otimes \cdots \otimes A_n \rangle$ that is only interested in observing the actions of Σ'.

```
CheckViol($A_1, \ldots, A_n, V$)
Input: a network $\mathcal{A} = \langle A_1, \ldots A_n \rangle$, where $A_i = (Q_i, \Sigma_i, \delta_i, Q_{0i}, F_i)$ for $1 \leq i \leq n$;
       an NFA $V = (Q_V, \Sigma_V, \delta_V, Q_{0v}, F_v)$.
Output: true if $L(A_1 \otimes \cdots \otimes A_n \otimes V)$ is nonempty, false otherwise.

$Q \leftarrow \emptyset$; $Q_0 \leftarrow Q_{01} \times \cdots \times Q_{0n} \times Q_{0v}$
$W \leftarrow Q_0$
while $W \neq \emptyset$ do
   pick $[q_1, \ldots, q_n, q]$ from $W$
   add $[q_1, \ldots, q_n, q]$ to $Q$
   for all $a \in \Sigma_1 \cup \ldots \cup \Sigma_n$ do
      for all $i \in [1..n]$ do
         if $a \in \Sigma_i$ then $Q'_i \leftarrow \delta_i(q_i, a)$ else $Q'_i = \{q_i\}$
      $Q' \leftarrow \delta_V(q, a)$
      for all $[q'_1, \ldots, q'_n, q'] \in Q'_1 \times \ldots \times Q'_n \times Q'$ do
         if $\bigwedge_{i=1}^{n} q'_i \in F_i$ and $q \in F_v$ then return true
         if $[q'_1, \ldots, q'_n, q'] \notin Q$ then add $[q'_1, \ldots, q'_n, q']$ to $W$
return false
```

Table 8.2: Algorithm to check violation of a property.

8.4.2 Compositional Verification

Consider the asynchronous product $A_1 \otimes A_2$ of two NFAs over alphabets Σ_1, Σ_2. Intuitively, A_2 does not observe the actions of $\Sigma_1 \setminus \Sigma_2$: they are "internal" actions of A_1. Therefore, A_1 can be replaced by any other automaton A'_1 satisfying $L(A'_1) = proj_{\Sigma_2}(L(A_1))$ without Σ_2 "noticing", meaning that the sequences of actions that A_2 can execute with A_1 and A'_1 as partners are the same, or, formally,

$$proj_{\Sigma_2}(A_1 \otimes A_2) = proj_{\Sigma_2}(A'_1 \otimes A_2)\ .$$

In particular, we then have $L(A_1 \otimes A_2) \neq \emptyset$ if and only if $L(A'_1 \otimes A_2) \neq \emptyset$, and so checking emptiness of $A_1 \otimes A_2$ can be replaced by checking emptiness of $A'_1 \otimes A_2$. It is easy to construct an automaton recognizing $proj_{\Sigma_2}(L(A_1))$: just replace all transitions of A_1 labeled with letters of $\Sigma_1 \setminus \Sigma_2$ by ε-transitions. This automaton has the same size as A_1, and so subsituting it for A_1 has no immediate advantage. However, after removing the ϵ-transitions, and reducing the resulting NFA, we may obtain an automaton A'_1 smaller than A_1.

This idea can be extended to the problem of checking emptiness of a product $A_1 \otimes \cdots \otimes A_n$ with an arbitrary number of components. Exploting the associativity of $\otimes$, we rewrite the product as $A_1 \otimes (A_2 \otimes \cdots \otimes A_n)$, and replace A_1 by a hopefully smaller automaton A'_1 over the alphabet $\Sigma_2 \cup \cdots \cup \Sigma_n$. In a second step we rewrite $A'_1 \otimes A_2 \otimes A_3 \otimes \cdots \otimes A_n$ as $(A'_1 \otimes A_2) \otimes (A_3 \otimes \cdots \otimes A_n)$, and, applying again the same procedure, replace $A'_1 \otimes A_2$ by a new automaton A'_2 over the alphabet

$\Sigma_3 \cup \cdots \cup \Sigma_n$. The procedure continues until we are left with one single automaton A'_n over Σ_n, whose emptiness can be checked directly on-the-fly.

To see this idea in action, consider the network of automata in The upper part of Figure 8.8. It models a 3-bit counter consisting of an array of three 1-bit counters, where each counter communicates with its neighbours.

We call the components of the network 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 "visible" actions $Vis = \{inc, 0, 1, \ldots, 7\}$. 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).

For example, here is a run of the network, starting an ending at configuration $[0, 0, 0]$.

$$\begin{array}{lllllll}
[0,0,0] & \xrightarrow{inc} & [0,0,1] & & & & \\
 & \xrightarrow{inc} & [0,0,aux_0] & \xrightarrow{inc_1} & [0,1,0] & & \\
 & \xrightarrow{inc} & [0,1,1] & & & & \\
 & \xrightarrow{inc} & [0,1,aux_0] & \xrightarrow{inc_1} & [0,aux_1,0] & \xrightarrow{inc_2} & [1,0,0] \\
 & \xrightarrow{inc} & [1,0,1] & & & & \\
 & \xrightarrow{inc} & [1,0,aux_0] & \xrightarrow{inc_1} & [1,1,0] & & \\
 & \xrightarrow{inc} & [1,1,1] & & & & \\
 & \xrightarrow{inc} & [1,1,aux_0] & \xrightarrow{inc_1} & [1,aux_1,0] & \xrightarrow{inc_2} & [0,0,0]\ldots
\end{array}$$

The bottom part of Figure 8.8 shows the asynchronous product of the network. (All states are final, but we have drawn them as simple instead of double ellipses for simplicity. The product has 18 states.

Assume we wish to check some property whose violations are given by the language of an automaton A_V over the alphabet *Vis* of visible actions. The specific shape of A_V is not relevant for the discussion above. Important is only that, instead of checking emptiness of $A_2 \otimes A_1 \otimes A_0 \otimes A_V$, we can also construct an automaton A'_0 such that $L(A'_0) = proj_{Vis}(L(A_2 \otimes A_1 \otimes A_0))$, and check emptiness of $A'_0 \otimes A_V$. If we compute A'_0 be first computing the asynchronous product $A_2 \otimes A_1 \otimes A_0$, and then removing invisible actions and reducing, then the maximum size of all intermediate automata involved is at least 18.

Let us now apply the procedure above, starting with A_2. Since $\Sigma_2 \subseteq (\Sigma_1 \cap \Sigma_0)$, the automaton A_2 cannot execute any moves hidden from A_1 and A_0, and so $A'_2 = A_2$. In the next step we compute

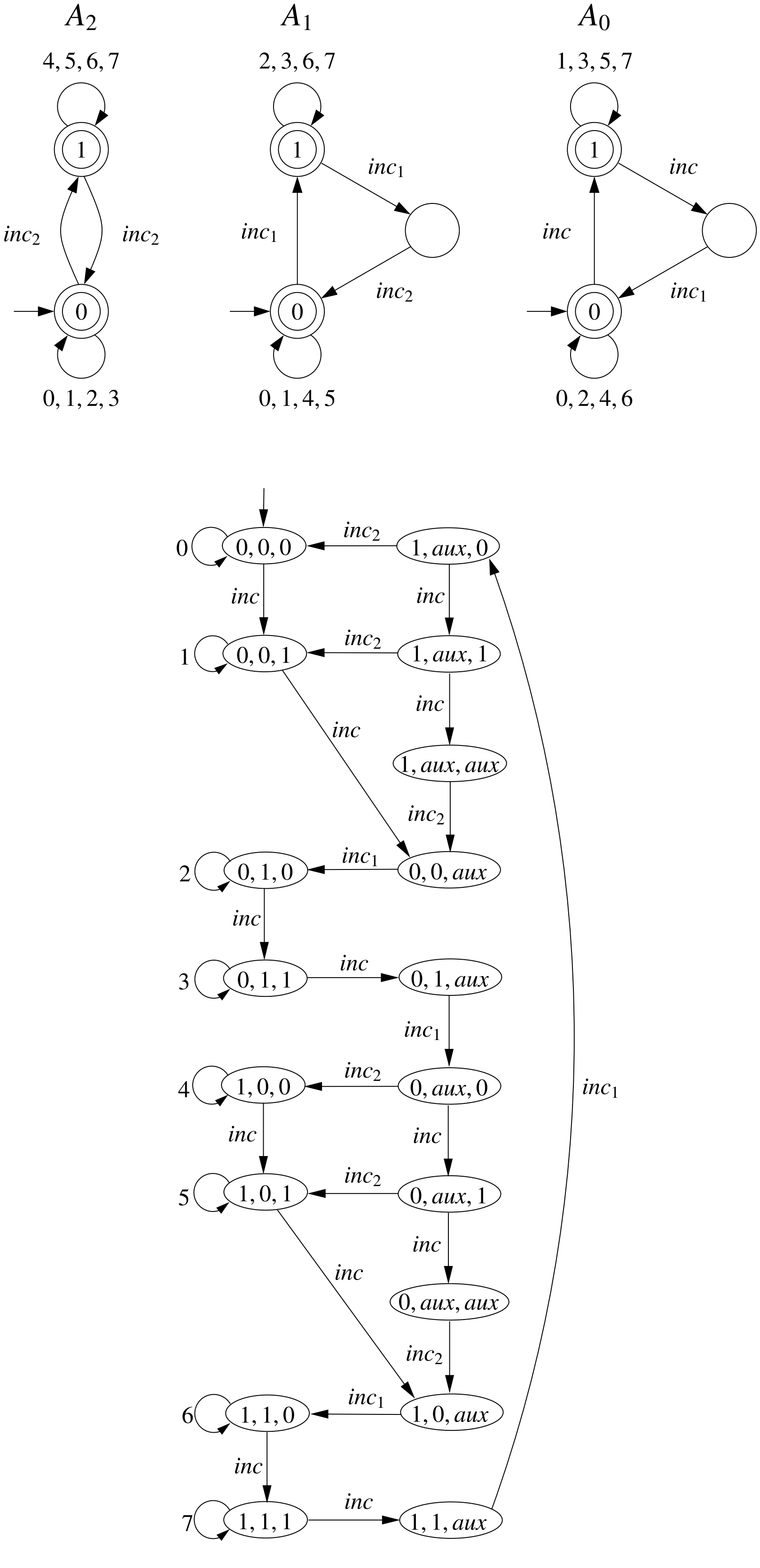


Figure 8.7: A network modeling a 3-bit counter and its asynchronous product.

the product $A_2 \otimes A_1$ shown in Figure 8.8, on the left. Now inc_2 is an internal action of $A_1 \otimes A_2$, because it belongs neither to the alphabet of A_0 nor to the alphabet of A_V. The result of eliminating ε-transitions and reducing is shown on the right of the figure.

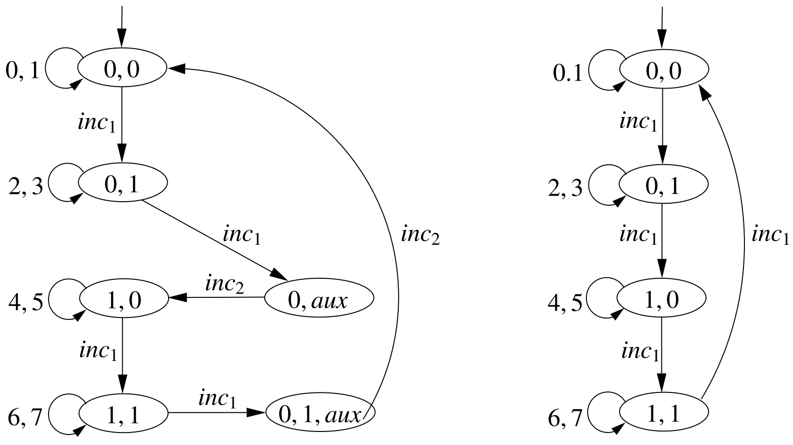


Figure 8.8: The asynchronous product $A_2 \otimes A_1$, and the reduced automaton A'_1.

In the next step we construct $A'_1 \otimes A_0$, with inc_1 as new internal action, not present in the alphabet of A_V, and reduce the automaton again. The results are shown in Figure 8.9. The important fact is that we have never had to construct an automaton with more than 12 states, a saving of six states with respect to the method that directly computes $A_2 \otimes A_1 \otimes A_0$. While saving six states is of course irrelevant in practice, in larger examples the savings can be significant.

8.4.3 Symbolic State-space Exploration

Figure 8.10 shows again the program of Example 8.1, and its flowgraph. An edge of the flowgraph 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 is 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

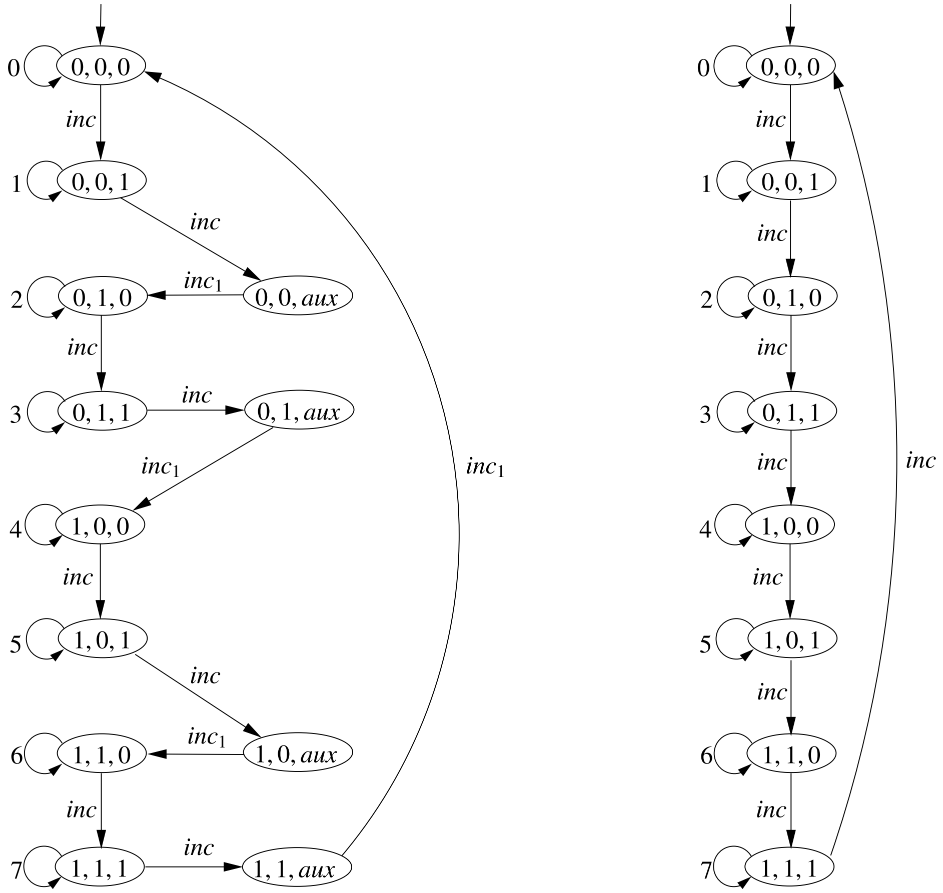


Figure 8.9: The asynchronous product $A'_1 \otimes A_0$, and the reduced automaton A'_0.

```
1  while $x = 1$ do
2     if $y = 1$ then
3        $x \leftarrow 0$
4     $y \leftarrow 1 - x$
5  end
```

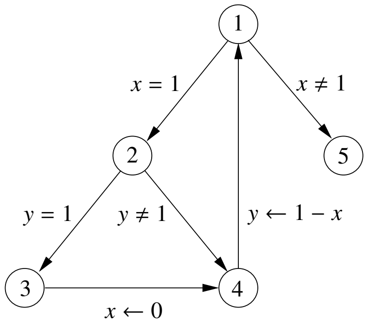


Figure 8.10: Flowgraph of the program of Example 8.1

$$S = \bigcup_{a,b \in C} S_{a,b}$$

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 $I$
$OldP \leftarrow \emptyset$; $P \leftarrow I$
while $P \neq OldP$ do
   $OldP \leftarrow P$
   $P \leftarrow$ Union($P$, Post($P, S$))
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 6, or, if they are fixed-length, by the algorithms of Chapter 7.

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 8.10. The fixed-length transducer for the step relation S is shown in Figure 8.11; 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,

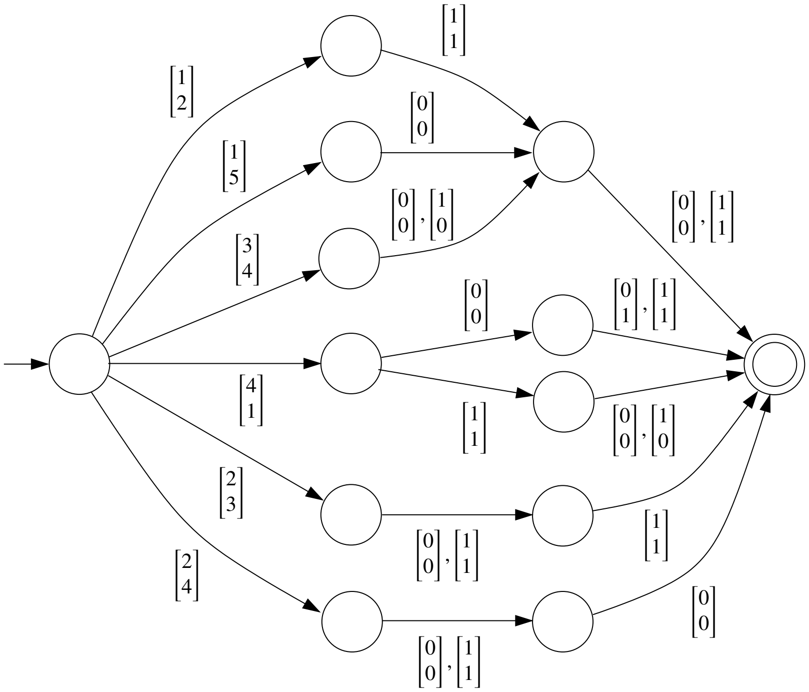

$\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}1\\2\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}$
$\begin{bmatrix}1\\5\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}3\\4\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}$
$\begin{bmatrix}0\\1\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}4\\1\end{bmatrix}$
$\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}$
$\begin{bmatrix}2\\3\end{bmatrix}$
$\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}2\\4\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}$

Figure 8.11: Transducer for the program of Figure 8.10

which describe the action of the instruction $y \leftarrow 1 - x$, namely

$$\begin{bmatrix}400\\101\end{bmatrix} \quad \begin{bmatrix}401\\101\end{bmatrix} \quad \begin{bmatrix}410\\110\end{bmatrix} \quad \begin{bmatrix}411\\110\end{bmatrix}.$$

Figure 8.12 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 8.11 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 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 8.13 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 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 8.14. 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 8.14 can be merged, yielding the minimal DFAs of Figure 8.15. Observe also that the set of all configurations, reachable or not, contains 120 elements, but is recognized by a five-state DFA.

8.5 Safety and Liveness Properties

Apart from the state-explosion problem, the automata-theoretic approach to automatic verification as described in this chapter has a second limitation: it assumes that the violations of the property can be witnessed by finite executions. In other words, if an execution violates the property, then we can detect the violation is already violated after finite time. Not all properties satisfy this assumption. 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 we cannot say that the property has been violated: the process might still enter the critical section 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 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. As an appetizer, the exercises start to discuss them.

Exercises

Exercise 95 Exhibit a family $\{P_n\}_{n\geq 1}$ of sequential programs (like the one of Example 8.1) satisfying the following conditions:

- P_n has $\mathcal{O}(n)$ variables, all of them boolean, $\mathcal{O}(n)$ lines, and exactly one initial configuration.
- P_i has at least 2^i reachable configurations.

Exercise 96 Modify *AsyncProduct* so that it produces system automata as those shown in Figure 8.1 for the program of Example 8.1.

Exercise 97 Prove:

(1) Parallel composition is associative, commutative, and idempotent. That is: $(L_1 \parallel L_2) \parallel L_3 = L_1 \parallel (L_2 \parallel L_3)$ (associativity); $L_1 \parallel L_2 = L_2 \parallel L_1$ (commutativity), and $L \parallel L = L$ (idempotence).

(2) If $L_1, L_2 \subseteq \Sigma^*$, then $L_1 \parallel L_2 = L_1 \cap L_2$.

(3) Let $\mathcal{A} = \langle A_1, \ldots, \mathcal{A}_n \rangle$ be a network of automata. Then $L(\mathcal{A})) = L(A_1) \parallel L(A_2)$.

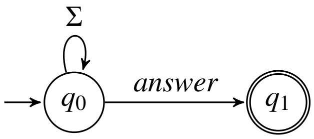
Exercise 98 Let $\Sigma = \{request, answer, working, idle\}$.

(1) Build a regular expression and 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* before which only *working* and *request* occur.

(4) Using automata theoretic constructions, prove that all words accepted by the automaton A below satisfy P_1, and give a regular expression for all words accepted by the automaton that violate P_2.



Exercise 99 Consider two processes (process 0 and process 1) being executed through the following generic mutual exclusion algorithm:

```
while true do
    enter(process_id)
    /* critical section                                   */
    leave(process_id)
    for arbitrarily many times do
        /* non critical section                           */
```

1. Consider the following implementations of **enter** and **leave**:

```
$x_0 \leftarrow 0$
enter($i$):
    while $x = 1 - i$ do
        pass
leave($i$):
    $x \leftarrow 1 - i$
```

   (a) Design a network of automata capturing the executions of the two processes.

   (b) Build the asynchronous product of the network.

   (c) Show that both processes cannot reach their critical sections at the same time.

   (d) If a process wants to enter its critical section, is it always the case that it can eventually enter it? (Hint: reason in terms of infinite executions.)

2. Consider the following alternative implementations of `enter` and `leave`:

```
$x_0 \leftarrow false$
$x_1 \leftarrow false$
enter($i$):
    $x_i \leftarrow true$
    while $x_{1-i}$ do
        pass
leave($i$):
    $x_i \leftarrow false$
```

   (a) Design a network of automata capturing the executions of the two processes.

   (b) Can a deadlock occur, i.e. can both processes get stuck trying to enter their critical sections?

Exercise 100 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. In other words, initially the trains occupy the tracks 2, 4, and 6.

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 network of automata $\mathcal{A} = \rangle C_0, \ldots, C_7, T_1, T_2, T_3\rangle$ 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 $i = 1, 2, 3$: $L(\mathcal{A}) \mid_{\Sigma_i} = (\, enter[i, 2i] \; enter[i, 2i \oplus 1] \ldots \; enter[i, 2i \oplus 7] \,)^*$.
  (No deadlocks, and every train eventually visits every segment.)

- For every word $w \in L(\mathcal{A})$: 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.)

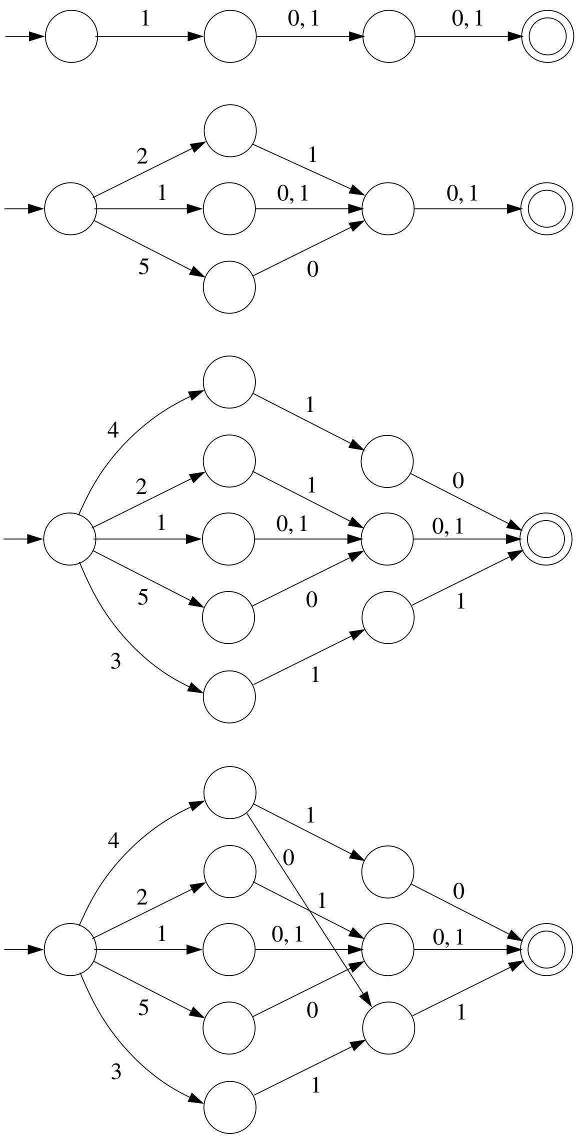


Figure 8.12: Minimal DFAs for the reachable configurations of the program of Figure 8.10

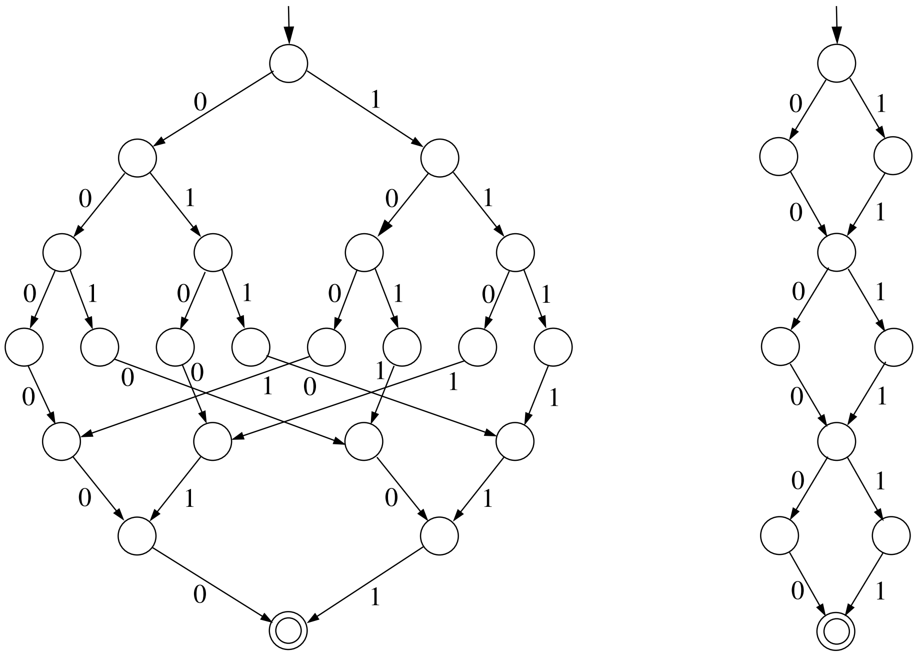


Figure 8.13: Minimal DFAs for the languages L_1 and L_2

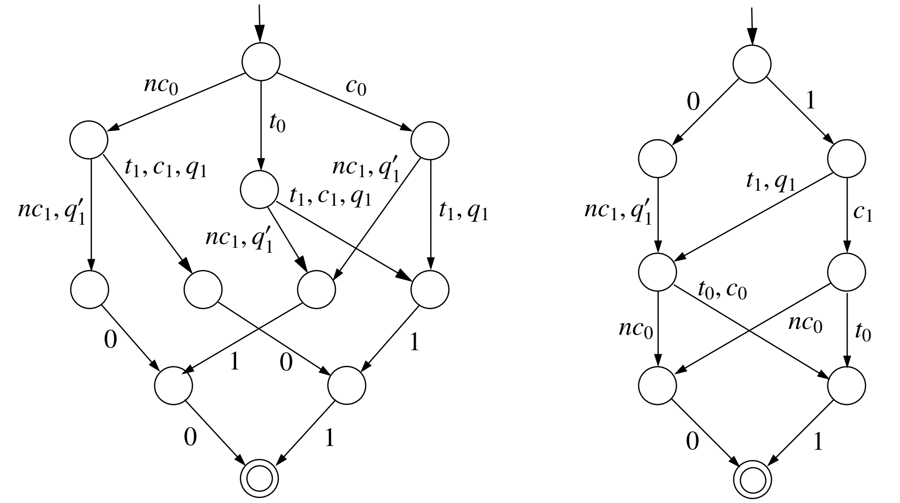


Figure 8.14: Minimal DFAs for the reachable configurations of Lamport's algorithm. On the left a configuration $\langle s_0, s_1, v_0, v_1, q \rangle$ is encoded by the word $s_0 s_1 v_0 v_1 q$, on the right by $v_1 s_1 s_0 v_0$.

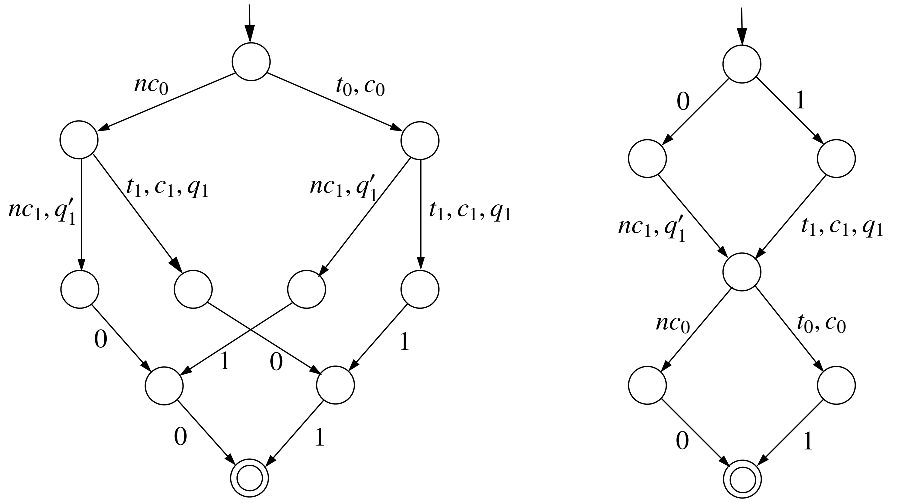


Figure 8.15: Minimal DFAs for the reachable configurations of Lamport's algorithm plus $\langle c_0, c_1, 1, 1 \rangle$.

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 interpreted 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 in *declarative* style, 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 a declarative description. A language may have a simple declarative description and a complicated operational description as a regular expression. For instance, the regular expression

$$(aa+bb+(ab+ba)(aa+bb)^*(ba+ab))^*$$

is a natural operational description of the language above, and it is arguably less intuitive than the declarative 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 declarative 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 declarative into operational descriptions, and vice versa.

9.1 First-Order Logic on Words

In declarative style, a language is defined by its *membership predicate*, i.e., the property that words must satisfy in order to belong to it. 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 as" 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 (i.e., lies to the left of) position y". For example, the property "if the letter at a position is an a, then all letters to the right of this position are also as" 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 set 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:*

$$\varphi := Q_a(x) \mid x < y \mid \neg\varphi \mid (\varphi \vee \varphi) \mid \exists x\, \varphi \ .$$

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 $(ab, x \mapsto 2)$.

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:*

$$\begin{array}{lcll}
(w, \mathcal{I}) & \models & Q_a(x) & \textit{iff} \quad w[\mathcal{I}(x)] = a \\
(w, \mathcal{I}) & \models & x < y & \textit{iff} \quad \mathcal{I}(x) < \mathcal{I}(y) \\
(w, \mathcal{I}) & \models & \neg\varphi & \textit{iff} \quad (w, \mathcal{I}) \not\models \varphi \\
(w, \mathcal{I}) & \models & \varphi 1 \vee \varphi_2 & \textit{iff} \quad (w, \mathcal{I}) \models \varphi_1 \textit{ or } (w, \mathcal{I}) \models \varphi_2 \\
(w, \mathcal{I}) & \models & \exists x\, \varphi & \textit{iff} \quad |w| \geq 1 \textit{ and some } i \in \{1, \ldots, |w|\} \textit{ satisfies } (w, \mathcal{I}[i/x]) \models \varphi
\end{array}$$

where $w[i]$ is the letter of w at position i, and $\mathcal{I}[i/x]$ is the mapping 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 $\mathfrak{I}$.

We use some standard abbreviations:

$$\forall x\, \varphi := \neg \exists\, x \neg\varphi \qquad \varphi_1 \wedge \varphi_2 := \neg\,(\neg\varphi_1 \vee \neg\varphi_2) \qquad \varphi_1 \rightarrow \varphi_2 := \neg\varphi_1 \vee \varphi_2$$

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. Further useful abbreviations are:

$$\begin{array}{rcll}
x = y & := & \neg(x < y \vee y < x) & \\
\text{first}(x) & := & \neg\exists y\, y < x & \text{“}x \text{ is the first position”} \\
\text{last}(x) & := & \neg\exists y\, x < y & \text{“}x \text{ is the last position”} \\
y = x + 1 & := & x < y \wedge \neg\exists\, z(x < z \wedge z < y) & \text{“}y \text{ is the successor position of } x\text{”} \\
y = x + 2 & := & \exists\, z(z = x + 1 \wedge y = z + 1) & \\
y = x + (k+1) & := & \exists\, z(z = x + k \wedge y = z + 1) &
\end{array}$$

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.”

$$\exists x\, Q_b(x) \wedge \forall x\, (\text{last}(x) \rightarrow Q_b(x) \;\wedge\; \neg\text{last}(x) \rightarrow Q_a(x))$$

- “Every a is immediately followed by a b.”

$$\forall x\, (Q_a(x) \rightarrow \exists y\, (y = x + 1 \wedge Q_b(y)))$$

- “Every a is immediately followed by a b, unless it is the last letter.”

$$\forall x\, (Q_a(x) \rightarrow \forall y\, (y = x + 1 \rightarrow Q_b(y)))$$

- “Between every a and every later b there is a c.”

$$\forall x \forall y\, (Q_a(x) \wedge Q_b(y) \wedge x < y \rightarrow \exists z\, (x < z \wedge z < y \wedge Q_c(z)))$$

□

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:

$$\begin{array}{rcl} x + k < y & := & \exists z\, (z = x + k \wedge z < y) \\ x < y + k & := & \exists z\, (z = y + k \wedge x < z) \\ k < last & := & \forall x\, (\text{last}(x) \rightarrow x > k) \end{array}$$

In these macros k is a constant, that is, $k < last$ standa for the infinite family of macros $1 < last, 2 < last, 3 < last \ldots$. 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 $(last > k_1 - 1 \wedge last < k_1 + 1) \vee \ldots \vee (last > k_1 - 1 \wedge last < k_1 + 1)$. 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:

$$\begin{aligned} \text{second}(x) &:= \exists y\ (\text{first}(y) \wedge x = y + 1) \\ \text{Even}(X) &:= \forall x\ \ (x \in X \leftrightarrow (\text{second}(x) \vee \exists y\ \ (x = y + 2 \wedge y \in X))) \end{aligned}$$

For the complete formula, we observe that the word has even length if its last position is even:

$$\text{EvenLength} := \exists X\ (\text{Even}(X) \wedge \forall x\ (\text{last}(x) \rightarrow x \in X)\ )$$

[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.

We now define the formal syntax and semantics of the logic.

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 $(w, \mathfrak{I})$ where $w \in \Sigma^*$, and $\mathfrak{I}$ is a mapping that assigns every free first-order variable x a* position *$\mathfrak{I}(x) \in \{1, \ldots, |w|\}$ and every free second-order variable X a set of positions $\mathfrak{I}(X) \subseteq \{1, \ldots, |w|\}$. (The mapping may also assign positions to other variables.)*

The satisfaction relation $(w, \mathfrak{I}) \models \varphi$ between a formula φ of $MSO(\Sigma)$ and an interpretation $(w, \mathfrak{I})$ of φ is defined as for $FO(\Sigma)$, with the following additions:

$$\begin{array}{lclll} (w,\mathfrak{I}) & \models & x \in X & \textit{iff} & \mathfrak{I}(x) \in \mathfrak{I}(X) \\ (w,\mathfrak{I}) & \models & \exists X\, \varphi & \textit{iff} & |w| > 0 \textit{ and some } S \subseteq \{1, \ldots, |w|\} \\ & & & & \textit{satisfies } (w, \mathfrak{I}[S/X]) \models \varphi \end{array}$$

where $\mathfrak{I}[S/X]$ is the interpretation that assigns S to X and otherwise coincides with $\mathfrak{I}$ — whether $\mathfrak{I}$ is defined for X or not. If $(w, \mathfrak{I}) \models \varphi$ we say that $(w, \mathfrak{I})$ 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

$$\forall x \in X\, \varphi \;:=\; \forall x\, (x \in X \rightarrow \varphi) \qquad \exists x \in X\, \varphi \;:=\; \exists x\, (x \in X \wedge \varphi)$$

9.2.1 Expressive power 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."

$$\text{Block}(X) := \forall x \in X\ \forall y \in X\ \ (x < y \rightarrow (\forall z\ (x < z \wedge z < y) \rightarrow z \in X))$$

- "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."

$$\begin{array}{rl}\text{Alternate}(X) := \ \forall x \in X & (\ Q_a(x) \rightarrow \forall y \in X\ (y = x + 1 \rightarrow Q_b(y)\ ) \\ & \qquad\qquad\qquad \wedge \\ & \ Q_b(x) \rightarrow \forall y \in X\ (y = x + 1 \rightarrow Q_a(y)\ )\ )\end{array}$$

- "The first letter in X is an a and the last is a b."

$$\begin{array}{rcl}\text{First\_a}(X) & := & \forall x \in X\ \forall y\ (y < x \rightarrow \neg y \in X) \rightarrow Q_a(x) \\ \text{Last\_b}(X) & := & \forall x \in X\ \forall y\ (y > x \rightarrow \neg y \in X) \rightarrow Q_a(x)\end{array}$$

Putting everything together, we get the formula

$$\begin{array}{ll}\exists X(& \text{Block}(X) \wedge \text{Before\_only\_c}(X) \wedge \text{After\_only\_d}(X) \wedge \\ & \text{Alternate}(X) \wedge \text{First\_a}(X) \wedge \text{Last\_b}(X)\ )\end{array}$$

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

$$P_q = \left\{ i \in \{1, \ldots, m\} \mid \hat{\delta}(q_0, a_1 \ldots a_i) = q \right\} .$$

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 a formula $\mathrm{Visits}(X_0, \ldots X_n)$ 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, $\mathrm{Visits}(X_0, \ldots X_n)$ 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

$$\psi_A := \exists X_0 \ldots \exists X_n \ \mathrm{Visits}(X_0, \ldots X_n) \wedge \exists x \left(\mathrm{last}(x) \wedge \bigvee_{q_i \in F} x \in X_i \right)$$

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 $\mathrm{Visits}(X_0, \ldots X_n)$. 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:

$$\mathrm{Init}(X_0, \ldots, X_n) = \exists x \left(\mathrm{first}(x) \wedge \left(\bigvee_{a \in \Sigma} (Q_a(x) \wedge x \in X_{i_a}) \right) \right)$$

(in words: if the letter at position 1 is a, then the position belongs to X_{i_a}).
Formula for (b):

$$\mathrm{Partition}(X_0, \ldots, X_n) = \forall x \left(\bigvee_{i=0}^{n} x \in X_i \ \wedge \bigwedge_{\substack{i,j=0 \\ i \neq j}}^{n} (x \in X_i \rightarrow x \notin X_j) \right)$$

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 $(w, \mathfrak{I})$ where $\mathfrak{I}$ 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

$$\left(aab\,, \begin{array}{c} x \mapsto 1 \\ y \mapsto 3 \\ X \mapsto \{2, 3\} \\ Y \mapsto \{1, 2\} \end{array} \right) \qquad \left(ba\,, \begin{array}{c} x \mapsto 2 \\ y \mapsto 1 \\ X \mapsto \emptyset \\ Y \mapsto \{1\} \end{array} \right)$$

Given an interpretation $(w, \mathfrak{I})$, 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 $(w, \mathfrak{I})$ 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

$$\begin{array}{c|ccc} & a & a & b \\ x & 1 & 0 & 0 \\ y & 0 & 0 & 1 \\ X & 0 & 1 & 1 \\ Y & 1 & 1 & 0 \end{array} \qquad \text{and} \qquad \begin{array}{c|cc} & b & a \\ x & 0 & 1 \\ y & 1 & 0 \\ X & 0 & 0 \\ Y & 1 & 0 \end{array}$$

corresponding to the words

$$\begin{bmatrix}a\\1\\0\\0\\1\end{bmatrix}\begin{bmatrix}a\\0\\0\\1\\1\end{bmatrix}\begin{bmatrix}b\\0\\1\\1\\0\end{bmatrix} \quad\text{and}\quad \begin{bmatrix}b\\0\\1\\0\\1\end{bmatrix}\begin{bmatrix}a\\1\\0\\0\\0\end{bmatrix} \quad\text{over } \Sigma\times\{0,1\}^4$$

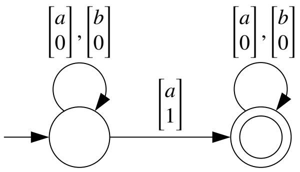
Definition 9.13 *Let φ be a formula with n free variables, and let $(w, \mathfrak{I})$ be an interpretation of φ. We denote by* $\mathrm{enc}(w, \mathfrak{I})$ *the word over the alphabet* $\Sigma \times \{0,1\}^n$ *described above. The language of φ is* $L(\varphi) = \{\mathrm{enc}(w, \mathfrak{I}) \mid (w, \mathfrak{I}) \models \varphi\}$.

Now that we have associated 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 *free*(φ) the set of free variables of φ.

- $\varphi = Q_a(x)$. Then *free*(φ) $= x$, and the interpretations of φ are encoded as words over $\Sigma\times\{0,1\}$. The language $L(\varphi)$ is given by

$$L(\varphi) = \left\{ \begin{bmatrix}a_1\\b_1\end{bmatrix} \begin{matrix}\cdots\\\cdots\end{matrix} \begin{bmatrix}a_k\\b_k\end{bmatrix} \;\middle|\; \begin{array}{l} k \geq 0, \\ a_i \in \Sigma \text{ and } b_i \in \{0,1\} \text{ for every } i \in \{1,\ldots,k\}, \text{ and} \\ b_i = 1 \text{ for exactly one index } i \in \{1,\ldots,k\} \text{ such that } a_i = a \end{array} \right\}$$

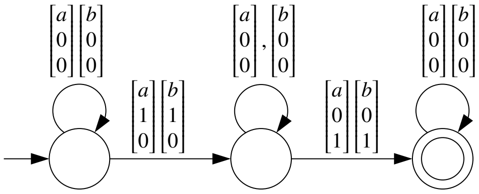
  and is recognized by


$\begin{bmatrix}a\\0\end{bmatrix},\begin{bmatrix}b\\0\end{bmatrix}$
$\begin{bmatrix}a\\1\end{bmatrix}$
$\begin{bmatrix}a\\0\end{bmatrix},\begin{bmatrix}b\\0\end{bmatrix}$

- $\varphi = x < y$. Then *free*(φ) $= \{x, y\}$, and the interpretations of ϕ are encoded as words over $\Sigma \times \{0,1\}^2$. The language $L(\varphi)$ is given by

$$L(\varphi) = \left\{ \begin{bmatrix}a_1\\b_1\\c_1\end{bmatrix} \begin{matrix}\cdots\\\cdots\end{matrix} \begin{bmatrix}a_k\\b_k\\c_k\end{bmatrix} \;\middle|\; \begin{array}{l} k \geq 0, \\ a_i \in \Sigma \text{ and } b_i, c_i \in \{0,1\} \text{ for every } i \in \{1,\ldots,k\}, \\ b_i = 1 \text{ for exactly one index } i \in \{1,\ldots,k\}, \\ c_j = 1 \text{ for exactly one index } j \in \{1,\ldots,k\}, \text{ and} \\ i < j \end{array} \right\}$$

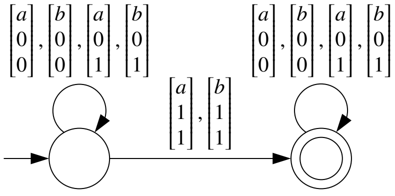
  and is recognized by



- $\varphi = x \in X$. Then $free(\varphi) = \{x, X\}$, and interpretations are encoded as words over $\Sigma \times \{0,1\}^2$. The language $L(\varphi)$ is given by

$$L(\varphi) = \left\{ \begin{bmatrix} a_1 \\ b_1 \\ c_1 \end{bmatrix} \begin{matrix} \dots \\ \dots \end{matrix} \begin{bmatrix} a_k \\ b_k \\ c_k \end{bmatrix} \left| \begin{array}{l} k \geq 0, \\ a_i \in \Sigma \text{ and } b_i, c_i \in \{0,1\} \text{ for every } i \in \{1, \dots, k\}, \\ b_i = 1 \text{ for exactly one index } i \in \{1, \dots, k\}, \text{ and} \\ \text{for every } i \in \{1, \dots, k\}, \text{ if } b_i = 1 \text{ then } c_i = 1 \end{array} \right. \right\}$$

and is recognized by



- $\varphi = \neg\psi$. Then $free(\varphi) = free(\psi)$, and by induction hypothesis there exists an automaton A_ψ s.t. $L\left(A_\psi\right) = L(\psi)$.

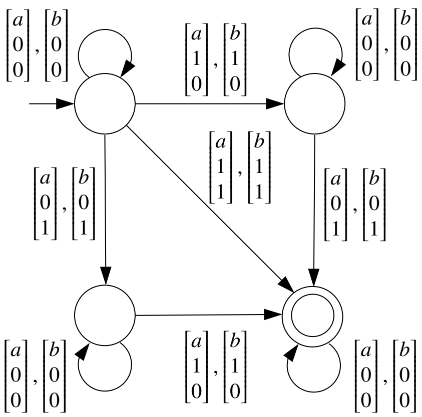
Observe that $L(\varphi)$ is *not* in general equal to $\overline{L(\psi)}$. To see why, consider for example the case $\psi = Q_a(x)$ and $\varphi = \neg Q_a(x)$. The word

$$\begin{bmatrix} a \\ 1 \end{bmatrix} \begin{bmatrix} a \\ 1 \end{bmatrix} \begin{bmatrix} a \\ 1 \end{bmatrix}$$

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 $A^{enc}_{x<y}$ is

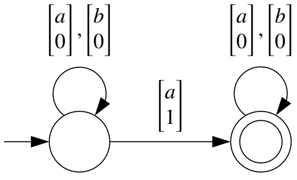

$\begin{bmatrix}a\\0\\0\end{bmatrix}, \begin{bmatrix}b\\0\\0\end{bmatrix}$
$\begin{bmatrix}a\\1\\0\end{bmatrix}, \begin{bmatrix}b\\1\\0\end{bmatrix}$
$\begin{bmatrix}a\\0\\0\end{bmatrix}, \begin{bmatrix}b\\0\\0\end{bmatrix}$
$\begin{bmatrix}a\\1\\1\end{bmatrix}, \begin{bmatrix}b\\1\\1\end{bmatrix}$
$\begin{bmatrix}a\\0\\1\end{bmatrix}, \begin{bmatrix}b\\0\\1\end{bmatrix}$
$\begin{bmatrix}a\\0\\1\end{bmatrix}, \begin{bmatrix}b\\0\\1\end{bmatrix}$
$\begin{bmatrix}a\\0\\0\end{bmatrix}, \begin{bmatrix}b\\0\\0\end{bmatrix}$
$\begin{bmatrix}a\\1\\0\end{bmatrix}, \begin{bmatrix}b\\1\\0\end{bmatrix}$
$\begin{bmatrix}a\\0\\0\end{bmatrix}, \begin{bmatrix}b\\0\\0\end{bmatrix}$

Observe that the number of states of A^{enc}_{ψ} grows exponentially in the number of free variables. This makes the negation operation expensive, even when the automaton A_{ϕ} is deterministic.

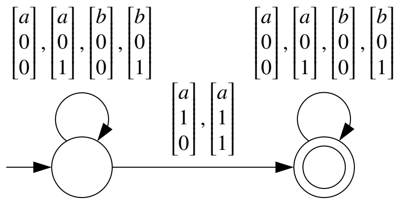
- $\varphi = \varphi_1 \vee \varphi_2$. Then $free(\varphi) = free(\varphi_1) \cup free(\varphi_2)$, and by induction hypothesis there are automata $A_{\varphi_i}, A_{\varphi_2}$ such that $\mathcal{L}(A_{\varphi_1}) = \mathcal{L}(\varphi_1)$ and $\mathcal{L}(A_{\varphi_2}) = \mathcal{L}(\varphi_2)$.

  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 $\mathcal{L}(Q_a(x))$

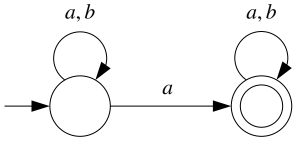

$\begin{bmatrix}a\\0\end{bmatrix}, \begin{bmatrix}b\\0\end{bmatrix}$
$\begin{bmatrix}a\\1\end{bmatrix}$
$\begin{bmatrix}a\\0\end{bmatrix}, \begin{bmatrix}b\\0\end{bmatrix}$

  we transform it into an automaton A_1 recognizing L_1


$\begin{bmatrix}a\\0\\0\end{bmatrix},\begin{bmatrix}a\\0\\1\end{bmatrix},\begin{bmatrix}b\\0\\0\end{bmatrix},\begin{bmatrix}b\\0\\1\end{bmatrix}$
$\begin{bmatrix}a\\1\\0\end{bmatrix},\begin{bmatrix}a\\1\\1\end{bmatrix}$
$\begin{bmatrix}a\\0\\0\end{bmatrix},\begin{bmatrix}a\\0\\1\end{bmatrix},\begin{bmatrix}b\\0\\0\end{bmatrix},\begin{bmatrix}b\\0\\1\end{bmatrix}$

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. $L\left(A_\psi\right) = L(\psi)$. 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.

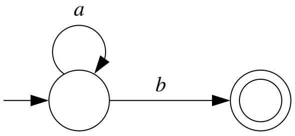
- $\varphi = \exists X\ \varphi$. We proceed as in the previous case.

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



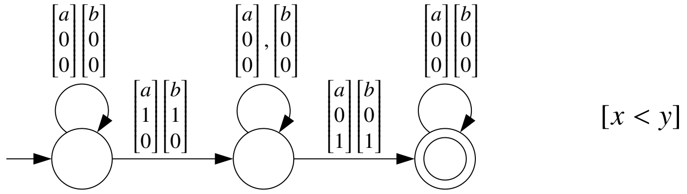
We derive this NFA by giving a formula φ such that $L(\varphi) = a^*b$, and then using the procedure described above. We shall see that the procedure is quite laborious. The formula states that the last letter is b, and all other letters are a's.

$$\varphi = \exists x\ (\text{last}(x) \wedge Q_b(x)) \ \wedge\ \forall x\ (\neg\text{last}(x) \rightarrow Q_a(x))$$

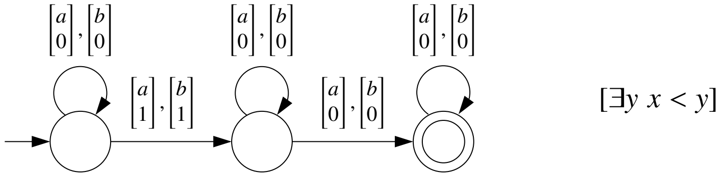
We first bring φ into the equivalent form

$$\psi = \exists x\ (\text{last}(x) \wedge Q_b(x)) \ \wedge\ \neg\exists x\ (\neg\text{last}(x) \wedge \neg Q_a(x))$$

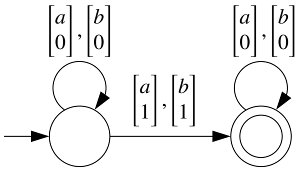
We transform ψ into an NFA. First, we compute an automaton for $\text{last}(x) \ = \ \neg\exists y\ x < y$. Recall that the automaton for $x < y$ is



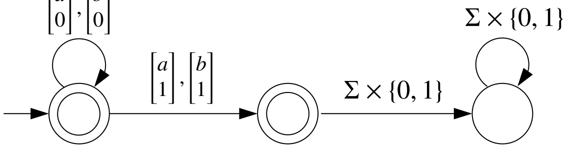
Applying the projection operation, we get following automaton for $\exists y\ x < y$



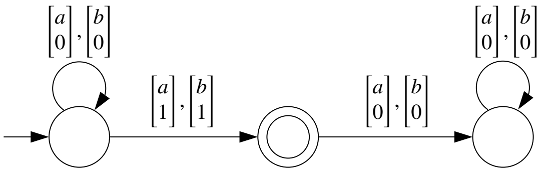
Recall that computing the automaton for the negation of a formula requires more than complementing the automaton. First,we need an automaton recognizing $Enc(\exists y\ x < y)$.



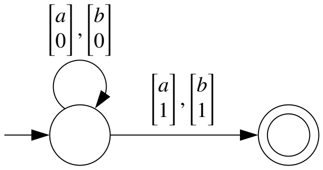
Second, we determinize and complement the automaton for $\exists y\ x < y$:



And finally, we compute the intersection of the last two automata, getting

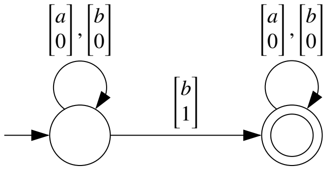

$\begin{bmatrix}a\\0\end{bmatrix}, \begin{bmatrix}b\\0\end{bmatrix}$
$\begin{bmatrix}a\\1\end{bmatrix}, \begin{bmatrix}b\\1\end{bmatrix}$
$\begin{bmatrix}a\\0\end{bmatrix}, \begin{bmatrix}b\\0\end{bmatrix}$
$\begin{bmatrix}a\\0\end{bmatrix}, \begin{bmatrix}b\\0\end{bmatrix}$

whose last state is useless and can be removed, yielding the following NFA for last(x):


$\begin{bmatrix}a\\0\end{bmatrix}, \begin{bmatrix}b\\0\end{bmatrix}$
$\begin{bmatrix}a\\1\end{bmatrix}, \begin{bmatrix}b\\1\end{bmatrix}$

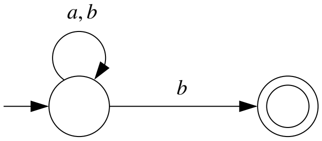
[last(x)]

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 $Q_b(x)$


$\begin{bmatrix}a\\0\end{bmatrix}, \begin{bmatrix}b\\0\end{bmatrix}$
$\begin{bmatrix}b\\1\end{bmatrix}$
$\begin{bmatrix}a\\0\end{bmatrix}, \begin{bmatrix}b\\0\end{bmatrix}$

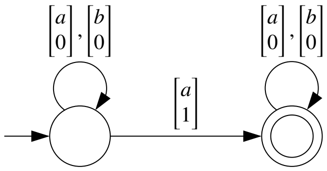
[$Q_b(x)$]

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



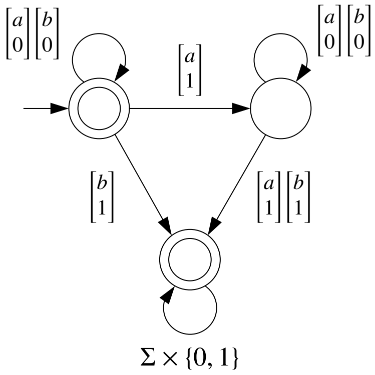
[$\exists x\ (\text{last}(x) \wedge Q_b(x))$]

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

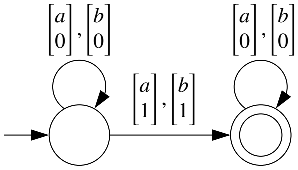

$\begin{bmatrix}a\\0\end{bmatrix}, \begin{bmatrix}b\\0\end{bmatrix}$
$\begin{bmatrix}a\\1\end{bmatrix}$
$\begin{bmatrix}a\\0\end{bmatrix}, \begin{bmatrix}b\\0\end{bmatrix}$

[$Q_a(x)$]

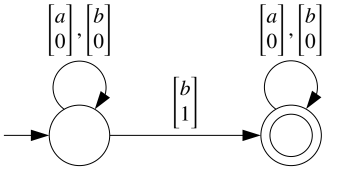
and after determinization and complementation we get


$\begin{bmatrix}a\\0\end{bmatrix}\begin{bmatrix}b\\0\end{bmatrix}$
$\begin{bmatrix}a\\1\end{bmatrix}$
$\begin{bmatrix}a\\0\end{bmatrix}\begin{bmatrix}b\\0\end{bmatrix}$
$\begin{bmatrix}b\\1\end{bmatrix}$
$\begin{bmatrix}a\\1\end{bmatrix}\begin{bmatrix}b\\1\end{bmatrix}$
$\Sigma \times \{0, 1\}$

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


$\begin{bmatrix}a\\0\end{bmatrix}, \begin{bmatrix}b\\0\end{bmatrix}$
$\begin{bmatrix}a\\1\end{bmatrix}, \begin{bmatrix}b\\1\end{bmatrix}$
$\begin{bmatrix}a\\0\end{bmatrix}, \begin{bmatrix}b\\0\end{bmatrix}$

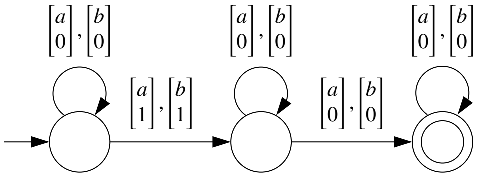
The intersection of the last two automata yields a three-state automaton for $\neg Q_a(x)$, but after eliminating a useless state we get


$\begin{bmatrix}a\\0\end{bmatrix}, \begin{bmatrix}b\\0\end{bmatrix}$
$\begin{bmatrix}b\\1\end{bmatrix}$
$\begin{bmatrix}a\\0\end{bmatrix}, \begin{bmatrix}b\\0\end{bmatrix}$

$[\neg Q_a(x)]$

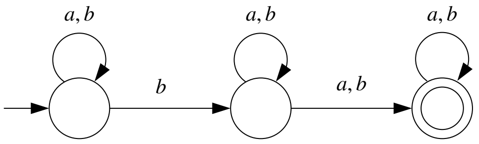
Notice that this is the same automaton we obtained for $Q_b(x)$, which is fine, because over the alphabet $\{a, b\}$ the formulas $Q_b(x)$ and $\neg Q_a(x)$ are equivalent.

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


$\begin{bmatrix}a\\0\end{bmatrix}, \begin{bmatrix}b\\0\end{bmatrix}$
$\begin{bmatrix}a\\1\end{bmatrix}, \begin{bmatrix}b\\1\end{bmatrix}$
$\begin{bmatrix}a\\0\end{bmatrix}, \begin{bmatrix}b\\0\end{bmatrix}$
$\begin{bmatrix}a\\0\end{bmatrix}, \begin{bmatrix}b\\0\end{bmatrix}$
$\begin{bmatrix}a\\0\end{bmatrix}, \begin{bmatrix}b\\0\end{bmatrix}$

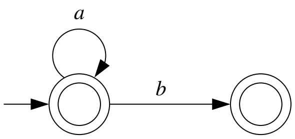
$[\neg last(x)]$

Intersecting the automata for $\neg last(x)$ and $\neg Q_a(x)$, and subsequently projecting onto the first component, we get an automaton for $\exists x\ (\neg last(x) \wedge \neg Q_a(x))$



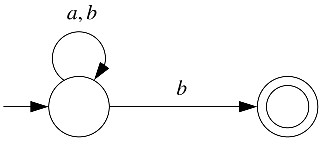
$[\exists x\ (\neg last(x) \wedge \neg Q_a(x))]$

Determinizing, complementing, and removing a useless state yields the following NFA for $\neg\exists x\ (\neg last(x) \wedge \neg Q_a(x))$:

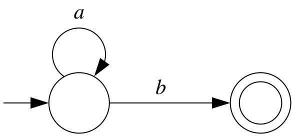


$[\neg\exists x\ (\neg last(x) \wedge \neg Q_a(x))]$

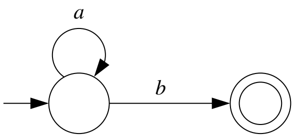
Summarizing, the automata for the two conjuncts of ψ are



and



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 101 Give formulations in plain English of the languages described by the following formulas of FO($\{a, b\}$), and give a corresponding regular expression:

(a) $\exists x\, first(x)$

(b) $\forall x\, first(x)$

(c)
$$\begin{array}{ll} & \neg\exists x \exists y\, (x < y \wedge Q_a(x) \wedge Q_b(y)) \\ \wedge & \forall x\, (Q_b(x) \rightarrow \exists y\; x < y \wedge Q_a(y)) \\ \wedge & \exists x\, \neg\exists y\; x < y \end{array}$$

Exercise 102 Let $\Sigma = \{a, b\}$.

(a) Give a formula $\varphi_n(x, y)$ of FO(Σ), of size $\mathcal{O}(n)$, that holds iff $y = x + 2^n$. (Notice that the abbreviation $y = x + k$ of page 9.1 has length $\mathcal{O}(k)$, and so it cannot be directly used.)

(b) Give a sentence of FO(Σ), of size $\mathcal{O}(n)$, for the language $L_n = \{ww \mid w \in \Sigma^* \text{ and } |w| = 2^n\}$.

(c) Show that the minimal DFA accepting L_n has at least 2^{2^n} states.
(Hint: consider the residuals of L_n.)

Exercise 103 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 $k \leq 2^n$.

Hint: Modify suitably the proof of Theorem 9.8.

Exercise 104 Let Σ be a finite alphabet. A language $L \subseteq \Sigma^*$ is *star-free* if it can be expressed by a star-free regular expression, i.e. a regular expression where the Kleene star operation is forbidden, but complementation is allowed. For example, Σ^* is star-free since $\Sigma^* = \overline{\emptyset}$, but $(aa)^*$ is not.

(a) Give star-free regular expressions and FO(Σ) sentences for the following star-free languages:

(i) Σ^+.
(ii) $\Sigma^* A \Sigma^*$ for some $A \subseteq \Sigma$.
(iii) A^* for some $A \subseteq \Sigma$.
(iv) $(ab)^*$.
(v) $\{w \in \Sigma^* \mid w \text{ does not contain } aa\}$.

(b) Show that finite and cofinite languages are star-free.

(c) Show that for every sentence $\varphi \in$ FO(Σ), there exists a formula $\varphi^+(x, y)$, with two free variables x and y, such that for every $w \in \Sigma^+$ and for every $1 \leq i \leq j \leq w$,

$$w \models \varphi^+(i, j) \quad \text{iff} \quad w_i w_{i+1} \cdots w_j \models \varphi\,.$$

(d) Give a polynomial time algorithm that decides whether the empty word satisfies a given sentence of FO(Σ).

(e) Show that every star-free language can be expressed by an FO(Σ) sentence. (Hint: use (c).)

Exercise 105 Give a MSO-formula Odd_card(X) expressing that the cardinality of the set of positions X is odd. *Hint*: Follow the pattern of the formula Even(X).

Exercise 106 Given a formula φ of MSO(Σ) and a second order variable X not occurring in φ, show how to construct a formula φ^X with X as free variable expressing "the projection of the word onto the positions of X satisfies φ". Formally, φ^X must satisfy the following property: for every interpretation $\mathcal{I}$ of φ^X, we have $(w, \mathcal{I}) \models \varphi^X$ iff $(w|_{\mathcal{I}(X)}, \mathcal{I}) \models \varphi$, where $w|_{\mathcal{I}(X)}$ denotes the result of deleting from w the letters at all positions that do not belong to $\mathcal{I}(X)$.

Exercise 107 (1) Given two sentences φ_1 and φ_2 of MSO(Σ), construct a sentence Conc(φ_1, φ_2) satisfying $L(\text{Conc}(\varphi_1, \varphi_2)) = L(\varphi_1) \cdot L(\varphi_2)$.

(2) Given a sentence φ of MSO(Σ), construct a sentence Star(φ) satisfying $L(\text{Star}(\varphi)) = L(\varphi)^*$.

(3) Give an algorithm *RegtoMSO* that accepts a regular expression r as input and directly constructs a sentence φ of MSO(Σ) such that $L(\varphi) = L(r)$, without first constructing an automaton for the formula.

Hint: Use the solution to Exercise 106.

Exercise 108 Consider the logic PureMSO(Σ) with syntax

$$\varphi := X \subseteq Q_a \mid X < Y \mid X \subseteq Y \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists X\, \varphi$$

Notice that formulas of PureMSO(Σ) do not contain first-order variables. The satisfaction relation of PureMSO(Σ) is given by:

$$\begin{array}{lcll}
(w, \mathcal{I}) & \models & X \subseteq Q_a & \text{iff} \quad w[p] = a \text{ for every } p \in \mathcal{I}(X) \\
(w, \mathcal{I}) & \models & X < Y & \text{iff} \quad p < p' \text{ for every } p \in \mathcal{I}(X),\ p' \in \mathcal{I}(Y) \\
(w, \mathcal{I}) & \models & X \subseteq Y & \text{iff} \quad p < p' \text{ for every } p \in \mathcal{I}(X),\ p' \in \mathcal{I}(Y)
\end{array}$$

with the rest as for MSO(Σ).

Prove that MSO(Σ) and PureMSO(Σ) have the same expressive power for sentences. That is, show that for every sentence ϕ of MSO(Σ) there is an equivalent sentence ψ of PureMSO(Σ), and vice versa.

Exercise 109 Recall the syntax of MSO(Σ):

$$\varphi := Q_a(x) \mid x < y \mid x \in X \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x\, \varphi \mid \exists X\, \varphi$$

We have introduced $y = x + 1$ ("y is the successor position of x") as an abbreviation

$$y = x + 1 := \; x < y \wedge \neg \exists z\, (x < z \wedge z < y)$$

Consider now the variant MSO′(Σ) in which, instead of an abbreviation, $y = x + 1$ is part of the syntax and replaces $x < y$. In other words, the syntax of MSO′(Σ) is

$$\varphi := Q_a(x) \mid y = x + 1 \mid x \in X \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x\, \varphi \mid \exists X\, \varphi$$

Prove that MSO′(Σ) has the same expressive power as MSO(Σ) by finding a formula of SO′(Σ) with the same meaning as $x < y$.

Exercise 110 Give a defining formula of MSO($\{a, b\}$) for the following languages:

(a) aa^*b^*.

(b) The set of words with an odd number of occurrences of a.

(c) The set of words such that every two b with no other b in between are separated by a block of a of odd length.

Exercise 111 1. Give a formula Block_between of MSO(Σ) such that Block_between(X, i, j) holds whenever $X = \{i, i + 1, \ldots, j\}$.

2. Let $0 \leq m < n$. Give a formula $\text{Mod}^{m,n}$ of MSO(Σ) such that $\text{Mod}^{m,n}(i, j)$ holds whenever $|w_i w_{i+1} \cdots w_j| \equiv m \,(\text{mod}\; n)$, i.e. whenever $j - i + 1 \equiv m \,(\text{mod}\; n)$.

3. Let $0 \leq m < n$. Give a sentence of MSO(Σ) for $a^m(a^n)^*$.

4. Give a sentence of MSO($\{a, b\}$) for the language of words such that every two b's with no other b in between are separated by a block of a's of odd length.

Exercise 112 Consider a formula $\phi(X)$ of MSO(Σ) that does not contain any occurrence of the $Q_a(x)$. Given any two interpretations that assign to X the same set of positions, we have that either both interpretations satisfy $\phi(X)$, or none of them does. So we can speak of the sets of natural numbers (positions) satisfying $\phi(X)$. In this sense, $\phi(X)$ expresses a property of the finite sets of natural numbers, which a particular set may satisfy or not.

This observation can be used to automatically prove some (very) simple properties of the natural numbers. Consider for instance the following "conjecture": every finite set of natural numbers has a minimal element[2]. The conjecture holds iff the formula

$$\text{Has\_min}(X) := \exists x \in X\; \forall y \in X\; x \leq y$$

is satisfied by every interpretation in which X is nonempty. Construct an automaton for Has_min(X), and check that it recognizes all nonempty sets.

[2] Of course, this also holds for every infinite set, but we cannot prove it using MSO over finite words.

Exercise 113 The encoding of a set is a string, that can be seen as the encoding of a number. We can use this observation to express addition in monadic second-order logic. More precisely, find a formula Sum(X, Y, Z) that is true iff $n_X + n_Y = n_Z$, where x, y, z are the numbers encoded by the sets X, Y, Z, respectively, using the LSBF-encoding. For instance, the words

$$\begin{array}{c} X \\ Y \\ Z \end{array} \quad \begin{bmatrix}0\\1\\1\end{bmatrix}\begin{bmatrix}1\\1\\0\end{bmatrix}\begin{bmatrix}0\\0\\1\end{bmatrix} \quad \text{and} \quad \begin{array}{c} X \\ Y \\ Z \end{array} \quad \begin{bmatrix}1\\1\\0\end{bmatrix}\begin{bmatrix}1\\1\\1\end{bmatrix}\begin{bmatrix}1\\1\\1\end{bmatrix}\begin{bmatrix}1\\1\\1\end{bmatrix}\begin{bmatrix}1\\0\\0\end{bmatrix}\begin{bmatrix}0\\0\\1\end{bmatrix}\begin{bmatrix}0\\0\\0\end{bmatrix}\begin{bmatrix}0\\0\\0\end{bmatrix}$$

should satisfy the formula: the first encodes $2 + 3 = 5$, and the second encodes $31 + 15 = 46$.

Chapter 10

Applications III: Presburger Arithmetic

Presburger arithmetic is a logical language for expressing properties of numbers by means of addition and comparison. A typical example of such a property is "$x + 2y > 2z$ and $2x - 3z = 4y$". The property is satisfied by some triples (n_x, n_y, n_z) of natural numbers, like $(4, 2, 0)$ and $(8, 1, 4)$, but not by others, like $(6, 0, 4)$ or $(2, 2, 4)$. Valuations satisfying the property are called *solutions* or *models*. We show how to construct for a given formula φ of Presburger arithmetic an NFA A_φ recognizing the solutions of φ. In Section 10.1 we introduce the syntax and semantics of Presburger arithemetic. Section 10.2 constructs a NFA recognizing all solutions over the natural numbers, and Section 10.3 a NFA recognizing all solutions over the integers.

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 standard abbreviations like $\forall$, $\wedge$, $\rightarrow$, we also introduce:

$$\begin{array}{rclrcl}
n & := & \underbrace{1 + 1 + \ldots + 1}_{n \text{ times}} & t \geq t' & := & t' \leq t \\
 & & & t = t' & := & t \leq t' \wedge t \geq t' \\
nx & := & \underbrace{x + x + \ldots + x}_{n \text{ times}} & t < t' & := & t \leq t' \wedge \neg(t = t') \\
 & & & t > t' & := & t' < t
\end{array}$$

An *interpretation* is a function $\mathcal{I} \colon V \rightarrow \mathbb{N}$. 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 $\mathcal{I} \models \varphi$ for an interpretation $\mathcal{I}$ and a formula φ is inductively defined as follows, 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:

$$\begin{array}{lll}
\mathcal{I} \models t \leq u & \text{iff} & \mathcal{I}(t) \leq \mathcal{I}(u) \\
\mathcal{I} \models \neg\varphi_1 & \text{iff} & \mathcal{I} \not\models \varphi_1 \\
\mathcal{I} \models \varphi_1 \vee \varphi_2 & \text{iff} & \mathcal{I} \models \varphi_1 \text{ or } \mathcal{I} \models \varphi_2 \\
\mathcal{I} \models \exists x\, \varphi & \text{iff} & \text{there exists } n \geq 0 \text{ such that } \mathcal{I}[n/x] \models \varphi
\end{array}$$

It is easy to see that whether $\mathcal{I}$ satisfies φ or not depends only on the values $\mathcal{I}$ assigns to the *free* variables of φ (i.e., if two interpretations assign the the same values to the free variables , then either both satisfy the formula, or none does). The *solutions* of φ are the projection onto the free variables of φ of the interpretations that satisfy φ. if we fix a total order on the set V of variables and a formula φ has k free variables, then its set of solutions can be represented as a subset of $\mathbb{N}^k$, or as relation of arity k over the universe $\mathbb{N}$. We call this subset the *solution space* of φ, and 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 free variables of the formula $\exists x\, (2x = y \wedge 2y = z)$ are y and z. The solutions of the formula are the pairs $\{(2n, 4n) \mid n \geq 0\}$, where we assume that the first and second components correspond to the values of y and z, respectively. □

Automata encoding natural numbers. We use transducers to represent, compute and manipulate solution spaces of formulas. As in Section 6.1 of Chapter 6, we encode natural numbers as strings over $\{0, 1\}$ using the least-significant-bit-first encoding *LSBF*. If a formula has free variables $x_1, \ldots, x_k$, then its solutions are encoded as words over $\{0, 1\}^k$. For instance, the word

$$\begin{array}{cc}
x_1 & \\
x_2 & \\
x_3 &
\end{array}
\begin{bmatrix}1\\0\\0\end{bmatrix}\begin{bmatrix}0\\1\\0\end{bmatrix}\begin{bmatrix}1\\0\\0\end{bmatrix}\begin{bmatrix}0\\1\\0\end{bmatrix}$$

encodes the solution $(3, 10, 0)$. The language of a formula ϕ is defined as

$$L(\varphi) = \bigcup_{s \in Sol(\varphi)} LSBF(s)$$

where *LSBF*(s) denotes the set of all encodings of the tuple s of natural numbers. In other words, $L(\varphi)$ is the encoding of the relation *Sol*(φ).

10.2 An NFA for the Solutions over the Naturals.

Given a Presburger formula φ, we construct a transducer A_φ such that $L\left(A_\varphi\right) = L(\varphi)$. Recall that *Sol*($\varphi$) is a relation over $\mathbb{N}$ whose arity is given by the number of free variables of ϕ. The last section of Chapter 6 implements operations 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 automata do not accept words encoding nothing). For Presburger arithmetic this is not necessary, because in the *LSBF* encoding *every* word encodes 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*(φ_1), *Sol*(φ_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*(φ_i). Transducers recognizing $Sol_{V_1 \cup V_2}(\varphi_i)$ for $i = 1, 2$ are easy to compute from transducers recognizing *Sol*(φ_i), and the solution space is **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 **Projection_I**(*Sol*(φ)), where I contains the indices of all variables with the exception of the index of x.

It only remains to construct automata 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 $\mathbb{N}$!). 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 $-x + y \leq -4$. Letting $a = (a_1, \ldots, a_n)$, $x = (x_1, \ldots, x_n)$, and denoting the scalar product of a and x by $a \cdot x$ we write $\varphi = \ a \cdot x \leq b$.

We construct a DFA for *Sol*(φ). The states of the DFA are integers. We choose transitions and final states of the DFA so that the following property holds:

$$\text{State } q \in \mathbb{Z} \text{ recognizes the encodings of the tuples } c \in \mathbb{N}^n \text{ such that } a \cdot c \leq q. \tag{10.1}$$

Given a state $q \in \mathbb{Z}$ and a letter $\zeta \in \{0,1\}^n$, let us determine the target state q' of the transition $q \xrightarrow{\zeta} q'$ of the DFA, where $\zeta \in \{0,1\}^n$. A word $w' \in (\{0,1\}^n)^*$ is accepted from q' iff the word $\zeta w'$

is accepted from q. Since we use the *lsbf* encoding, the tuple of natural numbers encoded by $\zeta w'$ is $2c' + \zeta$, where $c' \in \mathbb{N}^n$ is the tuple encoded by w'. So $c' \in \mathbb{N}^n$ is accepted from q' iff $2c' + \zeta$ is accepted from q. Therefore, in order to satisfy property 10.1 we must choose q' so that $a \cdot c \leq q$ iff $a \cdot (2c + \zeta) \leq q'$. A little arithmetic yields

$$q' = \left\lfloor \frac{1}{2}(q - a \cdot \zeta) \right\rfloor$$

and so we define the transition function of the DFA by

$$\delta(q, \zeta) = \frac{1}{2}(q - a \cdot \zeta) \; .$$

For the final states we observe that a state is final iff it accepts the empty word iff it accepts the tuple $(0, \ldots, 0) \in \mathbb{N}^n$. So in order to satisfy property 10.1 we must make state q final iff $q \geq 0$. As initial state we choose b. This leads to the algorithm *AFtoDFA*(φ) of Table 10.1, where for clarity the state corresponding to an integer $k \in \mathbb{Z}$ is denoted by s_k.

```
AFtoDFA($\varphi$)
Input: Atomic formula $\varphi = a \cdot x \leq b$
Output: DFA $A_\varphi = (Q, \Sigma, \delta, q_0, F)$ such that $L\left(A_\varphi\right) = L(\varphi)$
$Q, \delta, F \leftarrow \emptyset$; $q_0 \leftarrow s_b$
$W \leftarrow \{s_b\}$
while $W \neq \emptyset$ do
   pick $s_k$ from $W$
   add $s_k$ to $Q$
   if $k \geq 0$ then add $s_k$ to $F$
   for all $\zeta \in \{0,1\}^n$ do
      $j \leftarrow \left\lfloor \frac{1}{2}(k - a \cdot \zeta) \right\rfloor$
      if $s_j \notin Q$ then add $s_j$ to $W$
      add $(s_k, \zeta, s_j)$ to $\delta$
```

Table 10.1: Converting an atomic formula into a DFA recognizing the *lsbf* encoding of its solutions.

Example 10.2 Consider the atomic formula $2x - y \leq 2$. The DFA obtained by applying *AFtoDFA* to it is shown in Figure 10.1. The initial state is 2. Transitions leaving state 2 are given by

$$\delta(2, \zeta) = \left\lfloor \frac{1}{2}(2 - (2, -1) \cdot (\zeta_x, \zeta_y)) \right\rfloor = \left\lfloor \frac{1}{2}(2 - 2\zeta_x + \zeta_y) \right\rfloor$$

and so we have $2 \xrightarrow{[0,0]} 1$, $2 \xrightarrow{[0,1]} 1$, $2 \xrightarrow{[1,0]} 0$ and $2 \xrightarrow{[1,1]} 0$. States 2, 1 , and 0 are final. The DFA

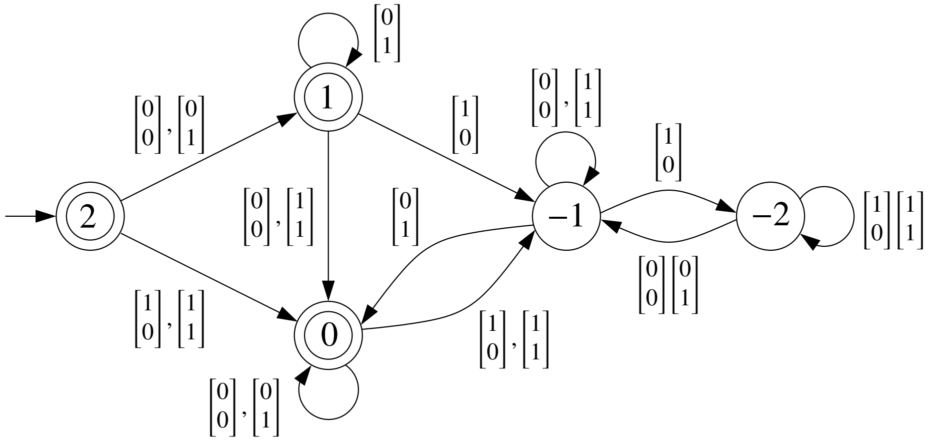

$\begin{bmatrix}0\\1\end{bmatrix}$
1
$\begin{bmatrix}0\\0\end{bmatrix},\begin{bmatrix}0\\1\end{bmatrix}$
$\begin{bmatrix}1\\0\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix},\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}1\\0\end{bmatrix}$
2
$\begin{bmatrix}0\\0\end{bmatrix},\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}0\\1\end{bmatrix}$
−1
−2
$\begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}$
$\begin{bmatrix}1\\0\end{bmatrix},\begin{bmatrix}1\\1\end{bmatrix}$
0
$\begin{bmatrix}1\\0\end{bmatrix},\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix},\begin{bmatrix}0\\1\end{bmatrix}$

Figure 10.1: DFA for the formula $2x - y \leq 2$.

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 encodes $x = 12$ and $y = 50$ and, indeed $24 - 50 \leq 2$. If we remove the last letter then the word encodes $x = 12$ and $y = 18$, and is not accepted, which indeed corresponds to $24 - 18 \nleq 2$.

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. Transitions leaving -4 are given by

$$\delta(-4, \zeta) = \left\lfloor \frac{1}{2}(-4 - (-1, -1) \cdot (\zeta_x, \zeta_y)) \right\rfloor = \left\lfloor \frac{1}{2}(-4 + \zeta_x + \zeta_y) \right\rfloor$$

and so we have $-4 \xrightarrow{[0,0]} -2$, $-4 \xrightarrow{[0,1]} -2$, $-4 \xrightarrow{[1,0]} -2$ and $-4 \xrightarrow{[1,1]} -1$. Notice that the DFA is not minimal, since sttaes 0 and 1 can be merged.

□

Partial correctness of *AFtoDFA* is easily proved by showing that for every $q \in \mathbb{Z}$ and every word $w \in (\{0, 1\}^n)^*$, the state q accepts w iff w encodes $c \in \mathbb{N}^n$ satisfying $a \cdot c \leq q$. The proof proceeds by induction of $|w|$. For $|w| = 0$ the result follows immediately from the definition of the final states, and for $|w| > 0$ from the fact that δ satisfies property 10.1 and from the induction hypothesis. Details are left to the reader. Termination of *AFtoDFA* also requires a proof: in principle the algorithm 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 let $s = \sum_{i=1}^{k} |a_i|$. All states s_j added to the workset during the execution of AFtoDFA(φ) satisfy*

$$-|b| - s \leq j \leq |b| + s.$$

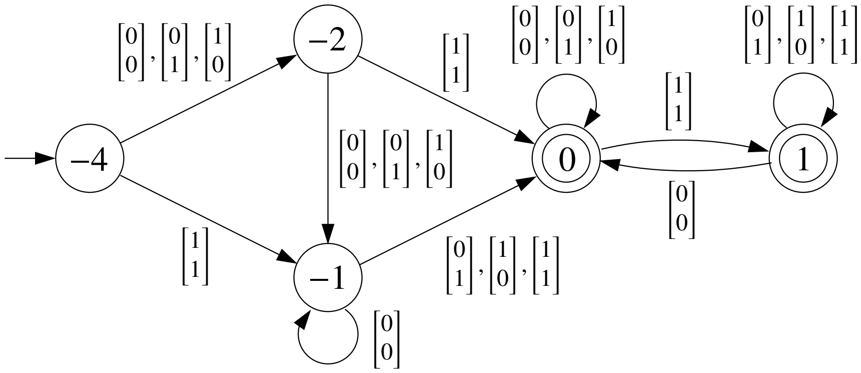

-4
-2
-1
0
1
$\begin{bmatrix}0\\0\end{bmatrix},\begin{bmatrix}0\\1\end{bmatrix},\begin{bmatrix}1\\0\end{bmatrix}$
$\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix},\begin{bmatrix}0\\1\end{bmatrix},\begin{bmatrix}1\\0\end{bmatrix}$
$\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}$
$\begin{bmatrix}0\\1\end{bmatrix},\begin{bmatrix}1\\0\end{bmatrix},\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix},\begin{bmatrix}0\\1\end{bmatrix},\begin{bmatrix}1\\0\end{bmatrix}$
$\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}$
$\begin{bmatrix}0\\1\end{bmatrix},\begin{bmatrix}1\\0\end{bmatrix},\begin{bmatrix}1\\1\end{bmatrix}$

Figure 10.2: DFA for the formula $x + y \geq 4$.

Proof: The property holds for s_b, the first state added to the workset. We show that, at any point in time, if all the states added to the workset 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 workset 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

$$\left\lfloor \frac{-|b| - s - a \cdot \zeta}{2} \right\rfloor \leq j \leq \left\lfloor \frac{|b| + s - a \cdot \zeta}{2} \right\rfloor \tag{10.2}$$

Now we manipulate the right and left ends of (10.2). A little arithmetic yields

$$\begin{array}{rcccl} -|b| - s & \leq & \dfrac{-|b| - 2s}{2} & \leq & \left\lfloor \dfrac{-|b| - s - a \cdot \zeta}{2} \right\rfloor \\ \left\lfloor \dfrac{|b| + s - a \cdot \zeta}{2} \right\rfloor & \leq & \dfrac{|b| + 2s}{2} & \leq & |b| + s \end{array}$$

which together with (10.2) leads to

$$-|b| - s \leq j \leq |b| + s$$

and we are done. □

Example 10.4 We compute all natural solutions of the system of linear 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

$$\exists z\, x = 4z \ \land\ \exists w\, y = 4w \ \land\ 2x - y \leq 2 \ \land\ x + y \geq 4$$

The minimal DFA for the first two conjuncts can be computed using projections and intersections, but the result is also easy to guess: it is the DFA of Figure 10.3 (where a trap state has been omitted).

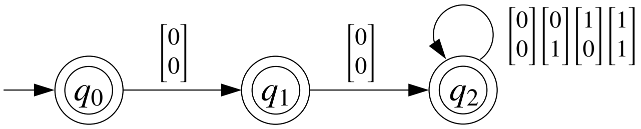

q_0
q_1
q_2
$\begin{bmatrix}0\\0\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}$

Figure 10.3: DFA for the formula $\exists z\, x = 4z \ \land\ \exists w\, y = 4w$.

The solutions are then represented by the intersection of the DFAs shown in Figures 10.1, 10.2 (after merging states 0 and 1), and 10.3. The result is shown in Figure 10.4. (Some states from which no final state can be reached are omitted.)

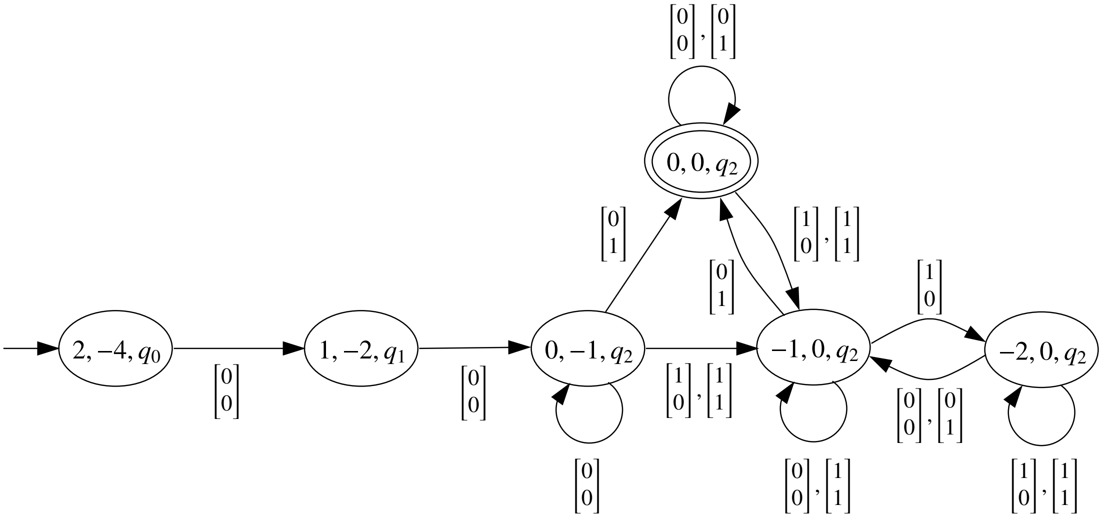

$\begin{bmatrix}0\\0\end{bmatrix},\begin{bmatrix}0\\1\end{bmatrix}$
$0, 0, q_2$
$\begin{bmatrix}0\\1\end{bmatrix}$
$\begin{bmatrix}0\\1\end{bmatrix}$
$\begin{bmatrix}1\\0\end{bmatrix},\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}1\\0\end{bmatrix}$
$2, -4, q_0$
$1, -2, q_1$
$0, -1, q_2$
$-1, 0, q_2$
$-2, 0, q_2$
$\begin{bmatrix}0\\0\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}$
$\begin{bmatrix}1\\0\end{bmatrix},\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix},\begin{bmatrix}0\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix},\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}1\\0\end{bmatrix},\begin{bmatrix}1\\1\end{bmatrix}$

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.

□

10.2.1 Equations

A slight modification of *AFtoDFA* directly constructs a DFA for the solutions of $a \cdot x = b$, without having to intersect DFAs for $a \cdot x \leq b$ and $-a \cdot x \leq -b$. The states of the DFA are a trap state q_t

accepting the empty language, plus integers satisfying:

$$\text{State } q \in \mathbb{Z} \text{ recognizes the encodings of the tuples } c \in \mathbb{N}^n \text{ such that } a \cdot c = q. \tag{10.3}$$

For the trap state q_t we take $\delta(q_t, \zeta) = q_t$ for every $\zeta \in \{0,1\}^n$. For a state $q \in \mathbb{Z}$ and a letter $\zeta \in \{0,1\}^n$ we determine the target state q' of transition $q \xrightarrow{\zeta} q'$. Given a tuple $c' \in \mathbb{N}^n$, property 10.3 requires $c' \in L(q')$ iff $a \cdot c' = q'$. As in the case of inequations, we have

$$\begin{array}{lll} & c' \in L(q') & \\ \text{iff} & 2c' + c_\zeta \in L(q) & \\ \text{iff} & a \cdot (2c' + \zeta) = q & \quad \text{(property 10.3 for } q\text{)} \\ \text{iff} & a \cdot c' = \frac{1}{2}(q - a \cdot \zeta) & \end{array}$$

If $q - a \cdot \zeta$ is odd, then, since $a \cdot c'$ is an integer, the equation $a \cdot c' = \frac{1}{2}(q - a \cdot \zeta)$ has no solution. So in order to satisfy property 10.3 we must choose q' satisfying $L(q') = \emptyset$, and so we take $q' = q_t$. If $q - a \cdot \zeta$ is even then we must choose q' satisfying $a \cdot c' = q'$, and so we take $q' = \frac{1}{2}(q - a \cdot \zeta)$. Therefore, the transition function of the DFA is given by:

$$\delta(q, \zeta) = \begin{cases} q_t & \text{if } q = q_t \text{ or } q - a \cdot \zeta \text{ is odd} \\ \frac{1}{2}(q - a \cdot \zeta) & \text{if } q - a \cdot \zeta \text{ is even} \end{cases}$$

For the final states, recall that a state is final iff it accepts the tuple $(0, \ldots, 0)$. So q_t is nonfinal and, by property 10.3, $q \in \mathbb{Z}$ is final iff $a \cdot (0 \ldots, 0) = q$. So the only final state is $q = 0$. The resulting algorithm is shown in Table 10.2. The algorithm does not construct the trap state.

Example 10.5 Consider the formulas $x + y \leq 4$ and $x + y = 4$. The result of applying *AFtoDFA* to $x + y \leq 4$ is shown at the top of Figure 10.5. Notice the similarities and differences with the DFA for $x + y \geq 4$ in Figure 10.2. The bottom part of the Figure shows the result of applying *EqtoDFA* to $x + y = 4$. Observe that the transitions are a subset of the transitions of the DFA for $x + y \leq 4$. This example shows that the DFA is not necessarily minimal, since state -1 can be deleted.

□

Partial correctness and termination of *EqtoDFA* are easily proved following similar steps to the case of inequations.

10.3 An NFA for the Solutions over the Integers.

We construct an NFA recognizing the encodings of the *integer solutions* (positive or negative) of a formula. In order to deal with negative numbers we use *2-complements*. A *2-complement encoding* of an integer $x \in \mathbb{Z}$ is any word $a_0 a_1 \ldots a_n$, where $n \geq 1$, satisfying

$$x = \sum_{i=0}^{n-1} a_i \cdot 2^i - a_n \cdot 2^n \tag{10.4}$$

```
$EqtoDFA(\varphi)$
Input: Equation $\varphi = \ a \cdot x = b$
Output:  DFA $A = (Q, \Sigma, \delta, q_0, F)$ such that $L(A) = L(\varphi)$
         (without trap state)
$Q, \delta, F \leftarrow \emptyset$; $q_0 \leftarrow s_b$
$W \leftarrow \{s_b\}$
while $W \neq \emptyset$ do
    pick $s_k$ from $W$
    add $s_k$ to $Q$
    if $k = 0$ then add $s_k$ to $F$
    for all $\zeta \in \{0,1\}^n$ do
        if $(k - a \cdot \zeta)$ is even then
            $j \leftarrow \frac{1}{2}(k - a \cdot \zeta)$
            if $s_j \notin Q$ then add $s_j$ to $W$
            add $(s_k, \zeta, s_j)$ to $\delta$
```

Table 10.2: Converting an equation into a DFA recognizing the *lsbf* encodings of its solutions.

We call a_n the *sign bit*. For example, 110 encodes $1 + 2 - 0 = 3$, and 111 encodes $1 + 2 - 4 = -1$. If the word has length 1 then its only bit is the sign bit; in particular, the word 0 encodes the number 0, and the word 1 encodes the number -1. The empty word encodes no number. Observe that all of $110, 1100, 11000, \ldots$ encode 3, and all of $1, 11, 111, \ldots$ encode -1. In general, it is easy to see that all words of the regular expression $a_0 \ldots a_{n-1} a_n a_n^*$ encode the same number: for $a_n = 0$ this is obvious, and for $a_n = 1$ both $a_0 \ldots a_{n-1} 1$ and $a_0 \ldots a_{n-1} 1 1^m$ encode the same number because

$$-2^{m+n} + 2^{m-1+n} + \ldots + 2^{n+1} = 2^n \ .$$

This property allows us to encode tuples of numbers using padding. Instead of padding with 0, we pad with the sign bit.

Example 10.6 The triple $(12, -3, -14)$ is encoded by all the words of

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

For example, the triples (x, y, z) and $(x', y'z')$ encoded by

$$\begin{bmatrix}0\\1\\0\end{bmatrix}\begin{bmatrix}0\\0\\1\end{bmatrix}\begin{bmatrix}1\\1\\0\end{bmatrix}\begin{bmatrix}1\\1\\0\end{bmatrix}\begin{bmatrix}0\\1\\1\end{bmatrix} \qquad \text{and} \qquad \begin{bmatrix}0\\1\\0\end{bmatrix}\begin{bmatrix}0\\0\\1\end{bmatrix}\begin{bmatrix}1\\1\\0\end{bmatrix}\begin{bmatrix}1\\1\\0\end{bmatrix}\begin{bmatrix}0\\1\\1\end{bmatrix}\begin{bmatrix}0\\1\\1\end{bmatrix}\begin{bmatrix}0\\1\\1\end{bmatrix}$$

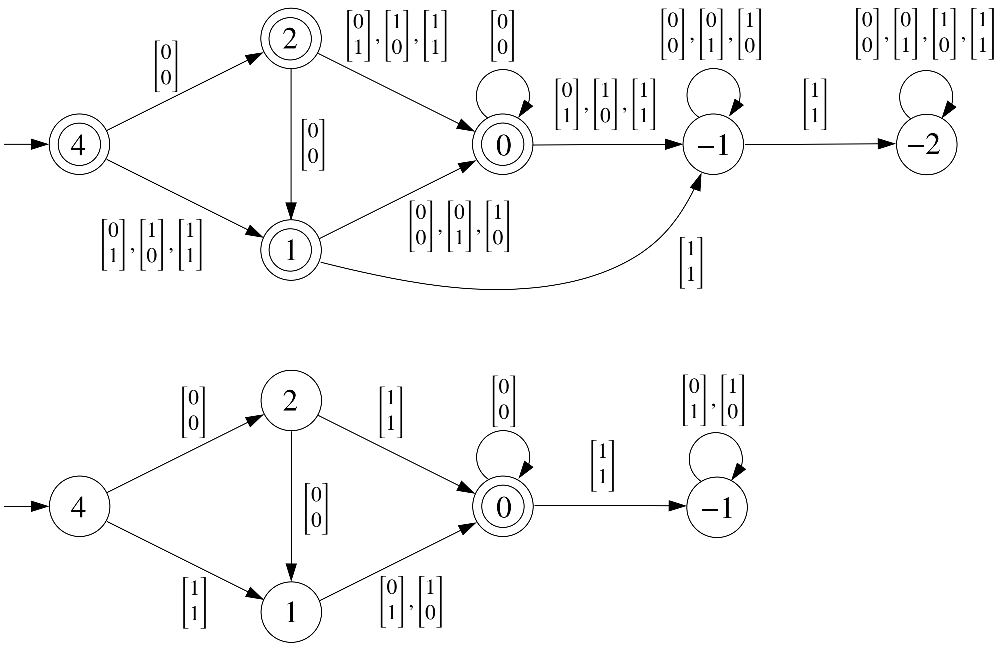

4
2
1
0
−1
−2
$\begin{bmatrix}0\\0\end{bmatrix}$
$\begin{bmatrix}0\\1\end{bmatrix},\begin{bmatrix}1\\0\end{bmatrix},\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix},\begin{bmatrix}0\\1\end{bmatrix},\begin{bmatrix}1\\0\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix},\begin{bmatrix}0\\1\end{bmatrix},\begin{bmatrix}1\\0\end{bmatrix},\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}0\\1\end{bmatrix},\begin{bmatrix}1\\0\end{bmatrix}$

Figure 10.5: DFAs for the formulas $x + y \leq 4$ and $x + y = 4$.

are given by

$$\begin{array}{rcrcr@{\qquad}rcrcr}
x & = & 0+0+4+8+0 & = & 12 & x' & = & 0+0+4+8+0+0+0+0 & = & 12\\
y & = & 1+0+4+8-16 & = & -3 & y' & = & 1+0+4+8+16+32-64 & = & -3\\
z & = & 0+2+0+0-16 & = & -14 & z' & = & 0+2+0+0+16+32-64 & = & -14
\end{array}$$

□

We construct a NFA (no longer a DFA!) recognizing the integer solutions of an atomic formula $a \cdot x \leq b$. As usual we take integers for the states, and the NFA should satisfy:

$$\text{State } q \in \mathbb{Z} \text{ recognizes the encodings of the tuples } c \in \mathbb{Z}^n \text{ such that } a \cdot c \leq q. \tag{10.5}$$

However, integer states are no longer enough, because no state $q \in \mathbb{Z}$ can be final: in the 2-complement encoding the empty word encodes no number, and so, since q cannot accept the empty word by property 10.5, q must be nonfinal. But we need at least one final state, and so we add to the NFA a unique final state q_f without any outgoing transitions, accepting only the empty word.

Given a state $q \in \mathbb{Z}$ and a letter $\zeta \in \{0, 1\}^n$, we determine the targets q' of the transitions $q \xrightarrow{\zeta} q'$ of the NFA, where $\zeta \in \{0, 1\}^n$. (As we will see, the NFA may have one or two such transitions.)

A word $w' \in (\{0,1\}^n)^*$ is accepted from some target state q' iff $\zeta w'$ is accepted from q. In the 2-complement encoding there are two cases:

(1) If $w' \neq \epsilon$, then $\zeta w'$ encodes the tuple of integers $2c' + \zeta$, where $c' \in \mathbb{Z}^n$ is the tuple encoded by w'. (This follows easily from the definition of 2-complements.)

(2) If $w' = \epsilon$, then $\zeta w'$ encodes the tuple of integers $-\zeta$, because in this case ζ is the sign bit.

In case (1), property 10.5 requires a target state q' such that $a \cdot c \leq q$ iff $a \cdot (2c + \zeta) \leq q'$. So we take

$$q' = \left\lfloor \frac{1}{2}(q - a \cdot \zeta) \right\rfloor$$

For case (2) property 10.5 only requires a target state q' if $a \cdot (-\zeta) \leq q$, and if so then it requires q' to be a *final* state. So if $q + a \cdot \zeta \geq 0$ then we add $q \xrightarrow{\zeta} q_f$ to the set of transitions, and in this case the automaton has two transitions leaving state q and labeled by ζ. Summarizing, we define the transition function of the NFA by

$$\delta(q, \zeta) = \begin{cases} \left\{ \left\lfloor \frac{1}{2}(q - a \cdot \zeta) \right\rfloor, q_f \right\} & \text{if } q + a \cdot \zeta \geq 0 \\ \left\{ \left\lfloor \frac{1}{2}(q - a \cdot \zeta) \right\rfloor \right\} & \text{otherwise} \end{cases}$$

Observe that the NFA contains all the states and transitions of the DFA for the natural solutions of $a \cdot x \leq b$, plus possibly other transitions. All integer states are now nonfinal, the only final state is q_f.

Example 10.7 Figure 10.6 shows at the top the NFA recognizing all integer solutions of $2x - y \leq 2$. It has all states and transitions of the DFA for the natural solutions, plus some more (compare with Figure 10.1). The final state q_f and the transitions leading to it are drawn in red. Consider for instance state -1. In order to determine the letters $\zeta \in \{0,1\}^2$ for which $q_f \in \delta(-1, \zeta)$, we compute $q + a \cdot \zeta = -1 + 2\zeta_x - \zeta_y$ for each $(\zeta_x, \zeta_y) \in \{0,1\}^2$, and compare the result to 0. We obtain that the letters leading to q_f are $(1,0)$ and $(1,1)$. □

10.3.1 Equations

If order to construct an NFA for the integer solutions of an equation $a \cdot x = b$ we can proceed as for inequations. The result is again an NFA containing all states and transitions of the DFA for the natural solutions computed in Section 10.2.1, plus possible some more. The automaton has an additional final state q_f, and a transition $q \xrightarrow{\zeta} q_f$ iff $q + a \cdot \zeta = 0$. Graphically, we can also obtain the NFA by starting with the NFA for $a \cdot x \leq b$, and removing all transitions $q \xrightarrow{\zeta} q'$ such that $q' \neq \frac{1}{2}(q - a \cdot \zeta)$, and all transitions $q \xrightarrow{\zeta} q_f$ such that $q + a \cdot \zeta \neq 0$.

Example 10.8 The NFA for the integer solutions of $2x - y = 2$ is shown in the middle of Figure 10.6. Its transitions are a subset of those of the NFA for $2x - y \leq 2$. □

The NFA for the integer solutions of an equation has an interesting property. Since $q + a \cdot \zeta = 0$ holds iff $\frac{1}{2}(q + a \cdot \zeta) = \frac{1}{2}(2q) = q$, the NFA has a transition $q \xrightarrow{\zeta} q_f$ iff it also has a self-loop $q \xrightarrow{\zeta} q$. (For instance, state 1 of the DFA in the middle of Figure 10.6 has a red transition labeled by $(0, 1)$ and a self-loop labeled by $(0, 1)$.) Using this property it is not difficult to see that the powerset construction does not cause a blowup in the number of states: it only adds one extra state for each predecessor of the final state.

Example 10.9 The DFA obtained by applying the powerset construction to the NFA for $2x - y = 2$ is shown at the bottom of Figure 10.6 (the trap state has been omitted). Each of the three predecessors of q_f gets "duplicated". □

Moreover, the DFA obtained by means of the powerset construction is *minimal*. This can be proved by showing that any two states recognize different languages. If exactly one of the states is final, we are done. If both states are nonfinal, say, k and k', then they recognize the solutions of $a \cdot x = k$ and $a \cdot x = k'$, and so their languages are not only distinct but even disjoint. If both states are final, then they are the "duplicates" of two nonfinal states k and k', and their languages are those of k and k', plus the empty word. So, again, their languages are distinct.

10.3.2 Algorithms

The algorithms for the construction of the NFAs are shown in Table 10.3. Additions to the previous algorithms are shown in blue.

Exercises

Exercise 114 t $r \geq 0, n \geq 1$. Give a Presburger formula φ such that $\mathcal{J} \models \varphi$ if, and only if, $\mathcal{J}(x) \geq \mathcal{J}(y)$ and $\mathcal{J}(x) - \mathcal{J}(y) \equiv r \pmod{n}$. Give an automaton that accepts the solutions of φ for $r = 0$ and $n = 2$.

Exercise 115 Construct a finite automaton for the Presburger formula $\exists y\ x = 3y$ using the algorithms of the chapter.

Exercise 116 *AFtoDFA* returns a DFA recognizing all solutions of a given linear inequation

$$a_1x_1 + a_2x_2 + \ldots + a_kx_k \leq b \text{ with } a_1, a_2, \ldots, a_k, b \in \mathbb{Z} \qquad (*)$$

encoded using the *lsbf* encoding of $\mathbb{N}^k$. We may also use the most-significant-bit-first (*msbf*) encoding, e.g.,

$$\text{msbf}\left(\begin{bmatrix} 2 \\ 3 \end{bmatrix}\right) = \mathcal{L}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}^* \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix}\right)$$

```
IneqZtoNFA($\varphi$)
Input: Inequation $\varphi = a \cdot x \leq b$ over $\mathbb{Z}$
Output: NFA $A = (Q, \Sigma, \delta, Q_0, F)$ such that
        $L(A) = L(\varphi)$

$Q, \delta, F \leftarrow \emptyset$; $Q_0 \leftarrow \{s_b\}$
$W \leftarrow \{s_b\}$
while $W \neq \emptyset$ do
    pick $s_k$ from $W$
    add $s_k$ to $Q$
    for all $\zeta \in \{0,1\}^n$ do
        $j \leftarrow \left\lfloor \frac{1}{2}(k - a \cdot \zeta) \right\rfloor$
        if $s_j \notin Q$ then add $s_j$ to $W$
        add $(s_k, \zeta, s_j)$ to $\delta$
        $j' \leftarrow k + a \cdot \zeta$
        if $j' \geq 0$ then
            add $q_f$ to $Q$ and $F$
            add $(s_k, \zeta, q_f)$ to $\delta$
```

```
EqZtoNFA($\varphi$)
Input: Equation $\varphi = a \cdot x = b$ over $\mathbb{Z}$
Output: NFA $A = (Q, \Sigma, \delta, Q_0, F)$ such that
        $L(A) = L(\varphi)$

$Q, \delta, F \leftarrow \emptyset$; $Q_0 \leftarrow \{s_b\}$
$W \leftarrow \{s_b\}$
while $W \neq \emptyset$ do
    pick $s_k$ from $W$
    add $s_k$ to $Q$
    for all $\zeta \in \{0,1\}^n$ do
        if $(k - a \cdot \zeta)$ is even then
            if $(k + a \cdot \zeta) = 0$ then add $k$ to $F$
            $j \leftarrow \frac{1}{2}(k - a \cdot \zeta)$
            if $s_j \notin Q$ then add $s_j$ to $W$
            add $(s_k, \zeta, s_j)$ to $\delta$
            $j' \leftarrow k + a \cdot \zeta$
            if $j' \geq 0$ then
                add $q_f$ to $Q$ and $F$
                add $(s_k, \zeta, q_f)$ to $\delta$
```

Table 10.3: Converting an inequality into a NFA accepting the 2-complement encoding of the solution space.

1. Construct a finite automaton for the inequation $2x - y \leq 2$ w.r.t. *msbf* encoding.

2. Adapt *AFtoDFA* to the msbf encoding.

Exercise 117 Consider the extension of FO(Σ) where addition of variables is allowed. Give a sentence of this logic for palindromes over $\{a, b\}$, i.e. $\{w \in \{a, b\}^* : w = w^R\}$.

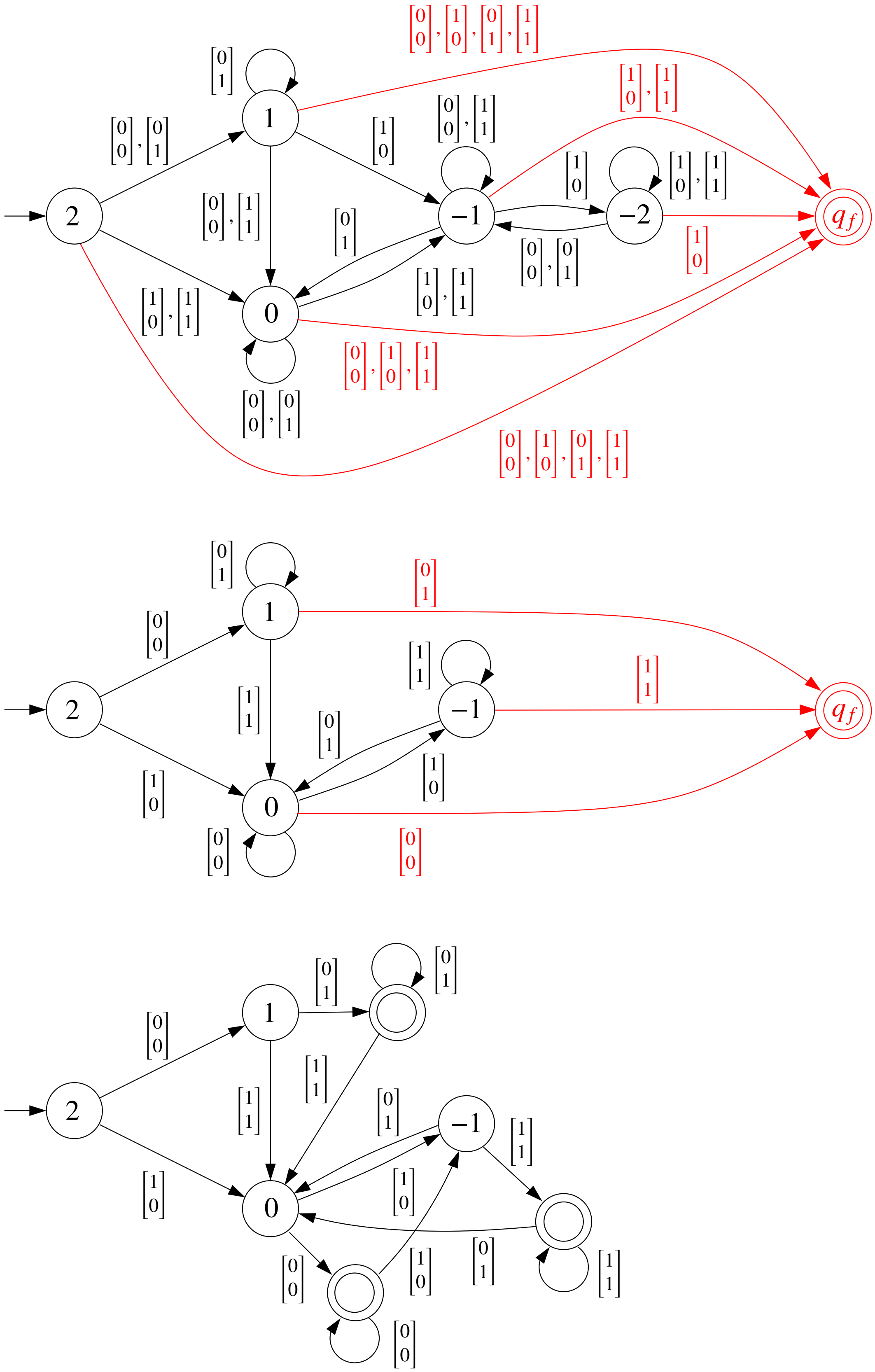


Figure 10.6: NFAs for the solutions of $2x - y \leq 2$ and $2x - y = 2$ over $\mathbb{Z}$, and minimal DFA for the solutions of $2x - y = 2$.

Part II

Automata on Infinite Words

Chapter 11

Classes of ω-Automata and Conversions

Automata on infinite words, also called ω-automata in this book, were introduced in the 1960s as an auxiliary tool for proving the decidability of some problems in mathematical logic. As the name indicates, they are automata whose input is a word of infinite length. The run of an automaton on a word typically is not expected to terminate.

Even a deterministic ω-automaton makes little sense as a language acceptor that decides if a word has a property or not: Not many people are willing to wait infinitely long to get an answer to a question! However, ω-automata still make perfect sense as a data structure, that is, as a finite representation of a (possibly infinite) set of infinite words.

There are objects that can only be represented as infinite words. The example that first comes to mind are the real numbers. A second example, more relevant for applications, are program executions. Programs may have nonterminating executions, either because of programming errors, or because they are designed this way. Indeed, many programs whose purpose is to keep a system running, like routines of an operating systems, network infrastructure, communication protocols, etc., are designed to be in constant operation. Automata on infinite words can be used to finitely represent the set of executions of a program, or an abstraction of it. They are an important tool for the theory and practice of program verification.

In the second part of this book we develop the theory of ω-automata as a data structure for languages of inifnite words. This first chapter introduces ω-regular expressions, a textual notation for defining languages of infinite words, and then proceeds to present different classes of automata on infinite words, most of them with the same expressive power as ω-regular expressions, and conversion algorithms between them.

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 $w_1 \cdot w_2$. 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 the ω-language $L_1 \cdot L_2 = \{w_1 w_2 \in \Sigma^\omega \mid w_1 \in L_1 \text{ and } w_2 \in L_2\}$. The *$\omega$-iteration* of a language $L \subseteq \Sigma^*$ is the ω-language $L^\omega = \{w_1 w_2 w_3 \ldots \mid w_i \in L \setminus \{\epsilon\}\}$. Observe that $\{\emptyset\}^\omega = \emptyset$, in contrast to the case of finite words, where $\{\emptyset\}^* = \{\epsilon\}$.

We extend regular expressions to ω-regular expressions, a formalism to define ω-languages.

Definition 11.1 *The ω-regular expressions over an alphabet Σ are defined by the following grammar, where $r \in \mathcal{RE}(\Sigma)$ is a regular expression*

$$s ::= r^\omega \mid r s_1 \mid s_1 + s_2$$

Sometimes we write $r \cdot s_1$ instead of $r s_1$. The set of all ω-regular expressions over Σ is denoted by $\mathcal{RE}_\omega(\Sigma)$. The language $L_\omega(s) \subseteq \Sigma$ *of an ω-regular expression $s \in \mathcal{RE}_\omega(\Sigma)$ is defined inductively as*

- $L_\omega(r^\omega) = (L(r))^\omega$*;*
- $L_\omega(r s_1) = L(r) \cdot L_\omega(s_1)$*; 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 $L = L_\omega(s)$.*

Observe that the empty ω-language is ω-regular because $L_\omega(\emptyset^\omega) = \emptyset$. As for regular expressions, we often identify an ω-regular expression s and its associated ω-language $L_\omega(s)$.

Example 11.2 The ω-regular expression $(a + b)^\omega$ denotes the language of all ω-words over a and b; $(a + b)^* b^\omega$ denotes the language of all ω-words over $\{a, b\}$ containing only finitely many as, and $(a^* ab + b^* ba)^\omega$ the language of all ω-words over $\{a, b\}$ containing infinitely many as and infinitely many bs; an even shorter expression for this latter language is $((a + b)^* ab)^\omega$. □

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 $w = a_0 a_1 a_2 \ldots$ over Σ. Intuitively, a run of A on w never terminates, and so we cannot define acceptance in terms of the state reached at the end of the run. In fact, even the name "final state" is no longer appropriate for Büchi automata. So from now on we speak of "accepting states", although we still denote the set of accepting states by F. We say that a run of a Büchi automaton is accepting if some accepting state is visited along the run *infinitely often*. Since the set of accepting states is finite, "some accepting state is visited infinitely often" is equivalent to "the set of accepting states is infinitely often".

Definition 11.3 *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 $a_0a_1a_2 \ldots \in \Sigma^\omega$ is an infinite sequence $\rho = q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} q_2 \ldots$, such that $q_i \in Q$ for $0 \leq i \leq n$, $q_0 \in Q_0$ and $q_{i+1} \in \delta(q_i, a_i)$ for every $0 \leq i$.*

Let $\inf(\rho)$ be the set $\{q \in Q \mid q = q_i$ for infinitely many i's$\}$, i.e., the set of states that occur in ρ infinitely often. A run ρ is accepting *if $\inf(\rho) \cap F \neq \emptyset$. A NBA* accepts *an ω-word $w \in \Sigma^\omega$ if it has an accepting run on w. The* language recognized *by a NBA A is the set $L_\omega(A) = \{w \in \Sigma^\omega \mid w$ is accepted by $A\}$.*

The condition $\inf(\rho) \cap F \neq \emptyset$ on runs is called the *the Büchi condition F*. In later sections we introduce other kinds of accepting conditions.

A Büchi automaton is deterministic if it is deterministic when seen as an automaton on finite words. NBAs with ϵ-transitions can also be defined, but we will not need them.[1]

Example 11.4 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 as. So, for instance, A accepts a^ω, ba^ω, $(ab)^\omega$, or $(ab^{100})^\omega$, but not b^ω or $a^{100}b^\omega$. To prove that this is indeed the language we 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, if w is 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, and so w contains infinitely many as.

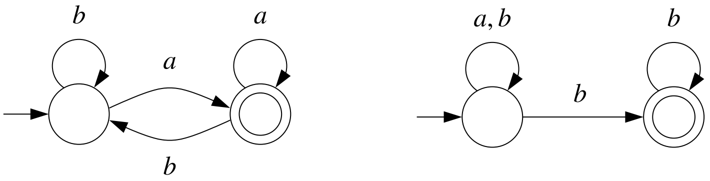


Figure 11.1: Two Büchi automata

The automaton on the right of the figure is not deterministic, and recognizes all ω-words over the alphabet $\{a, b\}$ that contain *finitely many* occurrences of a. The proof is similar. □

Example 11.5 Figure 11.2 shows three further Büchi automata over the alphabet $\{a, b, c\}$. The top-left automaton recognizes the ω-words in which for every occurrence of a there is a later occurrence

[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.

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.

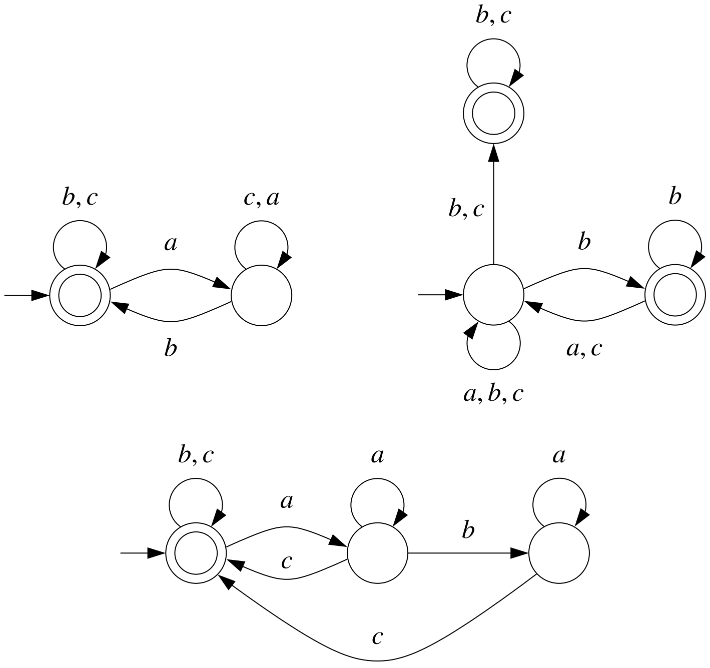


Figure 11.2: Three further Büchi automata

□

11.2.1 From ω-regular expressions to NBAs and back

We present algorithms for converting an ω-regular expression into a NBA, and vice versa. This provides a first "sanity check" for NBAs as data structure, by showing that NBAs can represent exactly the ω-regular languages.

From ω-regular expressions to NBAs. We give a procedure that transforms an ω-regular expression into an equivalent NBA with exactly one initial state, which moreover has no incoming transitions.

We proceed by induction on the structure of the ω-regular expression. Recall that for every regular expression r we can construct an NFA A_r with a unique initial state, a unique final state, no transition leading to the initial state, and no transition leaving the final state. An NBA for r^ω is obtained by adding to A_r new transitions leading from the final state to the targets of the transitions leaving the initial state, as shown at the top of Figure **??**. An NBA for $r \cdot s$ is obtained by merging states as shown in the middle of the figure. Finally, an NBA for $s_1 + s_2$ is obtained by merging the initial states of the NBAs for s_1 and s_2 as shown at the bottom.

From NBAs to ω-regular expressions. Let $A = (Q, \Sigma, \delta, Q_0, F)$ be a NBA. For every two states $q, q' \in Q$, let $A_q^{q'}$ be the NFA (not the NBA!) obtained from A by changing the set of initial states to $\{q\}$ and the set of final states to $\{q'\}$. Using algorithm *NFAtoRE* we can construct a regular expression denoting $L\left(A_q^{q'}\right)$. By slightly modifying $A_q^{q'}$ we can also onstruct a regular expression $r_q^{q'}$ denoting the words accepted by $L\left(A_q^{q'}\right)$ by means of runs that visit q' exactly once (how to do this is left as a little exercise). We use these expressions to compute an ω-regular expression denoting $L_\omega(A)$.

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 A 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 $w_1 \in L\left(r_{q_0}^q\right)$, and $w_i \in L\left(r_q^q\right)$ for every $i > 1$. So we have $L_q = L_\omega\left(r_{q_0}^q \left(r_q^q\right)^\omega\right)$, and therefore

$$\sum_{q \in F} r_{q_0}^q \left(r_q^q\right)^\omega$$

is the ω-regular expression we are looking for.

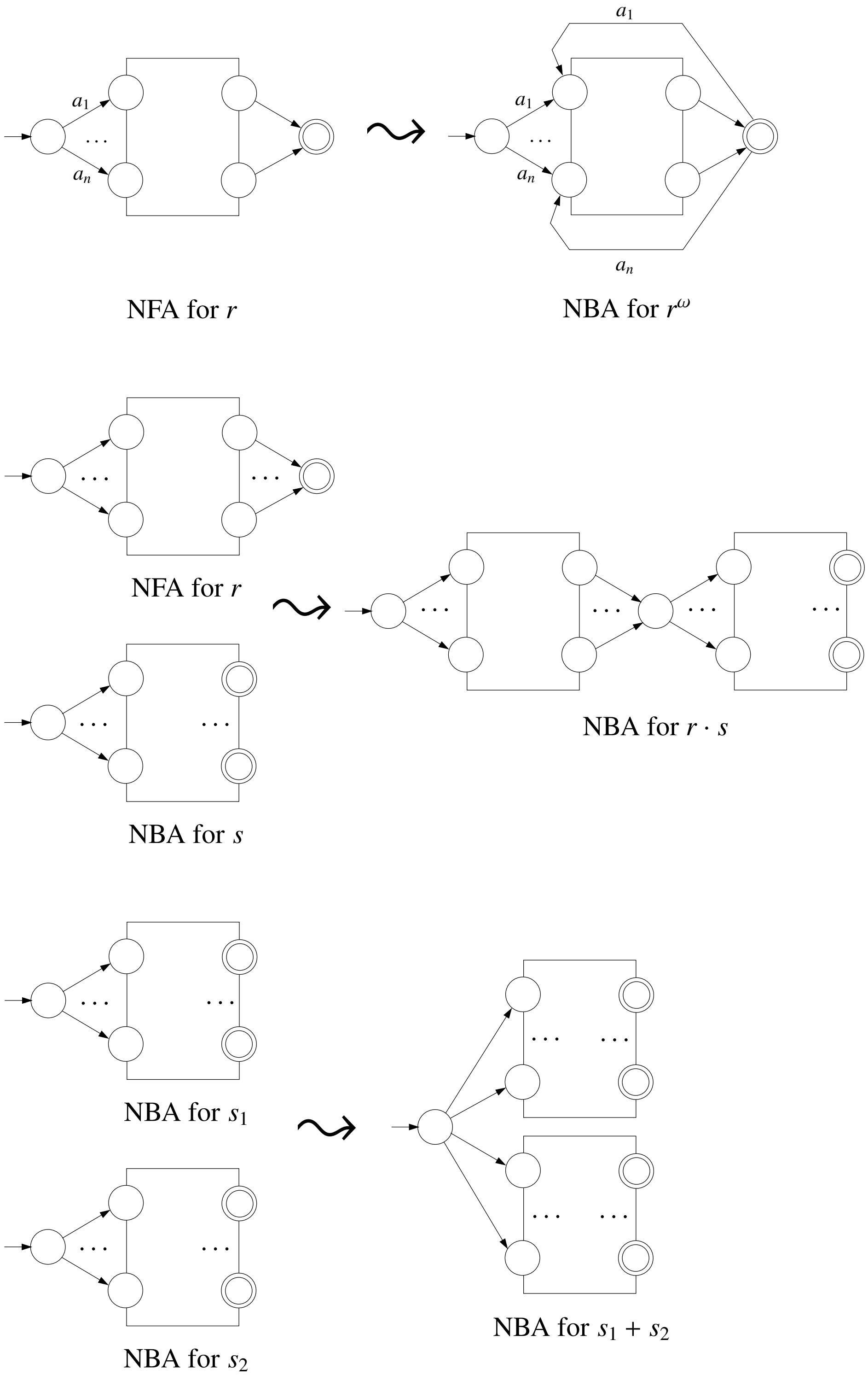
Example 11.6 Consider the top right NBA of Figure 11.2. We have to compute $r_0^1\left(r_1^1\right)^\omega + r_0^2\left(r_2^2\right)^\omega$. Using *NFAtoRE* and simplifying we get

$$\begin{array}{rcl} r_0^1 & = & (a+b+c)^*(b+c) \\ r_0^2 & = & (a+b+c)^*b \\ r_1^1 & = & (b+c) \\ r_2^2 & = & b + (a+c)(a+b+c)^*b \end{array}$$

and (after some further simplifications) we obtain the ω-regular expression

$$(a+b+c)^*(b+c)^\omega + (a+b+c)^*b\left(b + (a+c)(a+b+c)^*b\right)^\omega$$

. □



NFA for r

NBA for r^ω

NFA for r

NBA for s

NBA for $r \cdot s$

NBA for s_1

NBA for s_2

NBA for $s_1 + s_2$

Figure 11.3: From ω-regular expressions to Büchi automata

11.2.2 Non-equivalence of NBAs and DBAs

Unfortunately, DBAs do not recognize all ω-regular languages, and so they do not have the same expressive power as NBAs. We show that the language of ω-words containing finitely many occurrences of a is not recognized by any DBA. Intuitively, the NBA for this language "guesses" the last occurrence of a, and this guess cannot be determinized using only a finite number of states.

Proposition 11.7 *The language $L = (a+b)^* b^\omega$ (i.e., the language of all ω-words in which a occurs only finitely often) is not recognized by any DBA.*

Proof: Assume that $L = L_\omega(A)$ for a DBA $A = (\{a,b\}, Q, q_0, \delta, F)$, and define $\hat{\delta} : Q \times \{a,b\}^* \to Q$ by $\hat{\delta}(q,\epsilon) = q$ and $\hat{\delta}(q,wa) = \delta(\hat{\delta}(q,w),a)$. That is, $\hat{\delta}(q,w)$ denotes the unique state reached by reading w from state q. Consider the ω-word $w_0 = b^\omega$. Since $w_0 \in L$, the run of A on w_0 is accepting, and so $\hat{\delta}(q_0,u_0) \in F$ for some finite prefix u_0 of w_0. Consider now $w_1 = u_0\, a\, b^\omega$. We have $w_1 \in L$, and so teh run of A on w_1 is accepting, which implies $\hat{\delta}(q_0, u_0\, a\, u_1) \in F$ for some finite prefix $u_0\, a\, u_1$ of w_1. In a similar fashion we continue constructing finite words u_i such that $\hat{\delta}(q_0, u_0\, a\, u_1\, a \ldots a\, u_i) \in F$. Since Q is finite, there are indices $0 \le i < j$ such that $\hat{\delta}(q_0, u_0\, a \ldots u_i) = \hat{\delta}(q_0, u_0\, a \ldots u_i\, a \ldots a\, u_j)$. It follows that A has an accepting run on

$$u_0\, a \ldots u_i\, (a\, u_{i+1} \ldots a\, u_j)^\omega\ .$$

But a occurs infinitely often in this word, and so the word does not belong to L. □

Note that $\overline{L} = ((a+b)^* a)^\omega$ (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 convenient for implementing some operations, like for instance intersection. A *generalized Büchi automaton* (NGA) differs from a Büchi automaton in its accepting condition. Instead of a set F of accepting states, a NGA has a collection of sets of accepting states $\mathcal{F} = \{F_0, \ldots, F_{m-1}\}$. 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 translation is based on the following observation: a run ρ visits each set of $\mathcal{F}$ infinitely 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 of $\mathcal{F}$.)

This suggests to take for the NBA m "copies" of the NGA, but with a modification: the NBA "jumps" from the i-th to the $i \oplus 1$-th copy whenever it visits a state of F_i. More precisely, the transitions of the i-th copy that leave a state of F_i are redirected from the i-th copy to the $(i \oplus 1)$-th 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.

More formally, 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. If $q \notin F_i$ then the successors of $[q, i]$ are states of the i-th copy, and otherwise states of the $(i \oplus 1)$-th copy.

The pseudocode for the conversion algorithm is as follows:

```
NGAtoNBA(A)
Input: NGA $A = (Q, \Sigma, Q_0, \delta, \mathcal{F})$, where $\mathcal{F} = \{F_0, \ldots, F_{m-1}\}$
Output: NBA $A' = (Q', \Sigma, \delta', Q'_0, F')$
Q', δ', F' ← ∅; $Q'_0 \leftarrow \{[q_0, 0] \mid q_0 \in Q_0\}$
$W \leftarrow Q'_0$
while $W \neq \emptyset$ do
   pick $[q, i]$ from $W$
   add $[q, i]$ to $Q'$
   if $q \in F_0$ and $i = 0$ then add $[q, i]$ to $F'$
   for all $a \in \Sigma, q' \in \delta(q, a)$ do
      if $q \notin F_i$ then
         if $[q', i] \notin Q'$ then add $[q', i]$ to $W$
         add $([q, i], a, [q', i])$ to $\delta'$
      else /* $q \in F_i$ */
         if $[q', i \oplus 1] \notin Q'$ then add $[q', i \oplus 1]$ to $W$
         add $([q, i], a, [q', i \oplus 1])$ to $\delta'$
return $(Q', \Sigma, \delta', Q'_0, F')$
```

Example 11.8 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 $F_1 = \{r\}$, and so its accepting runs are those that visit *both* q and r infinitely often . It is easy to see that the automaton recognizes the ω-words containing infinitely many occurrences of *a and* infinitely many occurrences of *b*.

The NBA on the right consists of two copies of the NGA: the 0-th copy (pink) and the 1-st copy (blue). Transitions leaving $[q, 0]$ are redirected to the blue copy, and transitions leaving $[r, 1]$ are redirected to the pink copy. The only accepting state is $[q, 0]$. □

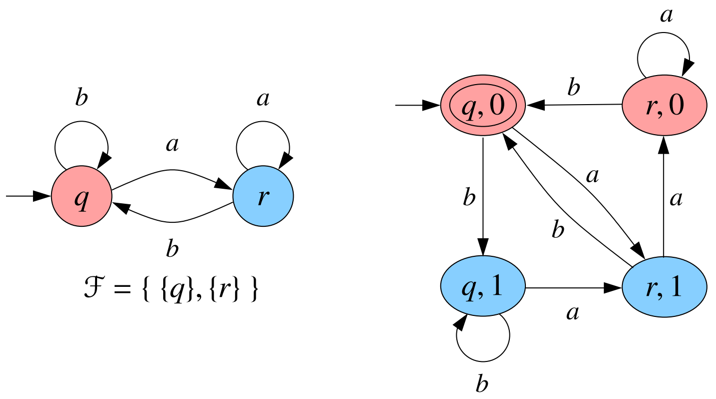


Figure 11.4: A NGA and its corresponding NBA

11.4 Other classes of ω-automata

Since not every NBA is equivalent to a DBA, there is no determinization procedure for Büchi automata. This raises the question whether such a procedure exists for other classes of automata. We shall see that the answer is yes, but the simplest determinizable classes 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

Like a Büchi automaton, a *(nondeterministic) co-Büchi automaton* (NCA) has a set F of accepting states. However, a run ρ of a NCA is accepting if it only visits states of F finitely 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").

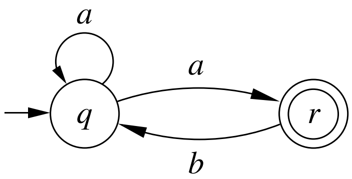


Figure 11.5: Running example for the determinization procedure

We show that co-Büchi automata can be determinized. We fix an NCA $A = (Q, \Sigma, \delta, Q_0, F)$ with n states, and, using Figure 11.5 as running example, construct an equivalent DCA B in three steps:

1. We define a mapping *dag* that assigns to each $w \in \Sigma^\omega$ a directed acyclic graph $dag(w)$.

2. We prove that w is rejected by A iff $dag(w)$ contains only finitely many *breakpoints*.

3. We construct a DCA B which accepts w if and only if $dag(w)$ contains finitely many breakpoints.

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, for $w = \sigma_1\sigma_2\ldots$ 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:

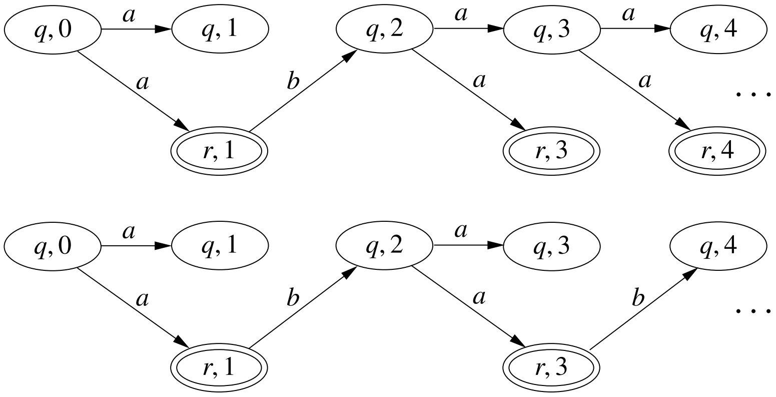


Figure 11.6: The (initial parts of) $dag(aba^\omega)$ and $dag((ab)^\omega)$

- $dag(w)$ contains a node $\langle q, 0\rangle$ for every initial state $q \in Q_0$.

- 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 $\langle q, i\rangle \xrightarrow{\sigma_{i+1}} \langle q', i+1\rangle$.

- $dag(w)$ 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*, with the i-th level containing all nodes of $dag(w)$ of the form $\langle q, i\rangle$.

One could be tempted to think that the accepting condition "some path of $dag(w)$ only visits accepting states finitely often" is equivalent to "only finitely many levels of $dag(w)$ contain accepting states", but $dag(aba^\omega)$ shows this is false: Even though all paths of $dag(aba^\omega)$ visit accepting states only finitely often, infinitely many levels (in fact, all all levels $i \geq 3$) contain accepting states. For this reason we introduce the set of *breakpoint levels* of the graph $dag(w)$, inductively defined as follows:

- The 0-th 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' (excluding nodes of l and including nodes of l') visits an accepting state is also a breakpoint.

We claim that "some path of $dag(w)$ only visits accepting states finitely often" is equivalent to "the set of breakpoint levels of $dag(w)$ is finite". If the breakpoint set is infinite, then by König's Lemma $dag(w)$ contains at least an infinite path, and moreover all infinite paths visit accepting states infinitely often. If the breakpoint set is finite, let i be the largest breakpoint. If $dag(w)$ is finite, 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. By König's lemma, this graph contains an infinite path π that never visits any accepting state, and we are done.

If we were able to tell that a level is a breakpoint by just examining it, we would be done: We would take the set of all possible levels as states of the DCA (i.e., the powerset of Q, as in the powerset construction for determinization of NFAs), the possible transitions between levels as transitions, and the breakpoints 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 breakpoints, as required by the co-Büchi acceptance condition. However, the level does not contain enough information for that. The solution is to add information to the states. We take for the states of the DCA pairs $[P, O]$, where $O \subseteq P \subseteq Q$, with the following intended meaning: P is the set of states of a level, and $q \in O$ iff q is the endpoint of some path, starting at the last breakpoint, that has not yet visited any accepting state(meaning that no edge of the path leads to a final state). We call O the set of *owing* states (states that "owe" a visit to the accepting states). To guarantee that O indeed has this intended meaning, we define the DCA $B = (\tilde{Q}, \Sigma, \tilde{\delta}, \tilde{q}_0, \tilde{F})$ as follows:

- The initial state is the pair $[\{q_0\}, \emptyset]$.

- The transition relation is given by $\tilde{\delta}([P, O], a) = [P', O']$, where $P' = \delta(P, a)$, and
  - if $O \neq \emptyset$, then $O' = \delta(O, a) \setminus F$;
  - if $O = \emptyset$, (i.e., if the current level is a breakpoint, and the automaton must start searching for the next one) then $O' = \delta(P, a) \setminus F$; in other words, all non-final states of the next level become owing.

- The accepting states are those at which a breakpoint is reached, i.e. $[P, O] \in \tilde{F}$ is accepting iff $O = \emptyset$.

With this definition, a run is accepting iff it contains infinitely many breakpoints. The algorithm for the construction is

```
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 $L_\omega(A) = \overline{B}$
$\tilde{Q}, \tilde{\delta}, \tilde{F} \leftarrow \emptyset$; if $q_0 \in F$ then $\tilde{q}_0 \leftarrow [\{q_0\}, \emptyset]$ else $\tilde{q}_0 \leftarrow [\{q_0\}, \{q_0\}]$
$W \leftarrow \{\ \tilde{q}_0\ \}$
while $W \neq \emptyset$ do
   pick $[P, O]$ from $W$; add $[P, O]$ to $\tilde{Q}$
   if $P = \emptyset$ then add $[P, O]$ to $\tilde{F}$
   for all $a \in \Sigma$ do
      $P' = \delta(P, a)$
      if $O \neq \emptyset$ then $O' \leftarrow \delta(O, a) \setminus F$ else $O' \leftarrow \delta(P, a) \setminus F$
      add $([P, O], a, [P', O'])$ to $\tilde{\delta}$
      if $[P', O'] \notin \tilde{Q}$ then add $[P', Q']$ to $W$
```

Figure 11.7 shows the result of applying the algorithm to our running example. The NCA is at the top, and the DCA below it on the left. On the right we show the DCA obtained by applying the powerset construction to the NCA. It is almost the same automaton, but with the important difference that the state $(\emptyset, \emptyset)$ is now accepting, and so the powerset construction does not yield a correct result. For example, the DCA obtained by the powerset construction accepts the word b^ω, which is not accepted by the original NCA, because it has no run on it. For the complexity,

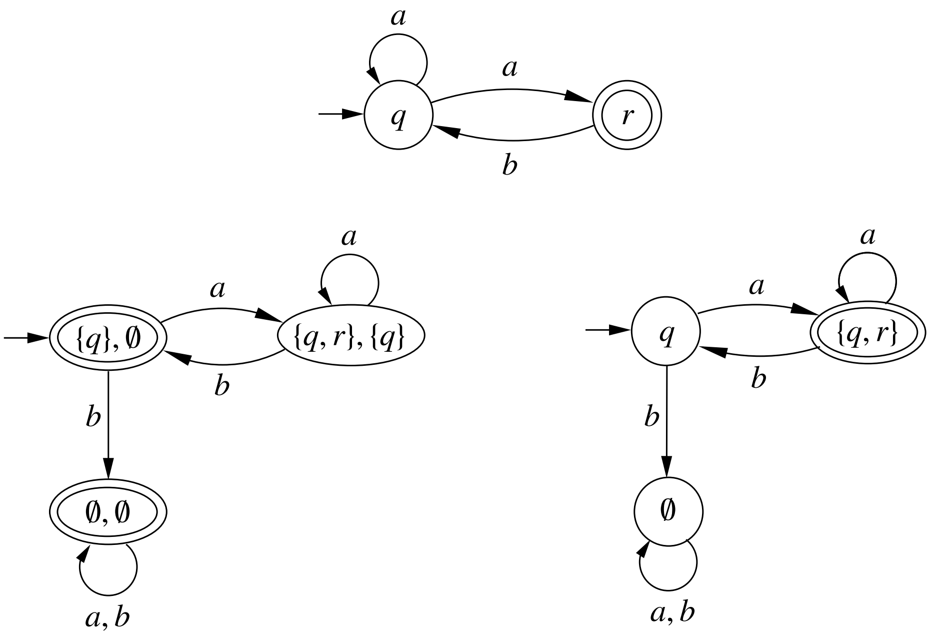


Figure 11.7: NCA of Figure 12.5 (top), DCA (lower left), and DFA (lower right)

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 possibilities: $q \in O$, $q \in P \setminus O$, and $q \in Q \setminus P$. So if A has n states then B has at most 3^n states.

Unfortunately, co-Büchi automata do not recognize all ω-regular languages. In particular, we claim that no NCA recognizes the language L of ω-words over $\{a, b\}$ containing infinitely many a's. To see why, assume some NCA recognizes L. Then, since every NCA can be determinized, some DCA A recognizes L. This automaton A, interpreted as a DBA instead of a DCA, recognizes the complement of L: indeed, a word w is recognized by the DCA A iff the run of A on w visits accepting states only finitely often iff w is not recognized by the DBA A. But the complement of L is $(a + b)^*b^\omega$, which by Proposition 11.7 is not accepted by any DBA. We have reached a contradiction, which proves the claim. So we now ask 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 $(q \in \inf(\rho)) \wedge \neg(q' \in \inf(\rho))$ 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 11.9 (Safra) *A NBA with n states can be effectively transformed into a DMA with $n^{O(n)}$ states.*

In particular, the DMA of Figure 11.8 with Muller condition $\{\ \{q_1\}\ \}$ recognizes the language $L = (a + b)^*b^\omega$, which, as shown in Proposition 11.7, is not recognized by any DBA. Indeed, a run ρ is accepting if $\inf(\rho) = \{q_1\}$, that is, if it visits state 1 infinitely often and state q_0 finitely often. So accepting runs initially move between states q_0 and q_1, but eventually jump to q_1 and never visit q_0 again. 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. Given a Muller automaton $A = (Q, \Sigma, Q_0, \delta, \{F_0, \ldots, F_{m-1}\})$,

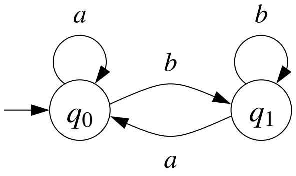


Figure 11.8: A Muller automaton for $(a + b)^*b^\omega$.

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*(); finally, we put the NBAs $A_0, \ldots, A_{m-1}$ "side by side" (i.e., take the disjoint union of their sets of states, initial states, final states, and transitions).

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 $q' \in F_i$ we add 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 11.10 Figure 11.9 shows on the left 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. On the right part 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 final NBA is the result of putting A'_0 and A'_1 side by side. □

Formally, the algorithm to convert a Muller automaton with only one accepting set into a NBA looks as follows:

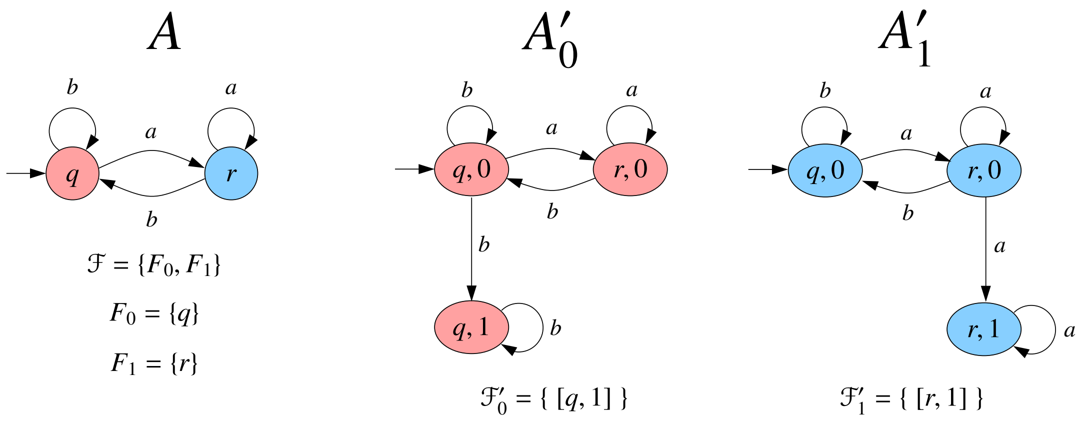

A
A'_0
A'_1
b
a
q
r
$q, 0$
$r, 0$
$q, 1$
$r, 1$
$\mathcal{F} = \{F_0, F_1\}$
$F_0 = \{q\}$
$F_1 = \{r\}$
$\mathcal{F}'_0 = \{ [q, 1] \}$
$\mathcal{F}'_1 = \{ [r, 1] \}$

Figure 11.9: A Muller automaton and its conversion into a NBA

```
NMA1toNGA(A)
Input: NMA $A = (Q, \Sigma, Q_0, \delta, \{F\})$
Output: NGA $A = (Q', \Sigma, Q'_0, \delta', \mathcal{F}')$
$Q', \delta', \mathcal{F}' \leftarrow \emptyset$
$Q'_0 \leftarrow \{[q_0, 0] \mid q_0 \in Q_0\}$
$W \leftarrow Q'_0$
while $W \neq \emptyset$ do
    pick $[q, i]$ from $W$; add $[q, i]$ to $Q'$
    if $q \in F$ and $i = 1$ then add $\{[q, 1]\}$ to $\mathcal{F}'$
    for all $a \in \Sigma,\ q' \in \delta(q, a)$ do
        if $i = 0$ then
            add $([q, 0], a, [q', 0])$ to $\delta'$
            if $[q', 0] \notin Q'$ then add $[q', 0]$ to $W$
            if $q' \in F$ then
                add $([q, 0], a, [q', 1])$ to $\delta'$
                if $[q', 1] \notin Q'$ then add $[q', 1]$ to $W$
        else /* $i = 1$ */
            if $q' \in F$ then
                add $([q, 1], a, [q', 1])$ to $\delta'$
                if $[q', 1] \notin Q'$ then add $[q', 1]$ to $W$
return $(Q', \Sigma, q'_0, \delta', \mathcal{F}')$
```

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 *Rabin automaton* is a set of pairs $\{\langle F_0, G_0\rangle, \ldots, \langle F_m, G_m\rangle\}$, where the F_i and G_i are sets of states. A run ρ is accepting if there is $i \in \{1, \ldots, m\}$ such that $\inf(\rho) \cap F_i \neq \emptyset$ and $\inf(\rho) \cap G_i = \emptyset$. If we say that a run visits a set whenever it visits one of its states, then we can concisely express this condition in words: a run is accepting if, for some pair $\langle F_i, G_i\rangle$, it visits F_i infinitely often G_i finitely often.

NBA can be easily transformed into nondterministic Rabin automata (NRA) and vice versa, without any exponential blow-up.

NBA → NRA. Just observe that a Büchi condition $\{q_1, \ldots, q_k\}$ is equivalent to the Rabin condition $\{\, (\{q_1\}, \emptyset), \ldots, (\{q_n\}, \emptyset) \,\}$.

NRA → 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 follows easily that, as in the case of Muller automata, $L_\omega(A) = \bigcup_{i=0}^{m-1} L_\omega(A_i)$ holds for the NRAs $A_i = (Q, \Sigma, Q_0, \delta, \{\langle F_i, G_i\rangle\})$. So it suffices to 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 ρ 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 $Q \setminus G_i$. So 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 11.11 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 on the right, 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.

To prove that DRAs are as expressive as NRAs it suffices to show that they are as expressive as DMAs. Indeed, since NRAs are as expressive as NBAs, both classes recognize the ω-regular languages, and, by Theorem 11.9, so do DMAs.

DMA → DRA. We sketch an algorithm that converts a Muller condition into a Rabin condition, while preserving determinism.

Let A be a DMA. Consider first the special case in which the Muller condition of A contains one single set $F = \{q_1, \ldots, q_n\}$. We use the same construction as in the conversion **NBA → NGA**: we take n copies of the DMA, and "jump" from the i-th copy to the next one whenever we visit state q_i. The result is a deterministic automaton A'. Given a run ρ of A on a word w, we have $\inf(\rho) \subseteq F$ if and only if $(q_1, 1), \in \inf(\rho')$, where ρ' is the run of A' on w Now we give A' the Rabin accepting condition consisting of the single Rabin pair $\langle \{(q_1, 1)\}, (Q \setminus F) \times \{1, \ldots, n\} \rangle$. We have:

$$\begin{array}{ll} & \rho \text{ is an accepting run of } A \\ \text{iff} & \inf(\rho) = F \\ \text{iff} & \inf(\rho) \subseteq F \text{ and } \inf(\rho) \cap (Q \setminus F) = \emptyset \\ \text{iff} & (q_1, 1) \in \inf(\rho') \text{ and } \inf(\rho') \cap (Q \setminus F \times \{1, \ldots, n\}) = \emptyset \\ \text{iff} & \rho' \text{ is an accepting run of } A' \end{array}$$

and so $L_\omega A = L_\omega A'$.

If the Muller condition of A contains multiple sets $\{F_0, \ldots, F_{m-1}\}$, then we have $L(()A) = \bigcup_{i=0}^{m-1} L(()A_i')$, where A_i is the DRA for the set F_i defined above Let $A_i = (Q_i, \Sigma, q_{0i}, \delta_i, \{\langle q_{fi}, G_i \rangle\})$. We construct a DRA A' by pairing the A_i (that is, a state of A' is a tuple of states of the A_i). Further, we give A' the Rabin condition with pairs $\langle F_0', G_0' \rangle \ldots \langle F_{m-1}', G_{m-1}' \rangle$ defined as follows: F_i' contains a tuple $(q_0, \ldots, q_{m-1})$ of states iff $q_i = q_{fi}$, and G_i' contains a tuple $(q_0, \ldots, q_{m-1})$ iff $q_i \in G_i$.

11.4.4 Streett automata

The accepting condition of Rabin automata is not "closed under negation". Indeed, the negation of

there is $i \in \{1, \ldots, m\}$ such that $\inf(\rho) \cap F_i \neq \emptyset$ and $\inf(\rho) \cap G_i = \emptyset$

has 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*. More precisely, the acceptance condition of a *Streett automaton* 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 $\inf(\rho) \cap F_i = \emptyset$ or $\inf(\rho) \cap G_i \neq \emptyset$ holds for *every* pair $\langle F_i, G_i \rangle$. Observe that the condition is equivalent to: if $\inf(\rho) \cap G_i = \emptyset$, then $\inf(\rho) \cap F_i = \emptyset$.

A Büchi automaton can be easily transformed into a Streett automaton and vice versa. However, the conversion from Streett to Büchi is exponential.

NBA → NSA. A Büchi condition $\{q_1, \ldots, q_k\}$ corresponds to the Streett condition $\{\langle Q, \{q_1, \ldots, q_k\} \rangle\}$.

NSA → NBA. We can transform an NSA into an NBA by following the path NSA → NMA → NBA. If the NSA has n states, the resulting NBA has $2n^2 2^n$ 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 11.12 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. □

Deterministic automata are useful for the design of complementation algorithms. However, neither DMAs, DRAs, nor DSAs yield a polynomial complementation procedure. Indeed, while we can complement a DMA with set of states Q and accepting condition $\mathcal{F}$ by changing the condition to $2^Q \setminus \mathcal{F}$, the number of acceptings sets of $2^Q \setminus \mathcal{F}$ can be exponentially larger. In the case of Rabin and Street automata, we can complement in linear time by negating the accepting condition, but the result is an automaton that belongs to the other class, and, again, if we wish to obtain an automaton of the same class, then the accepting condition becomes exponentially larger in the worst case.

These considerations lead us to out final question: is there a class of ω-automata that (1) recognizes all ω-regular languages, (2) has a determinization procedure, (3) has polynomial conversion algorithms to and from NBA, and (4) has a polynomial complementation procedure ?

11.4.5 Parity automata

The acceptance condition of a *nondeterministic parity automaton* (NPA) with set of states Q is a sequence $(F_1, F_2, \ldots, F_{2n})$ of sets of states, where $F_1 \subseteq F_2 \subseteq \cdots \subseteq F_{2n} = Q$. A run ρ of a parity automaton is *accepting* if the minimal index i such that $\inf(\rho) \cap F_i \neq \emptyset$ is even.

The following conversions show that NPAs recognize the ω-regular languages, can be converted to and from NBAs without exponential blowup, and have a determinization procedure.

NBA → NPA. A NBA with a set F of accepting states recognizes the same language as the same automaton with parity condition $(\emptyset, F, Q, Q)$.

NPA → NBA. Use the construction NPA → NRA shown below, followed by NRA → NBA.

NPA → NRA. A NPA with accepting condition $(F_1, F_2, \ldots, F_{2n})$ recognizes the same language as the same automaton with Rabin condition $\{\langle F_{2k}, F_{2k-1}\rangle, \ldots, \langle F_3, F_2\rangle, \langle F_1, \emptyset\rangle$. Incidentally, this shows that the parity condition is a speical case of the Rabin condition in which the sets appearing in the pairs form a chain with respect to set inclusion.

DMA → DPA. In order to construct a DPA equivalent to a given NPA we can proceed as follows. First we transform the NPA into a DMA, for example following the path NPA → NRA → NBA → DMA, and then transform the DMA into an equivalent DPA. This construction can be achieved by means of so-called *latest appearance records*. Alternatively, it is also possible to modify Safra's determinization procedure so that it yields a DPA instead of a DMA.

Theorem 11.13 (Safra, Piterman) *A NBA with n states can be effectively transformed into a DPA with $n^{O(n)}$ states and an accepting condition with $O(n)$ sets.*

Finally, DPAS have a very simple complementation procedure:

Complementation of DPAs. In order to complement a parity automaton with accepting condition $(F_1, F_2, \ldots, F_{2n})$, replace the condition by $(\emptyset, F_1, F_2, \ldots, F_{2n}, F_{2n})$.

11.4.6 Conclusion

We have presented a short overview of the "zoo" of classes of ω-automata. If we are interested in a determinizable class with a simple complementation procedure, then parity automata are the right choice. However, the determinization procedures for ω-automata not only have large complexity, but are also difficult to implement efficiently. For this reason in the next chapter we present implementations of our operations using NBA. Since not all NBAs can be determinized, we have to find a complementation operation that does not require to previously determinize the automaton.

Exercises

Exercise 118 Construct Büchi automata and ω-regular expressions, as small as possible, recognizing the following languages over the alphabet $\{a, b, c\}$. Recall that $\mathit{inf}(w)$ denotes the set of letters of $\{a, b, c\}$ that occur infinitely often in w.

(1) $\{w \in \{a, b, c\}^\omega \mid \{a, b\} \supseteq \mathit{inf}(w)\}$

(2) $\{w \in \{a, b, c\}^\omega \mid \{a, b\} = \mathit{inf}(w)\}$

(3) $\{w \in \{a, b, c\}^\omega \mid \{a, b\} \subseteq \mathit{inf}(w)\}$

(4) $\{w \in \{a, b, c\}^\omega \mid \{a, b, c\} = \mathit{inf}(w)\}$

(5) $\{w \in \{a, b, c\}^\omega \mid$ if $a \in \mathit{inf}(w)$ then $\{b, c\} \subseteq inf(w)\}$

Exercise 119 Give deterministic Büchi automata accepting the following ω-languages over $\Sigma = \{a, b, c\}$:

(1) $L_1 = \{w \in \Sigma^\omega : w$ contains at least one $c\}$,

(2) $L_2 = \{w \in \Sigma^\omega$: in w, every a is immediately followed by a $b\}$,

(3) $L_3 = \{w \in \Sigma^\omega$: in w, between two successive a's there are at least two b's$\}$.

Exercise 120 Prove or disprove:

1. For every Büchi automaton A, there exists a NBA B with a single initial state and such that $L_\omega(A) = L_\omega(B)$.
2. For every Büchi automaton A, there exists a NBA B with a single accepting state and such that $L_\omega(A) = L_\omega(B)$.

Exercise 121 Recall that every finite set of finite words is a regular language. We prove that not every finite set of ω-words is an ω-regular language.

(1) Prove that every ω-regular language contains an *ultimately periodic* ω-word, i.e., an ω-word of the form $u\, v^\omega$ for some finite words w, v.

(2) Give an ω-word w such that $\{w\}$ is not an ω-regular language.

Exercise 122 (Duret-Lutz) An ω-automaton has acceptance on transitions if the acceptance condition specifies which transitions must appear finitely or infinitely often in a run, instead of which states. All classes of ω-automata (Büchi, Rabin, etc.) can be defined with acceptance on states, or acceptance on transitions.

Give minimal deterministic automata for the language of words over $\{a, b\}$ containing infinitely many a and infinitely many b. of the following kinds (1) Büchi, (2) generalized Büchi, (3) Büchi with acceptance on transitions, and (4) generalized Büchi with acceptance on transitions.

Exercise 123 Consider the class of non deterministic automata over infinite words with the following acceptance condition: an infinite run is accepting if it visits a final state *at least once*. Show that no such automaton accepts the language of all words over $\{a, b\}$ containing infinitely many a and infinitely many b.

Exercise 124 The *limit* of a language $L \subseteq \Sigma^*$, denoted by $lim(L)$, is the ω-language defined as follows: $w \in lim(L)$ iff infinitely many prefixes of w are words of L. For example, the limit of $(ab)^*$ is $\{(ab)^\omega\}$.

(1) Determine the limit of the following regular languages over $\{a, b\}$: (i) $(a+b)^*a$; (ii) $(a+b)^*a^*$; (iii) the set of words containing an even number of as; (iv) a^*b.

(2) Prove: An ω-language is recognizable by a deterministic Büchi automaton iff it is the limit of a regular language.

(3) Exhibit a non-regular language whose limit is ω-regular.

(4) Exhibit a non-regular language whose limit is not ω-regular.

Exercise 125 Let $L_1 = (ab)^\omega$, and let L_2 be the language of all words containing infinitely many a and infinitely many b (both languages over the alphabet $\{a, b\}$).

(1) Show that no DBA with at most two states recognizes L_1 or L_2.

(2) Exhibit two different DBAs with three states recognizing L_1.

(3) Exhibit six different DBAs with three states recognizing L_2.

Exercise 126 Find ω-regular expressions (the shorter the better) for the following languages:

(1) $\{w \in \{a, b\}^\omega \mid k \text{ is even for each subword } ba^k b \text{ of } w\}$

(2) $\{w \in \{a, b\}^\omega \mid w \text{ has no occurrence of } bab\}$

Exercise 127 In Definition 3.18 we have introduced the quotient A/P of a NFA A with respect to a partition P of its states. In Lemma 3.20 we have proved $L(A) = L(A/P_\ell)$ for the language partition P_ℓ that puts two states q_1, q_2 in same block iff $L_A(q_1) = L_A(q_2)$.

Let $B = (Q, \Sigma, \delta, Q_0, F)$ be a NBA. Given a partition P of Q, define the quotient B/P of B with respect to P exactly as for NFA.

(1) Let P_ℓ be the partition of Q that puts two states q_1, q_2 of B in same block iff $L_{\omega,B}(q_1) = L_{\omega,B}(q_2)$, where $L_{\omega,B}(q)$ denotes the ω-language containing the words accepted by B with q has initial state. Does $L_\omega(B) = L_\omega(B/P_\ell)$ always hold ?

(2) Let CSR be the coarsest stable refinement of the equivalence relation with equivalence classes $\{F, Q \setminus F\}$. Does $L_\omega(A) = L_\omega(A/CSR)$ always hold ?

Exercise 128 Let L be an ω-language over Σ, and let $w \in \Sigma^*$. The *w-residual* of L is the ω-language $L^w = \{w' \in \Sigma^\omega \mid w\,w' \in L\}$. An ω-language L' is a *residual* of L if $L' = L^w$ for some word $w \in \Sigma^*$.

We show that the theorem stating that a language of finite words is regular iff it has finitely many residuals does not extend to ω-regular languages.

(1) Prove: If L is an ω-regular language, then it has finitely many residuals.

(2) Disprove: Every ω-language with finitely many residuals is ω-regular.
Hint: Let w be a non-ultimately-periodic ω-word and consider the language $Tail_w$ of infinite tails of w.

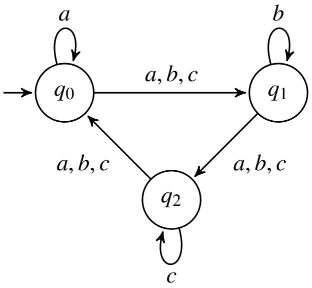
Exercise 129 The solution to Exercise 127(2) shows that the reduction algorithm for NFAs that computes the partition CSR of a given NFA A and constructs the quotient A/CSR can also be applied to NBAs. Generalize the algorithm so that it works for NGAs.

Exercise 130 Let $L = \{w \in \{a, b\}^\omega \mid w \text{ contains finitely many } a\}$

(1) Give a deterministic Rabin automaton for L.

(2) Give a NBA for L and try to "determinize" it by using the NFA to DFA powerset construction. Which is the language accepted by the deterministic automaton?

(3) What ω-language is accepted by the following Muller automaton with acceptance condition $\{\ \{q_0\}, \{q_1\}, \{q_2\}\ \}$? And with acceptance condition $\{\ \{q_0, q_1\}, \{q_1, q_2\}, \{q_2, q_0\}\ \}$?



(4) Show that any Büchi automaton that accepts the ω-language of (c) (with the first acceptance condition) has more than 3 states.

(5) For every $m, n \in \mathbb{N}_{>0}$, let $L_{m,n}$ be the ω-language over $\{a, b\}$ described by the ω-regular expression $(a + b)^*((a^m bb)^\omega + (a^n bb)^\omega)$.

(i) Describe a family of Büchi automata accepting the family of ω-languages $\{L_{m,n}\}_{m,n \in \mathbb{N}_{>0}}$.

(ii) Show that there exists $c \in \mathbb{N}$ such that for every $m, n \in \mathbb{N}_{>0}$ the language $L_{m,n}$ is accepted by a Rabin automaton with at most $\max(m, n) + c$ states.

(iii) Modify your construction in (ii) to obtain Muller automata instead of Rabin automata.

(iv) Convert the Rabin automaton for $L_{m,n}$ obtained in (ii) into a Büchi automaton.

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

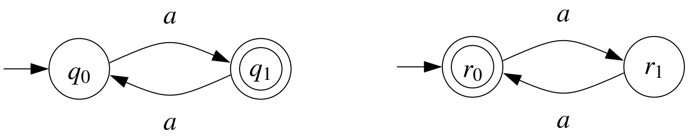


Figure 12.1: Two Büchi automata accepting the language a^ω

87 (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 $L_\omega(A_1 \cap A_2) = \emptyset$.

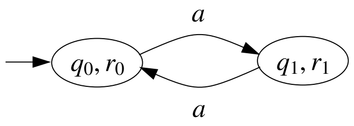


Figure 12.2: The automaton $A_1 \cap A_2$

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 → 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

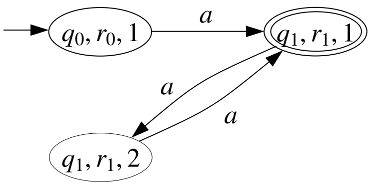


Figure 12.3: The NBA $A_1 \cap_\omega A_2$ for the automata A_1 and A_2 of Figure 12.1

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. $\square$

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 $L_\omega(A_1 \cap_\omega A_2) = L_\omega(A_1) \cap L_\omega(A_2)$

$Q, \delta, F \leftarrow \emptyset$
$q_0 \leftarrow [q_{01}, q_{02}, 1]$
$W \leftarrow \{\ [q_{01}, q_{02}, 1]\ \}$
while $W \neq \emptyset$ do
   pick $[q_1, q_2, i]$ from $W$
   add $[q_1, q_2, i]$ to $Q'$
   if $q_1 \in F_1$ and $i = 1$ then add $[q_1, q_2, 1]$ to $F'$
   for all $a \in \Sigma$ do
      for all $q'_1 \in \delta_1(q_1, a), q'_2 \in \delta(q_2, a)$ do
         if $i = 1$ and $q_1 \notin F_1$ then
            add $([q_1, q_2, 1], a, [q'_1, q'_2, 1])$ to $\delta$
            if $[q'_1, q'_2, 1] \notin Q'$ then add $[q'_1, q'_2, 1]$ to $W$
         if $i = 1$ and $q_1 \in F_1$ then
            add $([q_1, q_2, 1], a, [q'_1, q'_2, 2])$ to $\delta$
            if $[q'_1, q'_2, 2] \notin Q'$ then add $[q'_1, q'_2, 2]$ to $W$
         if $i = 2$ and $q_2 \notin F_2$ then
            add $([q_1, q_2, 2], a, [q'_1, q'_2, 2])$ to $\delta$
            if $[q'_1, q'_2, 2] \notin Q'$ then add $[q'_1, q'_2, 2]$ to $W$
         if $i = 2$ and $q_2 \in F_2$ then
            add $([q_1, q_2, 2], a, [q'_1, q'_2, 1])$ to $\delta$
            if $[q'_1, q'_2, 1] \notin Q'$ then add $[q'_1, q'_2, 1]$ to $W$
return $(Q, \Sigma, \delta, Q_0, F)$
```

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 → 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.

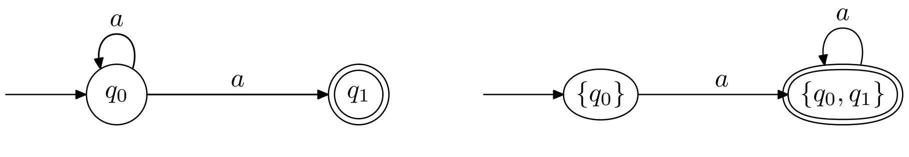


Figure 12.4: The subset construction does not preserve ω-languages

(b) The subset construction cannot be replaced by another determinization procedure, because no such procedure exists: As we have seen in Proposition 11.7, 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 $\overline{A}$ satisfying:

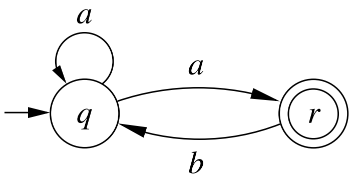


Figure 12.5: Running example for the complementation procedure

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 level rankings

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.

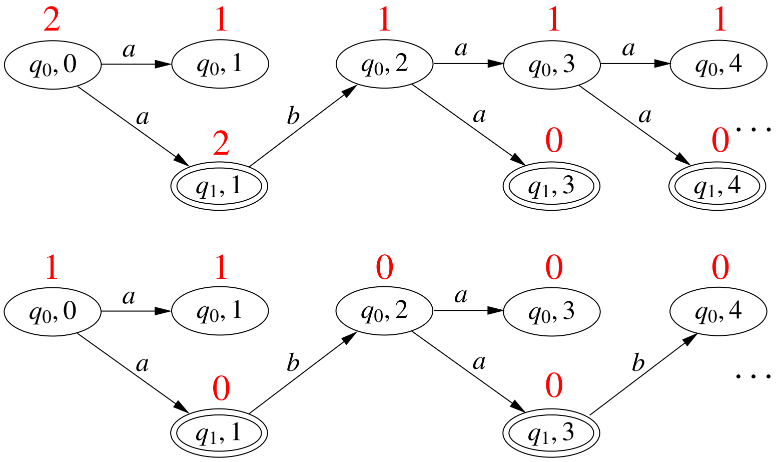


Figure 12.6: Rankings for $dag(aba^\omega)$ and $dag((ab)^\omega)$

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 rankings:

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 such that $lr(q) \neq \bot$ satisfies $q \xrightarrow{a} q'$, and
- $lr(q) \geq lr'(q')$ for every q satisfying $lr(q) \neq \bot$ and $q \xrightarrow{a} q'$.

12.2.3 A (possibly 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 lr_n defined by: $rl_n(q_0) = n$, and $lr_n(q) = \bot$ for every $q \neq q_0$.
- The transitions are the triples (lr, a, lr'), where lr and lr' are level rankings, $a \in \Sigma$, and $lr \overset{a}{\mapsto} lr'$ 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 lr is a level ranking and O is a subset of the states q for which $lr(q) \in \mathbb{N}$.
- The initial states are all pairs of the form $[lr, \emptyset]$, where $lr(q_0) \neq \bot$ and $lr_0(q) = \bot$ for every $q \neq q_0$.
- The transitions are the triples $[lr, O] \xrightarrow{a} [lr', O']$ such that $lr \overset{a}{\mapsto} lr'$ 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 level rankings 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 $\bot$.

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.

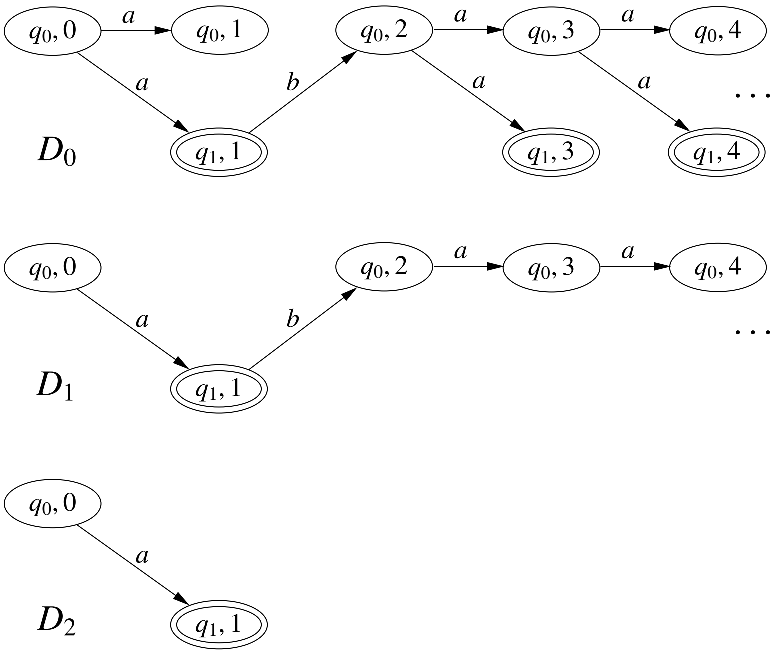


Figure 12.7: The DAGS D_0, D_1, D_2 for $dag(aba^\omega)$

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 $\langle 2, \emptyset \rangle$. Let us compute the successors of $\langle 2, \emptyset \rangle$ 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 $\langle 2, \emptyset \rangle$ 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 $\langle 0, \{q\} \rangle$. Similarly, the successors of $\langle 2, \{q\} \rangle$ are $\langle 2, \{q\} \rangle$ and $\langle 0, \{q\} \rangle$.

Since $\langle 2, \emptyset \rangle$ is its only accepting state, $\overline{B}_1$ 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 $\overline{B}_1$ is that, since q is no longer accepting, it can also have odd rank 1. So $\langle 2, \emptyset \rangle$ has three successors: $\langle 2, \{q\} \rangle$, $\langle 1, \emptyset \rangle$, and $\langle 0, \{q\} \rangle$. The successors of $\langle 1, \emptyset \rangle$ are $\langle 1, \emptyset \rangle$ and $\langle 0, \{q\} \langle$. The successors of $\langle 2, \{q\} \rangle$ are $\langle 2, \{q\} \rangle$, $\langle 1, \emptyset \rangle$, and $\langle 0, \{q\} \rangle$, and the only successor of $\langle 0, \{q\} \rangle$ is $\langle 0, \{q\} \rangle$. The accepting states are $\langle 2, \emptyset \rangle$ and $\langle 1, \emptyset \rangle$, and $\overline{B}_2$ recognizes a^ω. $\overline{B}_2$ is shown on the right of Figure 12.8. □

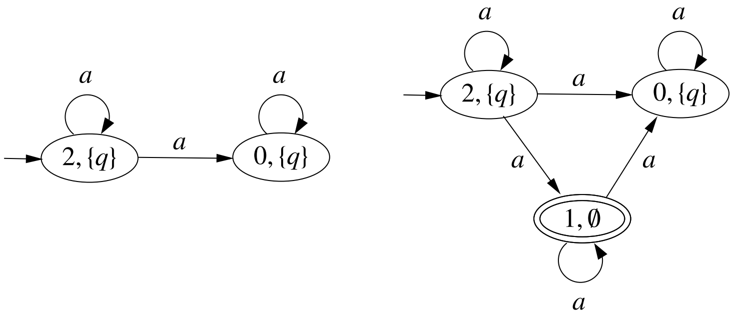


Figure 12.8: The NBAs $\overline{B}_1$ and $\overline{B}_2$

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 $q' \in Q$ we have: $lr'(q') = \bot$ iff no $q \in Q$ satisfies $lr(q) \neq \bot$ and $q \xrightarrow{a} q'$, and $lr(q) \geq lr'(q')$ for every q such that $lr(q) \neq \bot$ and $q \xrightarrow{a} q'$.

```
$CompNBA(A)$
Input: NBA $A = (Q, \Sigma, \delta, Q_0, F)$
Output: NBA $\overline{A} = (\overline{Q}, \Sigma, \overline{\delta}, \overline{q}_0, \overline{F})$ with $L_\omega(\overline{A}) = \overline{L_\omega(A)}$
$\overline{Q}, \overline{\delta}, \overline{F} \leftarrow \emptyset$
$\overline{q}_0 \leftarrow [lr_0, \{q_0\}]$
$W \leftarrow \{\ [lr_0, \{q_0\}]\ \}$
while $W \neq \emptyset$ do
    pick $[lr, P]$ from $W$; add $[lr, P]$ to $\overline{Q}$
    if $P = \emptyset$ then add $[lr, P]$ to $\overline{F}$
    for all $a \in \Sigma, lr' \in \mathcal{R}$ such that $lr \overset{a}{\mapsto} lr'$ do
        if $P \neq \emptyset$ then $P' \leftarrow \{q \in \delta(P, a) \mid lr'(q)$ is even $\}$
        else $P' \leftarrow \{q \in Q \mid lr'(q)$ is even $\}$
        add $([lr, P], a, [lr', P'])$ to $\overline{\delta}$
        if $[lr', P'] \notin \overline{Q}$ then add $[lr', P']$ to $W$
return $(\overline{Q}, \Sigma, \overline{\delta}, \overline{q}_0, \overline{F})$
```

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 $w \in L_n$.

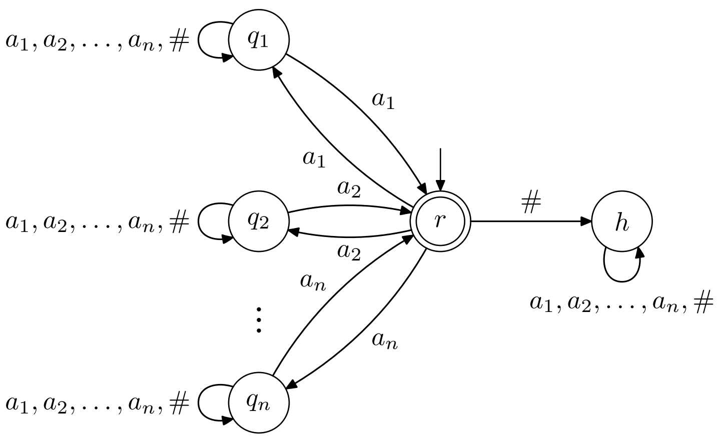

$a_1, a_2, \ldots, a_n, \#$
q_1
a_1
a_1
a_2
$a_1, a_2, \ldots, a_n, \#$
q_2
r
$\#$
h
a_2
a_n
$a_1, a_2, \ldots, a_n, \#$
a_n
$a_1, a_2, \ldots, a_n, \#$
q_n

Figure 12.9: The automaton A_n

Let ρ be a run of A_n accepting w, and let $Q_\rho = \mathit{inf}(\rho) \cap \{q_, \ldots, q_n\}$. 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 $\langle \tau(1), \tau(a_2) \rangle$, $\langle \tau(a_2), \tau(a_3) \rangle$, ..., $\langle \tau(a_{n-1}), \tau(a_n) \rangle$, 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

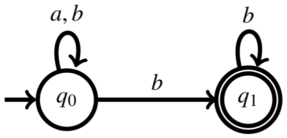
Exercise 131 1. Give deterministic Bchi automata for L_a, L_b, L_c where $L_\sigma = \{w \in \{a,b,c\}^\omega :$ w contains infinitely many σ's$\}$, and build the intersection of these automata.

2. Give Bchi automata for the following ω-languages:

- $L_1 = \{w \in \{a,b\}^\omega : w$ contains infinitely many a's$\}$,
- $L_2 = \{w \in \{a,b\}^\omega : w$ contains finitely many b's$\}$,
- $L_3 = \{w \in \{a,b\}^\omega :$ each occurrence of a in w is followed by a $b\}$,

and build the intersection of these automata.

Exercise 132 Consider the following Bchi automaton over $\Sigma = \{a,b\}$:



1. Sketch dag($abab^\omega$) and dag($(ab)^\omega$).

2. Let r_w be the ranking of dag(w) defined by

$$r_w(q,i) = \begin{cases} 1 & \text{if } q = q_0 \text{ and } \langle q_0, i\rangle \text{ appears in dag}(w), \\ 0 & \text{if } q = q_1 \text{ and } \langle q_1, i\rangle \text{ appears in dag}(w), \\ \bot & \text{otherwise.} \end{cases}$$

Are r_{abab^ω} and $r_{(ab)^\omega}$ odd rankings?

3. Show that r_w is an odd ranking if and only if $w \notin L_\omega(B)$.

4. Build a Bchi automaton accepting $\overline{L_\omega(B)}$ using the construction seen in class. (Hint: by (c), it is sufficient to use $\{0, 1\}$ as ranks.)

Exercise 133 Find algorithms (not necessarily eficient) for the following decision problems:

(1) Given finite words $u, v, x, y \in \Sigma^*$, decide whether the ω-words $u\,v^\omega$ and $x\,y^\omega$ are equal.

(2) Given a Büchi automaton A and finite words u, v, decide whether A accepts the ω-word $u\,v^\omega$.

Exercise 134 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 135 A B´uchi automaton $A = (Q, \Sigma, \delta, Q_0, F)$ is *weak* if no strongly connected component (SCC) of states contains of A both accepting and non-accepting states, that is, every SCC $C \subseteq Q$ satisfies either $C \subseteq F$ or $C \subseteq Q \setminus F$.

(a) Prove that a B´uchi automaton A is *weak* iff for every run ρ either $\mathit{inf}(\rho) \subseteq F$ or $\mathit{inf}(\rho) \subseteq Q \setminus F$.

(b) Prove that the algorithms for union, intersection, and complementation of DFAs are also correct for weak DBAs. More precisely, show that the algorithms return weak DBAs recognizing the union, intersection, and complement, respectively, of the languages of the input automata.

Exercise 136 Give algorithms that directly complement deterministic Muller and parity automata, without going through Büchi automata.

Exercise 137 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 138 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 $\mathit{dag}(w)$ 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 $A = (Q, \Sigma, \delta, Q_0, F)$
$S \leftarrow \emptyset$
$dfs(q_0)$

proc $dfs(q)$
    add $q$ to $S$
    for all $r \in \delta(q)$ do
        if $r \notin S$ then $dfs(r)$
    return
```

```
DFS_Tree(A)
Input: NBA $A = (Q, \Sigma, \delta, Q_0, F)$
Output: Time-stamped tree $(S, T, d, f)$
$S \leftarrow \emptyset$
$T \leftarrow \emptyset$; $t \leftarrow 0$
$dfs(q_0)$

proc $dfs(q)$
    $t \leftarrow t + 1$; $d[q] \leftarrow t$
    add $q$ to $S$
    for all $r \in \delta(q)$ do
        if $r \notin S$ then
            add $(q, r)$ to $T$; $dfs(r)$
    $t \leftarrow t + 1$; $f[q] \leftarrow t$
    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.

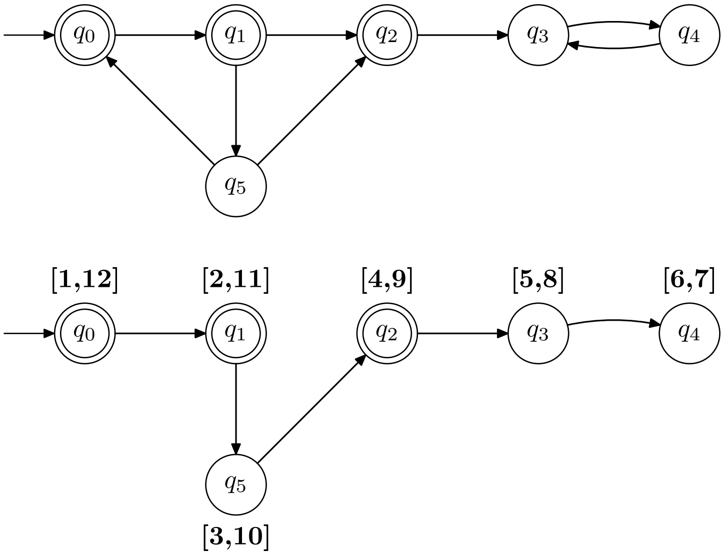


Figure 13.1: An NBA (the labels of the transitions have been omitted), and a possible run of *DFS_Tree* on it. The numeric intervals are the discovery and finishing times of the states, shown in the format $[d[q], f[q]]$.

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 $q_1 q_6 q_0$. □

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, . . . , 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.

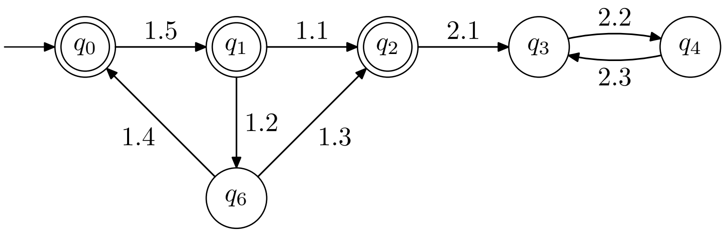


Figure 13.2: The transitions explored during the search starting at q_i are labelled by the index i. The search starting at q_1 stops with NONEMPTY.

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   $S \leftarrow \emptyset$
2   $dfs1(q_0)$
3   report EMP

4   proc $dfs1(q)$
5     add $[q, 1]$ to $S$
6     for all $r \in \delta(q)$ do
7       if $[r, 1] \notin S$ then $dfs1(r)$
8     if $q \in F$ then { $seed \leftarrow q$; $dfs2(q)$ }
9     return

10  proc $dfs2(q)$
11    add $[q, 2]$ to $S$
12    for all $r \in \delta(q)$ do
13      if $r = seed$ then report NEMP
14      if $[r, 2] \notin S$ 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   $S \leftarrow \emptyset$; $succ \leftarrow$ false
2   $dfs1(q_0)$
3   report EMP

4   proc $dfs1(q)$
5     add $[q, 1]$ to $S$
6     for all $r \in \delta(q)$ do
7       if $[r, 1] \notin S$ then $dfs1(r)$
8       if $succ$ = true then return $[q, 1]$
9     if $q \in F$ then
10       $seed \leftarrow q$; $dfs2(q)$
11      if $succ$ = true then return $[q, 1]$
12    return

13  proc $dfs2(q)$
14    add $[q, 2]$ to $S$
15    for all $r \in \delta(q)$ do
16      if $[r, 2] \notin S$ then $dfs2(r)$
17      if $r = seed$ then
18        report NEMP; $succ \leftarrow$ true
19      if $succ$ = true then return $[q, 2]$
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
$S \leftarrow \emptyset$; $P \leftarrow \emptyset$
dfs1($q_0$)
report EMP

proc dfs1($q$)
  add $[q, 1]$ to $S$ ; add $q$ to $P$
  for all $r \in \delta(q)$ do
    if $[r, 1] \notin S$ then dfs1($r$)
  if $q \in F$ then dfs2($q$)
  remove $q$ from $P$
  return

proc dfs2($q$)
  add $[q, 2]$ to $S$
  for all $r \in \delta(q)$ do
    if $r \in P$ then report NEMP
    if $[r, 2] \notin S$ then dfs2($r$)
  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, 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.

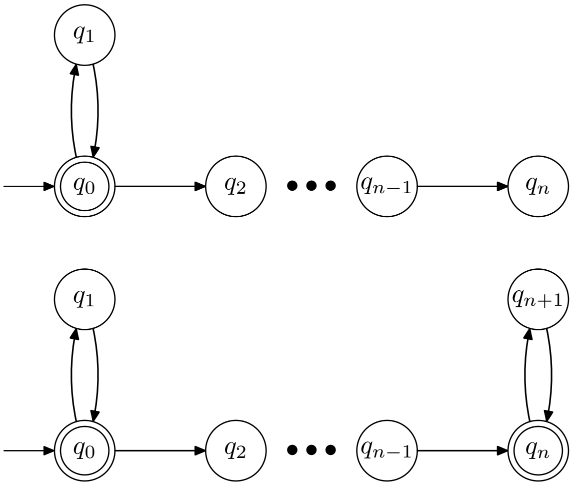


Figure 13.3: Two bad examples for *NestedDFS*

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 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 $A_{f[q]}$ □

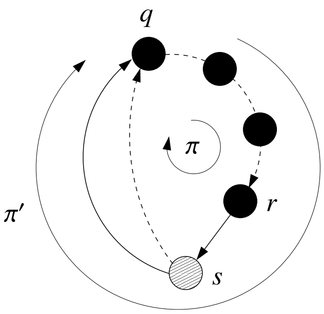


Figure 13.4: Illustration of the proof of Lemma 13.6

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 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.

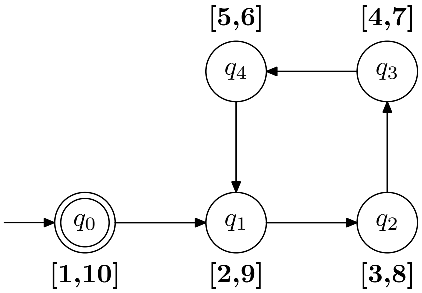


Figure 13.5: A DFS on an automaton

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 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

```
FirstAttempt($A$)
Input: NBA $A = (Q, \Sigma, \delta, Q_0, F)$
Output: EMP if $L_\omega(A) = \emptyset$
        NEMP otherwise
$S, C \leftarrow \emptyset$;
dfs($q_0$)
report EMP

proc dfs($q$)
   add $q$ to $S$; push($q, C$)
   for all $r \in \delta(q)$ do
      if $r \notin S$ then dfs($r$)
      else if $r \rightsquigarrow q$ then
         repeat
            $s \leftarrow$ pop($C$)
            if $s \in F$ then report NEMP
         until $s = r$
   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
$S, C \leftarrow \emptyset$;
dfs1($q_0$)
report EMP

proc dfs($q$)
   add $q$ to $S$; push($q, C$)
   for all $r \in \delta(q)$ do
      if $r \notin S$ then dfs($r$)
      else if $r \rightsquigarrow q$ then
         repeat
            $s \leftarrow$ pop($C$)
            if $s \in F$ then report NEMP
         until $s = r$
         push($r, C$)
   if top($C$) $= q$ then pop($C$)
```

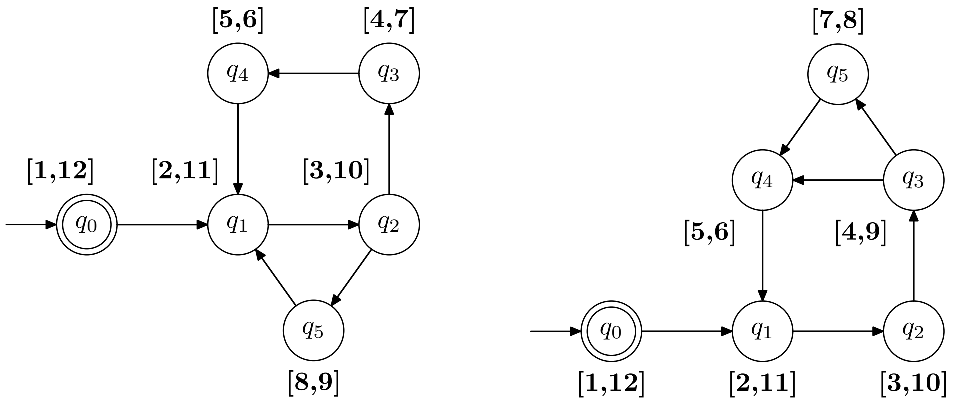


Figure 13.6: Two incorrect attempts at an emptiness checking algorithm

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
$S, C \leftarrow \emptyset$;
dfs($q_0$)
report EMP

dfs(q)
   add $q$ to $S$ ; push($q, C$)
   for all $r \in \delta(q)$ do
      if $r \notin S$ then dfs($r$)
      else if $r \rightsquigarrow q$ then
         repeat
            $s \leftarrow$ pop($C$); if $s \in F$ then report NEMP
         until $d[s] \leq d[r]$
         push($s, C$)
   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.

- 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 $d[r] \leq d[q']$. □

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:

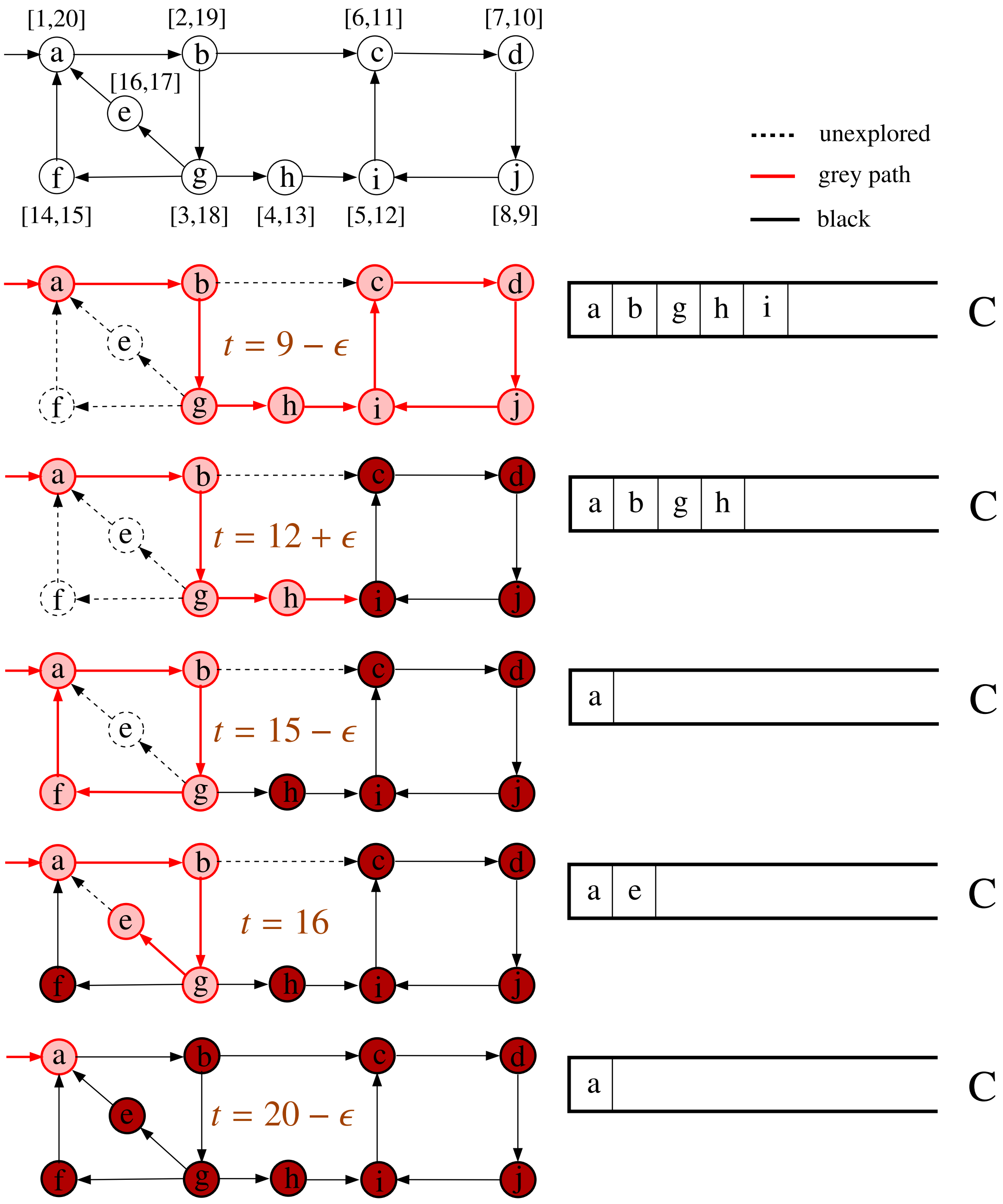


Figure 13.7: A run of *OneStack*

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.

(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.

[3] Notice that a path consisting of just a state q and no transitions is a path leading from q to 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 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 outgoing 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 $r \rightsquigarrow q$. □

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* **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 **top**$(C) = q$ at line 13.

Assume now that q is not a root. Then there is a path from q to the root 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 $q \neq$ **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 13.8 shows the run of *TwoStack* on the same example considered in Figure 13.7. The figure shows the content of V at different times when the policy above is followed. Right before state j is blackened, V 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 V. 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. □

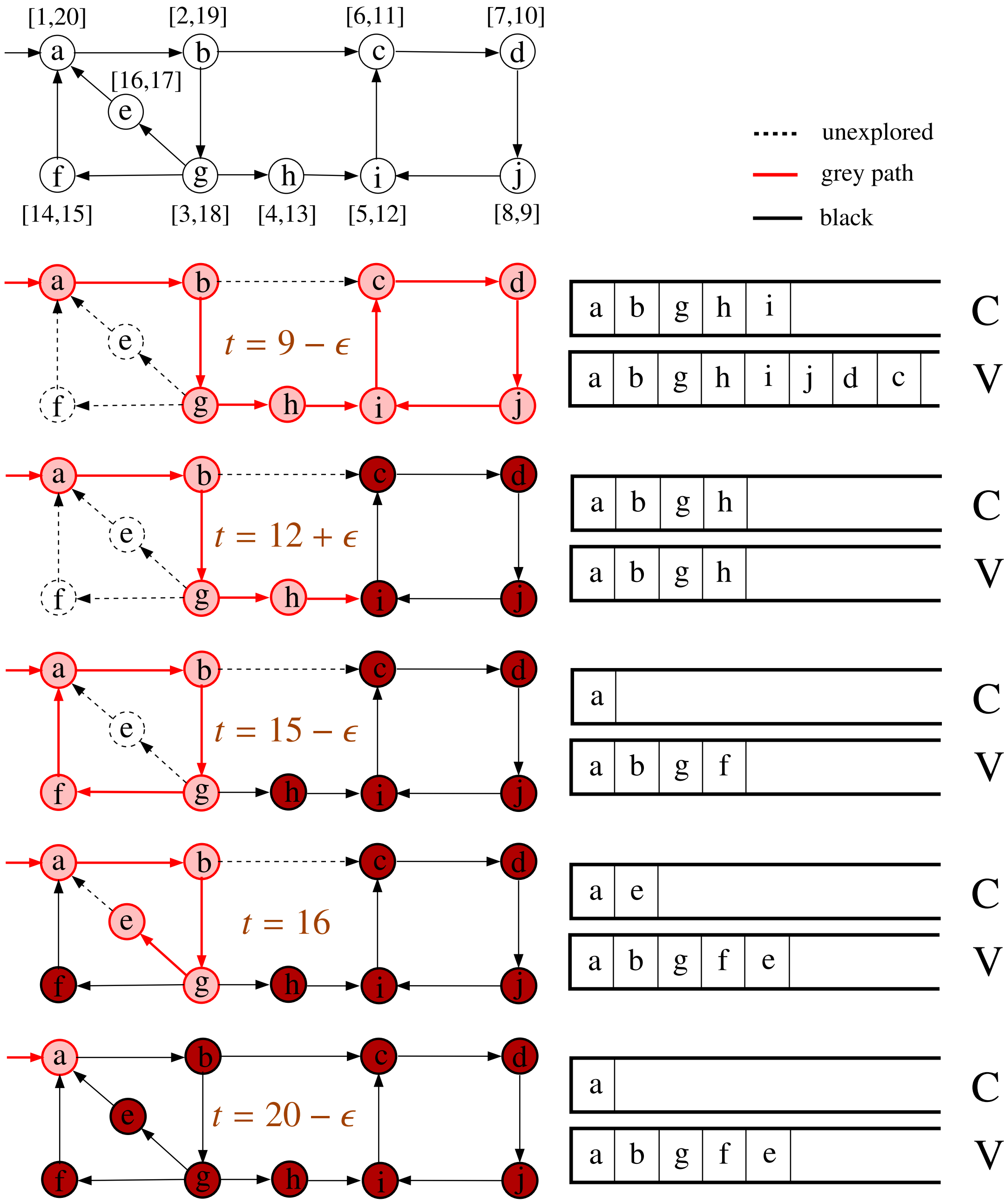


Figure 13.8: A run of *TwoStack*

This finally leads to the two-stack algorithm

```
*TwoStack*($A$)
**Input:** NBA $A = (Q, \Sigma, \delta, Q_0, F)$
**Output:** EMP if $L_\omega(A) = \emptyset$, NEMP otherwise
 1  $S, C, V \leftarrow \emptyset$;
 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** $r \in \delta(q)$ **do**
 7        **if** $r \notin S$ **then** $dfs(r)$
 8        **else if** $r \in V$ **then**
 9           **repeat**
10              $s \leftarrow$ **pop**$(C)$; **if** $s \in F$ **then report** NEMP
11           **until** $d[s] \leq d[r]$
12           **push**$(s, C)$
13     **if top**$(C) = q$ **then**
14        **pop**$(C)$
15        **repeat** $s \leftarrow$ **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 V 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  $S, C, V \leftarrow \emptyset$;
 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 $r \in \delta(q)$ do
 7      if $r \notin S$ then $dfs(r)$
 8      else if $r \in V$ then
 9        $I \leftarrow \emptyset$
10        repeat
11          $[s, J] \leftarrow$ pop($C$);
12          $I \leftarrow I \cup J$; if $I = K$ then report NEMP
13        until $d[s] \leq d[r]$
14        push($[s, I], C$)
15    if top($C$) $= (q, I)$for some $I$ then
16      pop($C$)
17      repeat $s \leftarrow$ 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 V at line 8).

The stacks C and V 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 $A = (Q, \Sigma, \delta, Q_0, F)$,
  $Q_0 \subseteq Q$

    $S, B \leftarrow Q_0$;
    repeat
      $B \leftarrow \delta(B) \setminus S$
      $S \leftarrow S \cup B$
    until $B = \emptyset$
```

```
BackwardBFS[A](Q_0)
Input:
  NBA $A = (Q, \Sigma, \delta, Q_0, F)$,
  $Q_0 \subseteq Q$

    $S, B \leftarrow Q_0$;
    repeat
      $B \leftarrow \delta^{-1}(B) \setminus S$
      $S \leftarrow S \cup B$
    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 $L \leftarrow Q$
2 repeat
3    $OldL \leftarrow L$
4    $L \leftarrow \mathbf{Pre}(OldL \cap F)$
5    $L \leftarrow BackwardBFS(L)$
6 until $L = OldL$
7 if $q_0 \in L$ 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 $L \leftarrow Q$
2 repeat
3    $OldL \leftarrow L$
4    $L \leftarrow \mathbf{Pre}(OldL \cap F) \setminus OldL$
5    $L \leftarrow BackwardBFS(L) \cup OldL$
6 until $L = OldL$
7 if $q_0 \in L$ 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 $L \leftarrow Q$
2 repeat
3    $OldL \leftarrow L$
4    for i=0 to $m - 1$
5       $L \leftarrow \mathbf{Pre}(OldL \cap F_i)$
6       $L \leftarrow BackwardBFS(L)$
7 until $L = OldL$
8 if $q_0 \in L$ 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 $\mathit{inf}(S)$ denote the states $q \in S$ such that some infinite path starting at q contains only states of S. Instead of computing $\mathbf{Pre}(\mathit{OldL} \cap F)$ at each iteration step, MEL computes $\mathbf{Pre}(\mathit{inf}(\mathit{OldL}) \cap F_i)$.

```
$MEL(A)$
Input: NGA $A = (Q, \Sigma, \delta, q_0, \{F_0, \ldots, F_{k-1}\})$
Output: EMP if $L_\omega(A) = \emptyset$, NEMP otherwise

$L \leftarrow Q$;
repeat
   $OldL \leftarrow L$
   $L \leftarrow inf(OldL)$
   $L \leftarrow \mathbf{Pre}(L \cap F)$
   $L \leftarrow BackwardBFS(L)$
until $L = OldL$
if $q_0 \in L$ then report NEMP
else report NEMP

function $inf(S)$
   repeat
      $OldS \leftarrow S$
      $S \leftarrow S \cap \mathbf{Pre}(S)$
   until $S = OldS$
   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 $D_{2i+1} := \mathtt{inf}(D_{2i})$ and $D_{2i+2} := \mathtt{pre}^+(D_{2i+1} \cap \alpha)$. 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 13.9. 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.

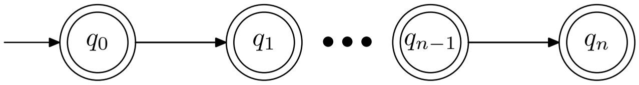


Figure 13.9: An example in which the MEL-algorithm outperforms the Emerson-Lei algorithm

A good case for Emerson-Lei's algorithm. Consider the automaton of Figure 13.10. 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.

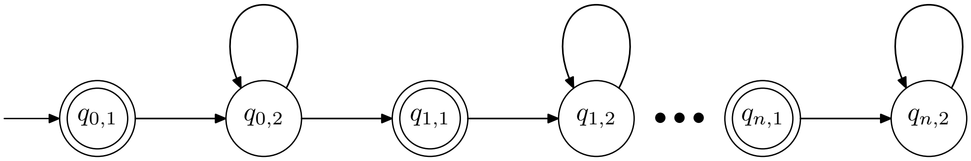
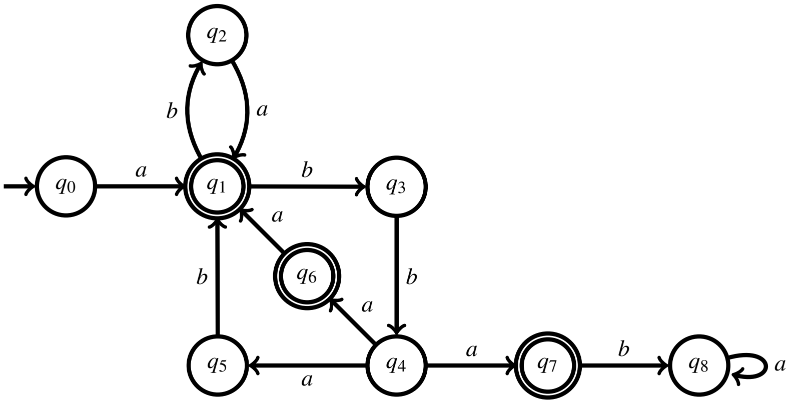


Figure 13.10: An example in which the EL-algorithm outperforms the MEL-algorithm

Exercises

Exercise 139 Let B be the following Bchi automaton:



1. Execute the emptiness algorithm *NestedDFS* on B.

2. Recall that *NestedDFS* is a non deterministic algorithm and different choices of runs may return different lassos. Which lassos of B can be found by *NestedDFS*?

3. Show that *NestedDFS* is non optimal by exhibiting some search sequence on B.

4. Execute the emptiness algorithm *TwoStack* on B.

5. Which lassos of B can be found by *TwoStack*?

Exercise 140 A Bchi automaton is weak if none of its strongly connected components contains both accepting and non-accepting states. Give an emptiness algorithm for weak Bchi automata. What is the complexity of the algorithm?

Exercise 141 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 142 (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
$L \leftarrow Q$
**repeat**
    $OldL \leftarrow L$
    $L \leftarrow \texttt{pre}^+(L \cap F, L)$
**until** $L = OldL$
**if** $q_0 \in L$ **then report** NEMP
**else report** NEMP
```

Is *MEL2* correct? What is the difference between the sequences of sets computed by *MEL* and *MEL2*?

Chapter 14

Applications I: 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 $c_0 c_1 c_2 \ldots$ 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.

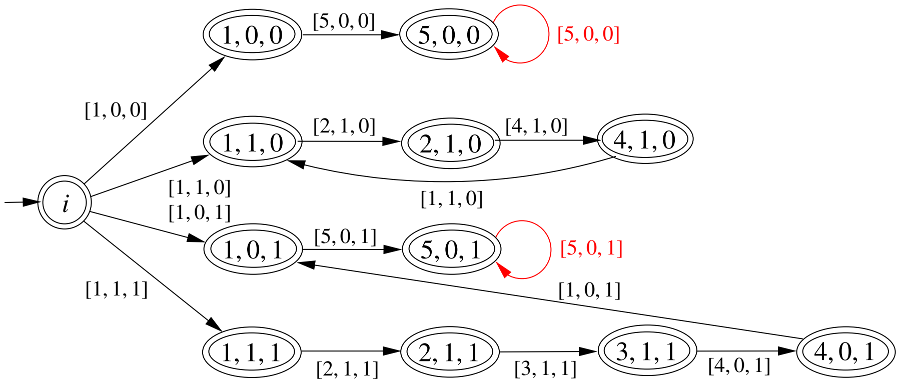


Figure 14.1: System NBA for the program

```
1 while $x = 1$ do
2    if $y = 1$ then
3       $x \leftarrow 0$
4    $y \leftarrow 1 - x$
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

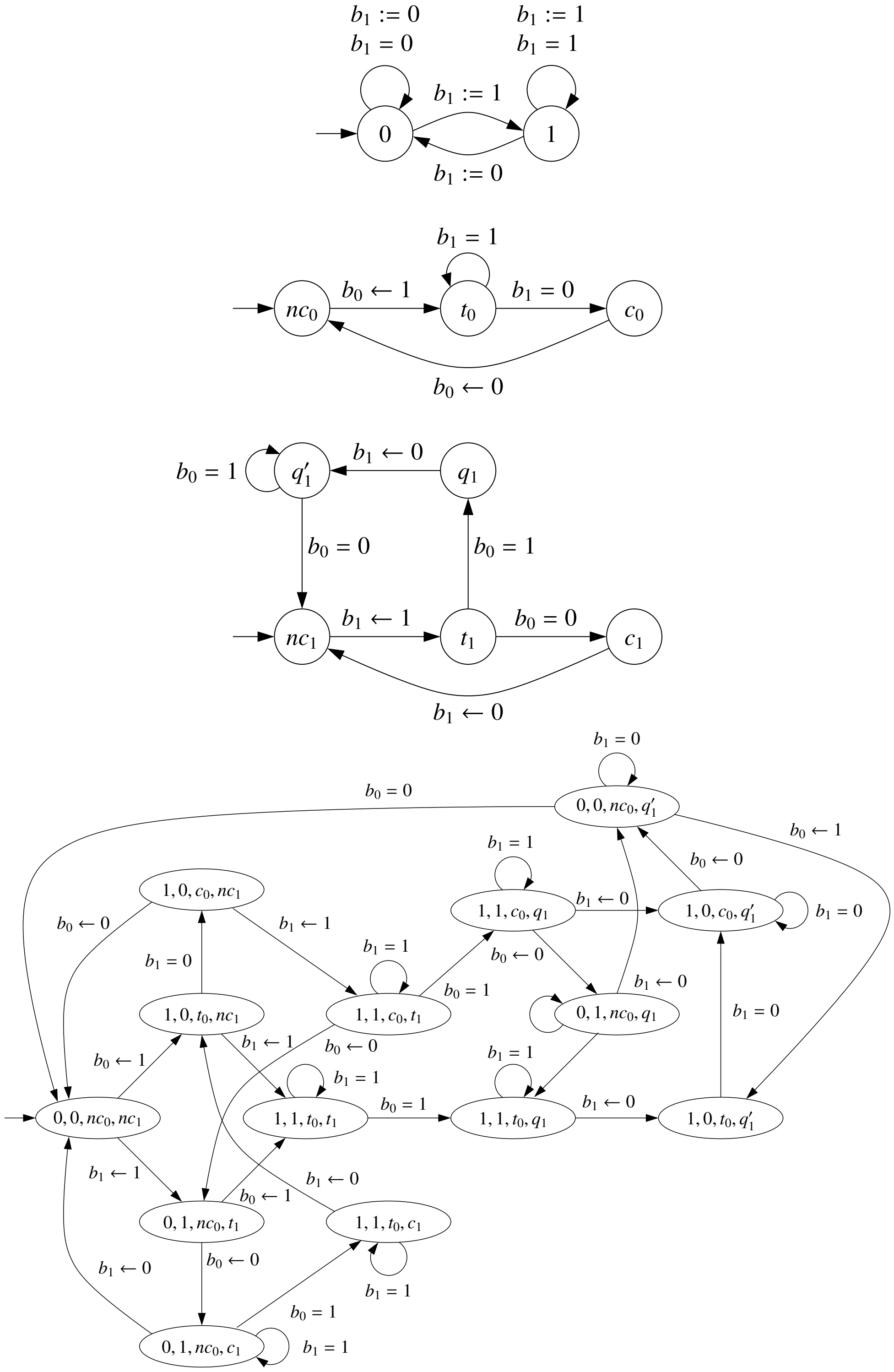


Figure 14.2: Lamport's algorithm and its asynchronous product.

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. We denote LTL(AP) the set of LTL formulas over AP.

Atomic propositions are combined by means of the usual Boolean operators and the temporal operators $\mathbf{X}$ ("next") and $\mathbf{U}$ ("until"). Intuitively, as a first approximation $\mathbf{X}\varphi$ means "φ holds at the next configuration" (the configuration reached after one step of the program), and $\varphi \, \mathbf{U} \, \psi$ means "φ holds until a configuration is reached satisfying ψ". Formally, the syntax of LTL(AP) is defined as follows:

Definition 14.2 *Let AP be a finite set of atomic propositions.* 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 atomic propositions satisfied by c_i is exactly σ_i. We now formally define when a computationm satisfies a formula.

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:

$$\begin{array}{rcl}
e_1 & = & [1,0,0]\ [5,0,0]^\omega \\
e_2 & = & (\ [1,1,0]\ [2,1,0]\ [4,1,0]\ )^\omega \\
e_3 & = & [1,0,1]\ [5,0,1]^\omega \\
e_4 & = & [1,1,1]\ [2,1,1]\ [3,1,1]\ [4,0,1]\ [1,0,1]\ [5,0,1]^\omega
\end{array}$$

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 $\mathcal{V}\colon AP \rightarrow 2^C$ 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

$$\begin{array}{rcl}
\sigma_1 & = & \{\texttt{at\_1}, \texttt{x=0}, \texttt{y=0}\}\ \{\texttt{at\_5}, \texttt{x=0}, \texttt{y=0}\}^\omega \\
\sigma_2 & = & (\ \{\texttt{at\_1}, \texttt{x=1}, \texttt{y=0}\}\ \{\texttt{at\_2}, \texttt{x=1}, \texttt{y=0}\}\ \{\texttt{at\_4}, \texttt{x=1}, \texttt{y=0}\}\ )^\omega \\
\sigma_3 & = & \{\texttt{at\_1}, \texttt{x=0}, \texttt{y=1}\}\ \{\texttt{at\_5}, \texttt{x=0}, \texttt{y=1}\}^\omega \\
\sigma_4 & = & \{\texttt{at\_1}, \texttt{x=1}, \texttt{y=1}\}\ \{\texttt{at\_2}, \texttt{x=1}, \texttt{y=1}\}\ \{\texttt{at\_3}, \texttt{x=1}, \texttt{y=1}\}\ \{\texttt{at\_4}, \texttt{x=0}, \texttt{y=1}\} \\
& & \{\texttt{at\_1}, \texttt{x=0}, \texttt{y=1}\}\ \{\texttt{at\_5}, \texttt{x=0}, \texttt{y=1}\}^\omega
\end{array}$$

We give some examples of properties:

- $\varphi_0 = \texttt{x=1} \wedge \mathbf{X}\texttt{y=0} \wedge \mathbf{XX}\texttt{at\_4}$. 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 $\sigma_2 \models \varphi_0$, and $\sigma_1, \sigma_3, \sigma_4 \not\models \varphi_0$.
- $\varphi_1 = \mathbf{F}\texttt{x=0}$. In natural language: x eventually gets the value 0. We have $\sigma_1, \sigma_2, \sigma_4 \models \varphi_1$, but $\sigma_3 \not\models \varphi_1$.
- $\varphi_2 = \texttt{x=0}\ \mathbf{U}\ \texttt{at\_5}$. 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 $\sigma_1, \sigma_3 \models \varphi_2$ and $\sigma_2, \sigma_4 \not\models \varphi_2$.
- $\varphi_3 = \texttt{y=1} \wedge \mathbf{F}(\texttt{y=0} \wedge \texttt{at\_5}) \wedge \neg(\mathbf{F}(\texttt{y=0} \wedge \mathbf{X}\texttt{y=1}))$. 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 $\psi \ \rightarrow \ \varphi$.

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.

□

Example 14.6 Formally speaking, it is not correct to say "$\mathbf{X}\varphi$ means that the next configuration satisfies φ" or "$\varphi\ \mathbf{U}\ \psi$ means that some future configuration satisfies ψ, and until then all configurations satisfy φ". The reason is that formulas do not hold at configurations, but at computations. Correct is: "the suffix of the computation starting at the next configuration (which is also a computation) satisfies φ", and "some suffix of the computation satisfies ψ, and until then all suffixes satisfy φ.

To illustrate this point, let $AP = \{p, q\}$, and consider the formula $\varphi = \mathbf{GF}p\ \mathbf{U}\ q$. Then the computation

$$\tau = \emptyset\,\emptyset\,\{q\}\,\emptyset\,\{p\}\,\emptyset^\omega$$

satisfies φ. Indeed, the suffix $\{q\}\,\emptyset\,\{p\}\,\emptyset^\omega$ satisfies q, and all "larger" suffixes, that is, $\emptyset\,\{q\}\,\emptyset\,\{p\}\,\emptyset^\omega$ and τ itself, satisfy $\mathbf{F}p$. □

14.3 From LTL formulas to generalized Büchi automata

We present an algorithm that, given a formula $\varphi \in \mathrm{LTL}(AP)$ 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 define the satisfaction sequence and the Hintikka sequence of a computation σ and a formula φ. We first need to introduce the notions of *closure* of a formula, and *atom* of the closure.

Definition 14.7 *Given a formula φ, the* negation *of φ is the formula ψ if $\varphi = \neg\psi$, and the formula $\neg\varphi$ otherwise. The* closure *$cl(\varphi)$ of a formula φ is the set containing all subformulas of φ and their negations. A nonempty set $\alpha \subseteq cl(\varphi)$ is an* atom *of $cl(\varphi)$ if it satisfies the following properties:*

(a0) If **true** $\in cl(\varphi)$*, then* **true** $\in \alpha$.

(a1) For every $\varphi_1 \wedge \varphi_2 \in cl(\varphi)$*:* $\varphi_1 \wedge \varphi_2 \in \alpha$ *if and only if* $\varphi_1 \in \alpha$ *and* $\varphi_2 \in \alpha$.

(a2) For every $\neg\varphi_1 \in cl(\varphi)$*:* $\neg\varphi_1 \in \alpha$ *if and only if* $\varphi_1 \notin \alpha$.

The set of all atoms of $cl(\varphi)$ *is denoted by* $at(\phi)$.

Observe that if α is the set of all formulas of $cl(\varphi)$ satisfied by a computation σ, then α is necessarily an atom. Indeed, every computation satisfies **true**; if a computation satisfies the conjunction of two formulas, then it satisfies each of the conjuncts; finally, if a computation satisfies a formula, then it does not satisfy its negation, and vice versa. Notice as well that, because of (a2), if $cl(\varphi)$ contains k formulas, then every atom of $cl(\varphi)$ contains exactly $k/2$ formulas.

Example 14.8 The closure of the formula $p \wedge (p\ \mathbf{U}\ q)$ is

$$\{\ p,\ \neg p,\ q, \neg q,\ p\ \mathbf{U}\ q\ , \neg(p\ \mathbf{U}\ q),\ p \wedge (p\ \mathbf{U}\ q), \neg(p \wedge (p\ \mathbf{U}\ q))\ \}\ .$$

We claim that the only two atoms containing $p \wedge (p\ \mathbf{U}\ q)$ are

$$\{\ p,\ q,\ p\ \mathbf{U}\ q,\ p \wedge (p\ \mathbf{U}\ q)\ \} \quad \text{and} \quad \{\ p,\ \neg q,\ p\ \mathbf{U}\ q,\ p \wedge (p\ \mathbf{U}\ q)\ \}\ .$$

Let us see why. By (a2), an atom always contains either a subformula or its negation, but not both. So in principle there are 16 possibilities for atoms, since we have to choose exactly one of p and $\neg p$, q and $\neg q$, $p\ \mathbf{U}\ q$ and $\neg(p\ \mathbf{U}\ q)$, and $p \wedge (p\ \mathbf{U}\ q)$ and $\neg(p \wedge (p\ \mathbf{U}\ q))$. Since we look for atoms containing $p \wedge (p\ \mathbf{U}\ q)$, we are left with 8 possibilities. But, by (a1), every atom α containing $p \wedge (p\ \mathbf{U}\ q)$ must contain both p and $p\ \mathbf{U}\ q$. So the only freedom left is the possibility to choose q or $\neg q$. None of these choices violates any of the conditions, and so exactly two atoms contain $p \wedge (p\ \mathbf{U}\ q)$. □

Definition 14.9 *The* satisfaction sequence *for a computation* σ *and a formula* φ *is the infinite sequence of atoms*

$$sats(\sigma, \varphi) = sats(\sigma, \varphi, 0)\ sats(\sigma, \varphi, 1)\ sats(\sigma, \varphi, 2)\ \ldots$$

where $sats(\sigma, \varphi, i)$ *is the atom containing the formulas of* $cl(\varphi)$ *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 at each moment.

Example 14.10 Let $\varphi = p\ \mathbf{U}\ q$, and consider the computations $\sigma_1 = \{p\}^\omega$, and $\sigma_2 = (\{p\}\ \{q\})^\omega$. We have

$$\begin{array}{rcl} sats(\sigma_1, \varphi) & = & \{\ p,\ \neg q,\ \neg(p\ \mathbf{U}\ q)\ \}^\omega \\ sats(\sigma_2, \varphi) & = & (\ \{p,\ \neg q,\ p\ \mathbf{U}\ q\ \}\ \{\ \neg p,\ q,\ p\ \mathbf{U}\ q\ \}\ )^\omega \end{array}$$

□

Observe that σ satisfies φ if and only if and only if $\varphi \in sats(\sigma, \varphi, 0)$, 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 determine whether a given 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 14.11 *A* pre-Hintikka sequence *for φ is an infinite sequence $\alpha_0\alpha_1\alpha_2\ldots$ of atoms satisfying the following conditions for every $i \geq 0$:*

(l1) For every $\mathbf{X}\varphi \in cl(\varphi)$: $\mathbf{X}\varphi \in \alpha_i$ if and only if $\varphi \in \alpha_{i+1}$.

(l2) For every $\varphi_1 \,\mathbf{U}\, \varphi_2 \in cl(\varphi)$: $\varphi_1 \,\mathbf{U}\, \varphi_2 \in \alpha_i$ if and only if $\varphi_2 \in \alpha_i$ or $\varphi_1 \in \alpha_i$ and $\varphi_1 \,\mathbf{U}\, \varphi_2 \in \alpha_{i+1}$.

A pre-Hintikka sequence is a Hintikka sequence *if it also satisfies*

(g) For every $\varphi_1 \,\mathbf{U}\, \varphi_2 \in \alpha_i$, there exists $j \geq i$ such that $\varphi_2 \in \alpha_j$.

A pre-Hintikka or Hintikka sequence α matches *a computation σ if $\sigma_i \subseteq \alpha_i$ for every $i \geq 0$.*

Observe that conditions (l1) and (l2) are *local*: in order to determine 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 distance between the indices i and j can be arbitrarily large.

Example 14.12 Let $\varphi = \neg(p \wedge q) \,\mathbf{U}\, (r \wedge s)$.

- Let $\alpha_1 = \{\ p,\ \neg q,\ r,\ s,\ \varphi\ \}$. The sequence α_1^ω is not a Hintikka sequence for φ, because α_1 is not an atom; indeed, by (a1) every atom containing r and s must contain $r \wedge s$.

- Let $\alpha_2 = \{\ \neg p,\ r,\ \neg\varphi\ \}^\omega$. The sequence α_2^ω is not a Hintikka sequence for φ, because α_2 is not an atom; indeed, by (a2) every atom mut contains either q or $\neg q$, and either s or $\neg s$.

- Let $\alpha_3 = \{\ \neg p,\ q,\ \neg r,\ s,\ r \wedge s,\ \varphi\}^\omega$. The sequence α_3^ω is not a Hintikka sequence for φ, because α_3 is not an atom; indeed, by (a2) every atom must contian either $(p \wedge q)$ or $\neg(p \wedge q)$.

- Let $\alpha_4 = \{\ p,\ q,\ (p \wedge q)\ r,\ s,\ r \wedge s,\ \neg\varphi\}$. The set α_4 is an atom, but the sequence α_4^ω is not a Hintikka sequence for φ, because it violates condition (l2): since α_4 contains $(r \wedge s)$, it must also contain φ.

- Let $\alpha_5 = \{\ p,\ \neg q,\ \neg(p \wedge q),\ \neg r,\ s,\ \neg(r \wedge s),\ \varphi\ \}^\omega$. The set α_5 is an atom, and the sequence α_5^ω is a pre-Hintikka sequence. However, it is not a Hintikka sequence because it violates condition (g): since α_5 contains φ, some atom in the sequence must contain $(r \wedge s)$, which is not the case.

- Let $\alpha_6 = \{\ p,\ q,\ (p \wedge q),\ r,\ s,\ (r \wedge s),\ \varphi\ \}$. The sequence $(\ \alpha_5\ \alpha_6\ )^\omega$ is a Hinktikka sequence for φ.

□

It follows immediately from the definition of a Hintikka sequence 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 is a satisfaction sequence.

Theorem 14.13 *Let σ be a computation and let φ be a formula. The unique Hintikka sequence for φ matching σ is the satisfaction sequence $sats(\sigma, \varphi)$.*

Proof: As observed above, it follows immediately from the definitions that $sats(\sigma, \varphi)$ is a Hintikka sequence for φ matching σ. To show that no other Hintikka sequence matches $sats(\sigma, \varphi)$, 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 $cl(\varphi)$. We prove that for every $i \geq 0$: $\psi \in \alpha_i$ if and only if $\psi \in sats(\sigma, \varphi, i)$.

The proof is by induction on the structure of ψ.

- $\psi = \mathbf{true}$. Then $\mathbf{true} \in sats(\sigma, \varphi, i)$ 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 $p \in sats(\sigma, \varphi, i)$. So $p \in \alpha_i$ if and only if $p \in sats(\sigma, \varphi, i)$.
- $\psi = \varphi_1 \wedge \varphi_2$. We have

$$\begin{array}{lll} & \varphi_1 \wedge \varphi_2 \in \alpha_i & \\ \Leftrightarrow & \varphi_1 \in \alpha_i \text{ and } \varphi_2 \in \alpha_i & \text{(condition (a1))} \\ \Leftrightarrow & \varphi_1 \in sats(\sigma, \varphi, i) \text{ and } \varphi_2 \in sats(\sigma, \varphi, i) & \text{(induction hypothesis)} \\ \Leftrightarrow & \varphi_1 \wedge \varphi_2 \in sats(\sigma, \varphi, i) & \text{(definition of } sats(\sigma, \varphi)) \end{array}$$

- $\psi = \neg\varphi_1$ or $\psi = \mathbf{X}\varphi_1$. The proofs are very similar to the last one.
- $\psi = \varphi_1\ \mathbf{U}\ \varphi_2$. We prove:

  (a) If $\varphi_1\ \mathbf{U}\ \varphi_2 \in \alpha_i$, then $\varphi_1\ \mathbf{U}\ \varphi_2 \in sats(\sigma, \varphi, i)$.
  By condition (l2) of the definition of a Hintikka sequence, we have to consider two cases:

  - $\varphi_2 \in \alpha_i$. By induction hypothesis, $\varphi_2 \in sats(\sigma, \varphi)$, and so $\varphi_1\ \mathbf{U}\ \varphi_2 \in sats(\sigma, \varphi, i)$.
  - $\varphi_1 \in \alpha_i$ and $\varphi_1\ \mathbf{U}\ \varphi_2 \in \alpha_{i+1}$. By condition (g), there is at least one index $j \geq i$ such that $\varphi_2 \in \alpha_j$. Let j_m be the smallest of these indices. We prove the result by induction on $j_m - i$. If $i = j_m$, then $\varphi_2 \in \alpha_j$, and we proceed as in the case $\varphi_2 \in \alpha_i$. If $i < j_m$, then since $\varphi_1 \in \alpha_i$, we have $\varphi_1 \in sats(\sigma, \varphi, i)$ (induction on ψ). Since $\varphi_1\ \mathbf{U}\ \varphi_2 \in \alpha_{i+1}$,

we have either $\varphi_2 \in \alpha_{i+1}$ or $\varphi_1 \in \alpha_{i+1}$. In the first case we have $\varphi_2 \in sats(\sigma, \varphi, i+1)$, and so $\varphi_1 \mathbf{U} \varphi_2 \in sats(\sigma, \varphi, i)$. In the second case, by induction hypothesis (induction on $j_m - i$), we have $\varphi_1 \mathbf{U} \varphi_2 \in sats(\sigma, \varphi, i+1)$, and so $\varphi_1 \mathbf{U} \varphi_2 \in sats(\sigma, \varphi, i)$.

(b) If $\varphi_1 \mathbf{U} \varphi_2 \in sats(\sigma, \varphi, i)$, then $\varphi_1 \mathbf{U} \varphi_2 \in \alpha_i$.
We consider again two cases.

- $\varphi_2 \in sats(\sigma, \varphi, i)$. By induction hypothesis, $\varphi_2 \in \alpha_i$, and so $\varphi_1 \mathbf{U} \varphi_2 \in \alpha_i$.
- $\varphi_1 \in sats(\sigma, \varphi, i)$ and $\varphi_1 \mathbf{U} \varphi_2 \in sats(\sigma, \varphi, i+1)$. By the definition of a satisfaction sequence, there is at least one index $j \geq i$ such that $\varphi_2 \in sats(\sigma, \varphi, j)$. Proceed now as in case (a).

□

14.3.2 Constructing the NGA for an LTL formula

Given a formula φ, we construct a *generalized* Büchi automaton A_φ recognizing $L(\varphi)$. By the definition of a satisfaction sequence, a computation σ satisfies φ if and only if $\varphi \in sats(\sigma, \varphi, 0)$. Moreover, by Theorem 14.13 $sats(\sigma, \varphi)$ is the (unique) Hintikka sequence for φ matching σ. So A_φ must recognize the computations σ satisfying: the first atom of the unique Hintikka sequence for φ matching σ contains φ.

To achieve this, we apply the following strategy:

(a) Define the states and transitions of the automaton so that the runs of A_φ are all the sequences

$$\alpha_0 \xrightarrow{\sigma_0} \alpha_1 \xrightarrow{\sigma_1} \alpha_2 \xrightarrow{\sigma_2} \ldots$$

such that $\sigma = \sigma_0 \sigma_1 \ldots$ is a computation, and $\alpha = \alpha_0 \alpha_1 \ldots$ is a pre-Hintikka sequence of φ matching σ.

(b) Define the sets of accepting states of the automaton (recall that A_φ is a NGA) so that a run is accepting if and only its corresponding pre-Hintikka sequence is also a Hintikka sequence.

Condition (a) determines the alphabet, states, transitions, and initial state of A_φ:

- The alphabet of A_φ is 2^{AP}.
- The states of A_φ are atoms of φ.
- The initial states are the atoms α such that $\varphi \in \alpha$.
- The output transitions of a state α (where α is an atom) are the triples $\alpha \xrightarrow{\sigma} \beta$ such that σ matches α, 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 condition (b). By the definition of a Hintikka sequence, we must guarantee that in every run $\alpha_0 \xrightarrow{\sigma_0} \alpha_1 \xrightarrow{\sigma_1} \alpha_2 \xrightarrow{\sigma_2} \ldots$, if any α_i contains a subformula $\varphi_1 \mathbf{U} \varphi_2$, then there is $j \geq i$ such that $\varphi_2 \in \alpha_j$. By condition (l2), this amounts to guaranteeing that every run contains infinitely many indices i such that $\varphi_2 \in \alpha_i$, or infinitely many indices j such that $\neg(\varphi_1 \mathbf{U} \varphi_2) \in \alpha_j$. So we choose the sets of accepting states as follows:

- The accepting condition contains a set $F_{\varphi_1 \mathbf{U} \varphi_2}$ of accepting states for each subformula $\varphi_1 \mathbf{U} \varphi_2$ of φ. An atom belongs to $F_{\varphi_1 \mathbf{U} \varphi_2}$ if it does not contain $\varphi_1 \mathbf{U} \varphi_2$ or if it contains φ_2.

The pseudocode for the translation algorithm is shown below.

```
*LTLtoNGA*($\varphi$)
**Input:** formula $\varphi$ of AP
**Output:** NGA $A_\varphi = (Q, 2^{AP}, Q_0, \delta, \mathcal{F})$ with $L\left(A_\varphi\right) = L(\varphi)$
$Q_0 \leftarrow \{\alpha \in at(\phi) \mid \varphi \in \alpha\}$; $Q \leftarrow \emptyset$;$\delta \leftarrow \emptyset$
$W \leftarrow Q_0$
while $W \neq \emptyset$ do
   pick $\alpha$ from $W$
   add $\alpha$ to $Q$
   for all $\varphi_1 \mathbf{U} \varphi_2 \in cl(\varphi)$ do
      if $\varphi_1 \mathbf{U} \varphi_2 \notin \alpha$ or $\varphi_2 \in \alpha$ then add $\alpha$ to $F_{\varphi_1 \mathbf{U} \varphi_2}$
   for all $\beta \in at(\phi)$ do
      if $\alpha$,$\beta$ satisfies (l1) and (l2) then
         add $(\alpha, \alpha \cap AP, \beta)$ to $\delta$
         if $\beta \notin Q$ then add $\beta$ to $W$
$\mathcal{F} \leftarrow \emptyset$
for all $\varphi_1 \mathbf{U} \varphi_2 \in cl(\varphi)$ do $\mathcal{F} \leftarrow \mathcal{F} \cup \{F_{\varphi_1 \mathbf{U} \varphi_2}\}$
return $(Q, 2^{AP}, Q_0, \delta, \mathcal{F})$
```

Example 14.14 We construct the automaton A_φ for the formula $\varphi = p \mathbf{U} q$. The closure $cl(\varphi)$ has eight atoms, corresponding to all the possible ways of choosing between p and $\neg p$, q and $\neg q$, and $p \mathbf{U} q$ 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 those transitions would violate condition (l2). So these states can be removed, and we are left with the five atoms shown in Figure 14.3. The three atoms on the left contain $p \mathbf{U} q$, and so they become the initial states. Figure 14.3 uses some conventions to simplify the graphical representation. Observe that every transition of A_φ leaving an atom α is labeled by $\alpha \cap AP$. For instance, all transitions leaving the state $\{\neg p,\ q,\ p \mathbf{U} q\}$ are labeled with $\{q\}$, and all transitions leaving $\{\neg p,\ \neg q,\ \neg(p \mathbf{U} q)\}$ are labeled with $\emptyset$. Therefore, since the label of a transition can be deduced from its source state, we omit them in the figure. Moreover, since φ only has one subformula of the form $\varphi_1 \, U \, \varphi_2$, the NGA is in fact a NBA, and we can represent the accepting states as for NBAs. The accepting states of $F_{p \mathbf{U} q}$ are the atoms that do not

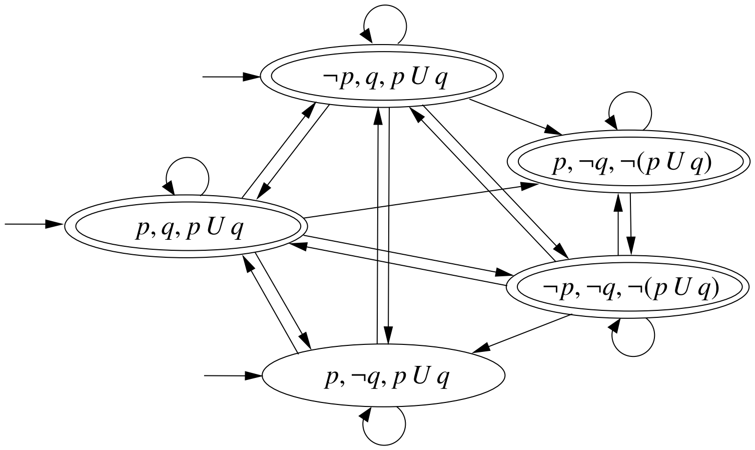


Figure 14.3: NGA (NBA) for the formula p **U** q.

contain p **U** q—the two atoms on the right—and the atoms containing q—the leftmost atom and the atom at the top.

Consider for example the atoms $\alpha = \{\neg p, \neg q, \neg(p \mathbf{U} q)\}$ 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 $X\psi$, while condition (l2) holds because $p \mathbf{U} q \notin \alpha$ and $q \notin \beta$ and $p \notin \alpha$. On the other hand, 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 $p \mathbf{U} q \in \alpha$. □

NGAs obtained from LTL formulas by means of *LTLtoNGA* have a very particular structure:

- As observed above, all transitions leaving a state carry the same label.
- Every computation accepted by the NGA has one single accepting run.
  By the definiiton of the NGA, if $\alpha_0 \xrightarrow{\sigma_0} \alpha_1 \xrightarrow{sigma_1} \cdots$ is an accepting run, then $\alpha_0\, \alpha_1\, \alpha_2 \ldots$ is the satisfaction sequence of $\sigma_0\, \sigma_1\, \sigma_2 \ldots$. Since the satisfaction sequence of a given computation is by definition unique, there can be only an accepting run.
- The sets of computations recognized by any two distinct states of the NGA are disjoint.
  Let σ be a computation, and let $sats(\sigma, \varphi) = sats(\sigma, \varphi, 0)\, sats(\sigma, \varphi, 1) \ldots$ be its satisfaction sequence. Then σ is only accepted from the state $sats(\sigma, \varphi, 0)$.

14.3.3 Size of the NGA

Let n be the length of the formula φ. It is easy to see that the set $cl(\varphi)$ has size $\mathcal{O}(n)$. Therefore, the NGA A_φ has at most $O(2^n)$ states. Since φ contains at most n subformulas of the form $\varphi_1 \mathbf{U} \varphi_2$, the automaton 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 $\{\varphi_n\}_{n\geq 1}$ such that φ_n has length $\mathcal{O}(n)$, and every NGA recognizing $L_\omega(\varphi_n)$ has at least 2^n states. For this, we exhibit a family $\{D_n\}_{n\geq 1}$ of ω-languages over an alphabet Σ such that for every $n \geq 0$:

(1) every NGA recognizing D_n has at least 2^n states; and

(2) there is a formula $\varphi_n \in \text{LTL}(\Sigma)$ of length $\mathcal{O}(n)$ such that $L_\omega(\varphi_n) = D_n$.

Notice that in (2) we are abusing language, because if $\varphi_n \in \text{LTL}(\Sigma)$, then $L_\omega(\varphi_n)$ contains words over the alphabet 2^Σ, and so $L_\omega(\varphi_n)$ and D_n are languages over different alphabets. With $L_\omega(\varphi_n) = D_n$ we mean that for every computation $\sigma \in (2^\Sigma)^\omega$ we have $\sigma \in L_\omega(\varphi_n)$ iff $\sigma = \{a_1\}\{a_2\}\{a_3\}\cdots$ for some ω-word $a_1\, a_2\, a_3 \ldots \in D_n$.

We let $\Sigma = \{0, 1, \#\}$ and choose the language D_n as follows:

$$D_n = \{w\, w\, \#^\omega \mid w \in \{0,1\}^n\}$$

(1) Every NGA recognizing D_n has at least 2^n states.
Assume that a generalized Büchi automaton $A = (Q, \{0,1,\#\}, \delta, q_0, \{F_1, \ldots, F_k\})$ with $|Q| < 2^n$ recognizes D_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 pigeonhole principle we have $q_{w_1} = q_{w_2}$ 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.

(2) There is a formula $\varphi_n \in \text{LTL}(\Sigma)$ of length $\mathcal{O}(n)$ such that $L_\omega(\varphi_n) = D_n$.
We first construct the following auxiliary formulas:

- $\varphi_{n1} = \mathbf{G}(\ (0 \vee 1 \vee \#) \wedge \neg(0 \wedge 1) \wedge \neg(0 \wedge \#) \wedge \neg(1 \wedge \#)\ )$.
This formula expresses that at every position exactly one atomic proposition holds.
- $\varphi_{n2} = \neg\,\# \wedge \left(\bigwedge_{i=1}^{2n-1} \mathbf{X}^i \neg\,\#\right) \wedge \mathbf{X}^{2n}\mathbf{G}\,\#$.
This formula expresses that # does not hold at any of the first $2n$ positions, and it holds at all later positions.
- $\varphi_{n3} = \mathbf{G}(\ (0 \rightarrow \mathbf{X}^n(0 \vee \#))\ \wedge\ (1 \rightarrow \mathbf{X}^n(1 \vee \#))\ )$.
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, $\varphi_n = \varphi_{n1} \wedge \varphi_{n2} \wedge \varphi_{n3}$ is the formula we are looking for. Observe that φ_n contains $\mathcal{O}(n)$ 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 $\nu\colon AP \to 2^C$, 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_ν for the *negation* of the formula φ. A_ν recognizes all the computations that *violate* φ.

(2) Compute a NGA $A_\nu \cap A_s$ recognizing the executable computations of the system that violate the formula.

(3) Check emptiness of $A_\nu \cap A_s$.

Step (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_ν and A_s are different: the alphabet of A_ν is 2^{AP}, while the alphabet of A_s is the set C of configurations. By applying the valuation ν we transform A_ν into an automaton with C as alphabet. Since all the states of system NBAs are accepting, the automaton $A_\nu \cap A_s$ can be computed by *interNFA*.

It is important to observe that the three steps can be carried out simultaneously. The states of $A_\nu \cap A_s$ are pairs $[\alpha, c]$, where α is an atom of φ, and c is a configuration. The following algorithm takes a pair $[\alpha, c]$ as input and returns its successors in the NGA $A_\nu \cap A_s$. The algorithm 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 (11) and (12). The successors of $[\alpha, c]$ are the pairs $[\beta, c']$.

```
Succ([$\alpha, c$])
S ← ∅
   for all $c' \in \delta_s(c)$ do
   P ← ∅
   for all $p \in AP$ do
      if $c' \in \nu(p)$ then add $p$ to $P$
   for all $\beta \in at(\phi)$ matching $P$ do
      if $\alpha, \beta$ satisfies (11) and (12) then add $c'$ to $S$
return S
```

This algorithm 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 *Succ*:

```
6      for all $[\beta, c'] \in Succ([\alpha, c])$ do
```

Exercises

Exercise 143 Prove formally the following equivalences:

1. $\neg \mathbf{X}\varphi \equiv \mathbf{X}\neg\varphi$
2. $\neg \mathbf{F}\varphi \equiv \mathbf{G}\neg\varphi$
3. $\neg \mathbf{G}\varphi \equiv \mathbf{F}\neg\varphi$
4. $\mathbf{XF}\varphi \equiv \mathbf{FX}\varphi$
5. $\mathbf{XG}\varphi \equiv \mathbf{GX}\varphi$

Exercise 144 (Santos Laboratory). The *weak until* operator $\mathbf{W}$ 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 $\sigma^k \models \phi_1$ 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 145 Let $AP = \{p, q\}$ and let $\Sigma = 2^{AP}$. Give LTL formulas defining the following languages:

1. $\{p, q\}\ \emptyset\ \Sigma^\omega$
2. $\Sigma^*\ (\{p\} + \{p, q\})\ \Sigma^*\ \{q\}\ \Sigma^\omega$
3. $\Sigma^*\ \{q\}^\omega$
4. $\{p\}^*\ \{q\}^*\ \emptyset^\omega$

Exercise 146 (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 true between q and r.
- p precedes q before r.
- p precedes q after r.

- after p and q eventually r.
- p alternates between true and false.

Exercise 147 Let $AP = \{p, q\}$ and let $\Sigma = 2^{AP}$. Give Büchi automata for the ω-languages over Σ defined by the following LTL formulas:

1. $\mathbf{XG}\neg p$
2. $(\mathbf{GF}p) \rightarrow (\mathbf{F}q)$
3. $p \wedge \neg(\mathbf{XF}p)$
4. $\mathbf{G}(p \,\mathbf{U}\, (p \rightarrow q))$
5. $\mathbf{F}q \rightarrow (\neg q \,\mathbf{U}\, (\neg q \wedge p))$

Exercise 148 ich of the following equivalences hold?

1. $\mathbf{X}(\varphi \vee \psi) \equiv \mathbf{X}\varphi \vee \mathbf{X}\psi$
2. $\mathbf{X}(\varphi \wedge \psi) \equiv \mathbf{X}\varphi \wedge \mathbf{X}\psi$
3. $\mathbf{X}(\varphi \,\mathbf{U}\, \psi) \equiv (\mathbf{X}\varphi \,\mathbf{U}\, \mathbf{X}\psi)$
4. $\mathbf{F}(\varphi \vee \psi) \equiv \mathbf{F}\varphi \vee \mathbf{F}\psi$
5. $\mathbf{F}(\varphi \wedge \psi) \equiv \mathbf{F}\varphi \wedge \mathbf{F}\psi$
6. $\mathbf{G}(\varphi \vee \psi) \equiv \mathbf{G}\varphi \vee \mathbf{G}\psi$
7. $\mathbf{G}(\varphi \wedge \psi) \equiv \mathbf{G}\varphi \wedge \mathbf{G}\psi$
8. $\mathbf{GF}(\varphi \vee \psi) \equiv \mathbf{GF}\varphi \vee \mathbf{GF}\psi$
9. $\mathbf{GF}(\varphi \wedge \psi) \equiv \mathbf{GF}\varphi \wedge \mathbf{GF}\psi$
10. $\rho \,\mathbf{U}\, (\varphi \vee \psi) \equiv (\rho \,\mathbf{U}\, \varphi) \vee (\rho \,\mathbf{U}\, \psi)$
11. $(\varphi \vee \psi) \,\mathbf{U}\, \rho \equiv (\varphi \,\mathbf{U}\, \rho) \vee (\psi \,\mathbf{U}\, \rho)$
12. $\rho \,\mathbf{U}\, (\varphi \wedge \psi) \equiv (\varphi \,\mathbf{U}\, \rho) \wedge (\psi \,\mathbf{U}\, \rho)$
13. $(\varphi \wedge \psi) \,\mathbf{U}\, \rho \equiv (\varphi \,\mathbf{U}\, \rho) \wedge (\psi \,\mathbf{U}\, \rho)$

Exercise 149 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 $\mathbf{F}$ and $\mathbf{G}$.

Exercise 150 (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.

- $\mathbf{G}p \rightarrow \mathbf{F}p$
- $\mathbf{G}(p \rightarrow q) \rightarrow (\mathbf{G}p \rightarrow \mathbf{G}q)$
- $\mathbf{F}(p \wedge q) \leftrightarrow (\mathbf{F}p \wedge \mathbf{F}q)$
- $\neg\mathbf{F}p \rightarrow \mathbf{F}\neg\mathbf{F}p$
- $(\mathbf{G}p \rightarrow \mathbf{F}q) \leftrightarrow (p \,\mathbf{U}\, (\neg p \vee q))$

- $(\mathbf{FG}p \rightarrow \mathbf{GF}q) \leftrightarrow \mathbf{G}(p\ \mathbf{U}\ (\neg p \vee q))$
- $\mathbf{G}(p \rightarrow \mathbf{X}p) \rightarrow (p \rightarrow \mathbf{G}p)$.

Exercise 151 In this exercise we show how to construct a deterministic Büchi automaton for negation-free LTL formulas. Let φ be a formula of LTL$AP$ of atomic propositions, and let $\nu \in 2^{AP}$. We inductively define the formula $af(\varphi, \nu)$ as follows:

$$\begin{array}{rcl@{\qquad}rcl}
af(\mathbf{true}, \nu) & = & \mathbf{true} & af(\varphi \wedge \psi, \nu) & = & af(\varphi, \nu) \wedge af(\psi, \nu) \\
af(\mathbf{false}, \nu) & = & \mathbf{false} & af(\varphi \vee \psi, \nu) & = & af(\varphi, \nu) \vee af(\psi, \nu) \\
af(a, \nu) & = & \begin{cases} \mathbf{true} & \text{if } a \in \nu \\ \mathbf{false} & \text{if } a \notin \nu \end{cases} & \begin{array}{r} af(\mathbf{X}\varphi, \nu) \\ af(\varphi\ \mathbf{U}\ \psi, \nu) \end{array} & \begin{array}{c} = \\ = \end{array} & \begin{array}{l} \varphi \\ af(\psi, \nu) \vee (af(\varphi, \nu) \wedge \varphi\ \mathbf{U}\ \psi) \end{array} \\
af(\neg a, \nu) & = & \begin{cases} \mathbf{false} & \text{if } a \in \nu \\ \mathbf{true} & \text{if } a \notin \nu \end{cases} & & &
\end{array}$$

We extend the definition to finite words: $af(\varphi, \epsilon) = \varphi$; and $af(\varphi, \nu w) = af(af(\varphi, \nu), w)$ for every $\nu \in 2^{AP}$ and every finite word w. Prove:

(a) For every formula φ, finite word $w \in \left(2^{AP}\right)^*$ and ω-word $w' \in \left(2^{AP}\right)^\omega$:

$$w\,w' \models \varphi \text{ iff } w' \models af(\varphi, w)\ .$$

So, intuitively, $af(\varphi, w)$ is the formula that must hold "after reading w" so that φ holds "at the beginning" of the ω-word $w\,w'$.

(b) For every negation-free formula φ: $w \models \varphi$ iff $af(\varphi, w') \equiv \mathbf{true}$ for some finite prefix w' of w.

(c) For every formula φ and ω-word $w \in \left(2^{AP}\right)^\omega$: $af(\varphi, w)$ is a boolean combination of proper subformulas of φ.

(d) For every formula φ of length n: the set of formulas $\{af(\varphi, w) \mid w \in \left(2^{AP}\right)^*\}$ has at most 2^{2^n} equivalence classes up to LTL-equivalence.

(e) Use (b)-(d) to construct a deterministic Büchi automaton recognizing $L_\omega(\varphi)$ with at most 2^{2^n} states.

Exercise 152 In this exercise we show that the reduction algorithm of Exercise **??** does not reduce the Büchi automata generated from LTL formulas, and show that a little modification to *LTLtoNGA* can alleviate this problem.

Let φ be a formula of LTL(AP), and let $A_\varphi = LTLtoNGA(\varphi)$.

(1) Prove that the reduction algorithm of Exercise **??** does not reduce A, that is, show that $A = A/CSR$.

(2) Let B_φ be the result of modifying A_φ as follows:

- Add a new state q_0 and make it the unique initial state.
- For every initial state q of A_φ, add a transition $q_0 \xrightarrow{q \cap AP} q$ to B_φ (recall that q is an atom of $cl(\varphi)$, and so $q \cap AP$ is well defined).
- Replace every transition $q_1 \xrightarrow{q_1 \cap AP} q_2$ of A_φ by $q_1 \xrightarrow{q_2 \cap AP} q_2$.

Prove that $L_\omega(B_\varphi) = L_\omega(A_\varphi)$.

(3) Construct the automaton B_φ for the automaton of Figure 14.3.

(4) Apply the reduction algorithm of Exercise **??** to B_φ.

Exercise 153 (Kupferman and Vardi) We prove that, in the worst case, the number of states of the smallest deterministic Rabin automaton for an LTL formula may be double exponential in the size of the formula. Let $\Sigma_0 = \{a, b\}$, $\Sigma_1 = \{a, b, \#\}$, and $\Sigma = \{a, b, \#, \$\}$. For every $n \geq 0$ define the ω-language $L_n \subseteq \Sigma^\omega$ as follows (we identify an ω-regular expression with its language):

$$L_n = \sum_{w \in \Sigma_0^n} \Sigma_1^* \; \# \; w \; \# \; \Sigma_1^* \; \$ \; w \; \#^\omega$$

Informally, an ω-word belongs to L_n iff

- it contains one single occurrence of \$;
- the word to the left of \$ is of the form $w_0 \# w_1 \# \cdots \# w_k$ for some $k \geq 1$ and (possibly empty) words $w_0, \ldots, w_k \in \Sigma_0^*$;
- the ω-word to the right of \$ consists of a word $w \in \Sigma_0^n$ followed by an infinite tail $\#^\omega$, and
- w is equal to at least one of $w_0, \ldots, w_n$.

The exercise has two parts:

(1) Exhibit an infinite family $\{\varphi_n\}_{n \geq 0}$ of formulas of LTL(Σ) such that φ_n has size $\mathcal{O}(n^2)$ and $L_\omega(\varphi_n) = L_n$ (abusing language, we write $L_\omega(\varphi_n) = L_n$ for: $\sigma \in L_\omega(\varphi_n)$ iff $\sigma = \{a_1\}\{a_2\}\{a_3\}\cdots$ for some ω-word $a_1\, a_2\, a_3 \ldots \in L_n$).

(2) Show that the smallest DRA recognizing L_n has at least 2^{2^n} states.

The solution to the following two problems can be found in "The Blow-Up in Translating LTL to Deterministic Automata", by Orna Kupferman and Adin Rosenberg:

- Consider a variant L'_n of L_n in which each block of length n before the occurrence of \$ is prefixed by a binary encoding of its position in the block. Show that L'_n can be recognized by a formula of length $\mathcal{O}(n \log n)$ over a fixed-size alphabet, and that the smallest DRA recognizing it has at least 2^{2^n} states.

- Consider a variant L''_n of L_n in which each block of length n before the occurrence of \$ is prefixed by a different letter. (So every language L_n has a different alphabet.) Show that L''_n can be recognized by a formula of length $\mathcal{O}(n)$ over a linear size alphabet, and that the smallest DRA recognizing it has at least 2^{2^n} states.

Chapter 15

Applications II: Monadic Second-Order Logic and Linear Arithmetic

In Chapter 9 we showed that the languages expressible in monadic second-order logic on finite words were exactly the regular languages, and derived an algorithm that, given a formula, constructs an NFA accepting exactly the set of interpretations of the formula. We show that this result can be easily extended to the case of infinite words: in Section 15.1 we show that the languages expressible in monadic second-order logic on ω-words are exactly the ω-regular languages.

In Chapter 10 we introduced Presburger Arithmetic, a logical language for expressing properties of the integers, and showed how to construct for a given formula φ of Presburger Arithmetic an NFA A_φ recognizing the solutions of φ. In Section 15.2 we extend this result to Linear Arithmetic, a language for describing properties of real numbers with exactly the same syntax as Presburger arithmetic.

15.1 Monadic Second-Order Logic on ω-Words

Monadic second-oder logic on ω-words has the same syntax as and a very similar semantics to its counterpart on finite words.

Definition 15.1 *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 $(w, \mathfrak{I})$ where $w \in \Sigma^\omega$, and $\mathfrak{I}$ is a mapping that assigns every free first-order variable x a* position *$\mathfrak{I}(x) \in \mathbb{N}$ and every free second-order variable X a set of positions $\mathfrak{I}(X) \subseteq \mathbb{N}$. (The mapping may also assign positions to other variables.)*

The satisfaction relation $(w, \mathfrak{I}) \models \varphi$ between a formula φ of $MSO(\Sigma)$ and an interpretation $(w, \mathfrak{I})$ of φ is defined as follows:

$$\begin{array}{lcll}
(w, \mathfrak{I}) & \models & Q_a(x) & \textit{iff} \quad w[\mathfrak{I}(x)] = a \\
(w, \mathfrak{I}) & \models & x < y & \textit{iff} \quad \mathfrak{I}(x) < \mathfrak{I}(y) \\
(w, \mathfrak{I}) & \models & \neg\varphi & \textit{iff} \quad (w, \mathfrak{I}) \not\models \varphi \\
(w, \mathfrak{I}) & \models & \varphi 1 \vee \varphi_2 & \textit{iff} \quad (w, \mathfrak{I}) \models \varphi_1 \textit{ or } (w, \mathfrak{I}) \models \varphi_2 \\
(w, \mathfrak{I}) & \models & \exists x\, \varphi & \textit{iff} \quad |w| \geq 1 \textit{ and some } i \in \mathbb{N} \textit{ satisfies } (w, \mathfrak{I}[i/x]) \models \varphi \\
(w, \mathfrak{I}) & \models & x \in X & \textit{iff} \quad \mathfrak{I}(x) \in \mathfrak{I}(X) \\
(w, \mathfrak{I}) & \models & \exists X\, \varphi & \textit{iff} \quad \textit{some } S \subseteq \mathbb{N} \textit{ satisfies } (w, \mathfrak{I}[S/X]) \models \varphi
\end{array}$$

where $w[i]$ is the letter of w at position i, $\mathfrak{I}[i/x]$ is the interpretation that assigns i to x and otherwise coincides with $\mathfrak{I}$, and $\mathfrak{I}[S/X]$ is the interpretation that assigns S to X and otherwise coincides with $\mathfrak{I}$ — whether $\mathfrak{I}$ is defined for i, X or not. If $(w, \mathfrak{I}) \models \varphi$ we say that $(w, \mathfrak{I})$ 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 $L(\varphi) = \{w \in \Sigma^\omega \mid w \models \phi\}$. An ω-language $L \subseteq \Sigma^\omega$ is* MSO-definable *if $L = L(\varphi)$ for some formula $\varphi \in MSO(\Sigma)$.*

15.1.1 Expressive power of $MSO(\Sigma)$ on ω-words

We show that the ω-languages expressible in monadic second-order logic are exactly the ω-regular languages. The proof is very similar to its counterpartfor languages of finite words (Proposition 9.12), even a bit simpler.

Proposition 15.2 *If $L \subseteq \Sigma^\omega$ is regular, then L is expressible in $MSO(\Sigma)$.*

Proof: Let $A = (Q, \Sigma, \delta, Q_0, F)$ be a NBA with $Q = \{q_0, \ldots, q_n\}$ and $L(A) = L$. We construct a formula φ_A such that for every $w \in \Sigma^\omega$, $w \models \varphi_A$ iff $w \in L(A)$.

We start with some notations. Let $w = a_1\, a_2\, a_3 \ldots$ be an ω-word over Σ, and let

$$P_q = \left\{ i \in \mathbb{N} \mid q \in \hat{\delta}(q_0, a_0 \ldots a_i) \right\} .$$

In words, $i \in P_q$ iff A can be in state q immediately *after* reading the letter a_i. Then A accepts w iff $m \in \bigcup_{q \in F} P_q$.

We construct a formula $\text{Visits}(X_0, \ldots X_n)$ with free variables $X_0, \ldots X_n$ exactly as in Proposition 9.12. The formula has the property that $\mathfrak{I}(X_i) = P_{q_i}$ holds for *every* model $(w, \mathfrak{I})$ 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 $0 \leq i \leq n$. So we can take

$$\varphi_A := \exists X_0 \ldots \exists X_n \ \text{Visits}(X_0, \ldots X_n) \wedge \forall x\, \exists y \left(\bigvee_{q_i \in F} y \in X_i \right)$$

□

We proceed to prove that MSO-definable ω-languages are regular. Given a sentence $\varphi \in MS\,O(\Sigma)$, we encode an interpretation $(w, \mathfrak{I})$ as an ω-word. We proceed as for finite words. Consider for instance a formula with first-order variables x, y and second-order variables X, Y. Consider the interpretation

$$\left(a(ab)^\omega \; , \; \begin{array}{l} x \mapsto 2 \\ y \mapsto 6 \\ X \mapsto \text{set of prime numbers} \\ Y \mapsto \text{set of even numbers} \end{array} \right)$$

We encode it as

$$\begin{array}{c|cccccccccc}
 & a & a & b & a & b & a & b & a & b & \ldots \\
x & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots \\
y & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \ldots \\
X & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & \ldots \\
Y & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & \ldots
\end{array}$$

corresponding to the ω-word

$$\begin{bmatrix} a \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} a \\ 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} b \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \begin{bmatrix} a \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} b \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \begin{bmatrix} a \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} b \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \begin{bmatrix} a \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} b \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{matrix} \ldots \\ \\ \ldots \\ \\ \ldots \end{matrix} \qquad \text{over } \Sigma \times \{0,1\}^4$$

Definition 15.3 *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* $\Sigma \times \{0,1\}^\omega$ *described above. The ω-language of φ is $L_\omega(\varphi) = \{\mathrm{enc}(w, \mathfrak{I}) \mid (w, \mathfrak{I}) \models \varphi\}$.*

We can now prove by induction on the structure of φ that $L_\omega(\varphi)$ is ω-regular. The proof is a straightforward modification of the proof for the case of finite words. The case of negation requires to replace the complementation operation for NFAs by the complementation operation for NBAs.

15.2 Linear Arithmetic

Linear Arithmetic is a language for describing properties of real numbers. It has the same syntax as Presburger arithmetic (see Chapter 10), but formulas are interpreted over the reals, instead of over the naturals or the integers. Given a formula φ or real Presburger Arithmetic, we show how to construct a NBA A_φ recognizing the solutions of φ. Section 15.2.1 discusses how to encode real numbers as strings, and Section 15.3 constructs the NBA.

15.2.1 Encoding Real Numbers

We encode real numbers as infinite strings in two steps. First we encode reals as pairs of numbers, and then these pairs as strings.

We encode a real number $x \in \mathbb{R}$ as a pair (x_I, x_F), where $x_I \in \mathbb{Z}$, $x_F \in [0,1]$ and $x = x_I + x_F$. We call x_I and x_F the *integer* and *fractional parts* of x. So, for instance, $(1, 1/3)$ encodes $4/3$, and $(-1, 2/3)$ encodes $-1/3$ (*not* $-5/3$). Every integer is encoded by two different pairs, e.g., 2 is encoded by $(1,1)$ and $(2,0)$. We are not bothered by this. (In the standard decimal representation of real numbers integers also have two representations, for example 2 is represented by both 2 and $1.\overline{9}$.)

We encode pairs (x_I, x_F) as infinite strings $w_I \star w_F$. The string w_i is a 2-complement encoding of x_I (see Chapter 10). However, unlike Chapter 10, we use the *msbf* instead of the *lsbf* encoding (this is not essential, it leads to a more elegant construction.) So w_I is any word $w_I = a_n a_{n-1} \ldots a_0 \ldots \in \{0,1\}^*$ satisfying

$$x_I = \sum_{i=0}^{n-1} a_i \cdot 2^i - a_0 \cdot 2^n \tag{15.1}$$

The string w_F is any ω-word $b_1 b_2 b_3 \ldots \in \{0,1\}^\omega$ satisfying

$$x_F = \sum_{i=1}^{\infty} b_i \cdot 2^{-i} \tag{15.2}$$

The only word $b_1 b_2 b_3 \ldots$ for which we have $x_F = 1$ is 1^ω. So, in particular, the encodings of the integer 1 are $0^*01 \star 0^\omega$ and $0^*0 \star 1^\omega$. Equation 15.2 also has two solutions for fractions of the form 2^{-k}. For instance, the encodings of $1/2$ are $0^*0 \star 10^\omega$ and $0^*0 \star 01^\omega$. Other fractions have a unique encoding: $0^*0 \star (01)^\omega$ is the unique encoding of $1/3$.

Example 15.4 The encodings of 3 are $0^*011 \star 0^\omega$ and $0^*010 \star 1^\omega$.
The encodings of $3.\overline{3}$ are $0^*011 \star (01)^\omega$.
The encodings of -3.75 are $1 * 100 \star 010^\omega$ and $1 * 100 \star 001^\omega$. □

Tuples of reals are encoded using padding to make the $\star$-symbols fall on the same column. For instance, the encodings of the triple $(-6.75, 12.\overline{3}, 3)$ are

$$\left(\begin{bmatrix}1\\0\\0\end{bmatrix}\right)^* \begin{bmatrix}1\\1\\0\end{bmatrix} \begin{bmatrix}0\\1\\0\end{bmatrix} \begin{bmatrix}0\\0\\1\end{bmatrix} \begin{bmatrix}1\\0\\1\end{bmatrix} \begin{bmatrix}\star\\\star\\\star\end{bmatrix} \begin{bmatrix}0\\0\\0\end{bmatrix} \begin{bmatrix}1\\1\\0\end{bmatrix} \left(\begin{bmatrix}0\\0\\0\end{bmatrix} \begin{bmatrix}0\\1\\0\end{bmatrix}\right)^\omega$$

15.3 Constructing an NBA for the Real Solutions

Given a real Presburger formula φ, we construct a NBA A_φ accepting the encodings of the solutions of φ. If φ is a negation, disjunction, or existential qunatification, we proceed as in Chapter 10, replacing the operations on NFAs and transducres by operations on NBAs.

Consider now an atomic formula $\varphi = a \cdot x \leq b$. The NBA A_φ must accept the encodings of all the tuples $c \in \mathbb{R}^n$ satisfying $a \cdot c \leq b$. We decompose the problem into two subproblems for integer and fractional parts. Given $c \in \mathbb{R}^n$, let c_I and c_F be the integer and fractional part of c for some encoding of c. For instance, if $c = (2.\overline{3}, -2.75, 1)$, then we can have $c_I = (2, -3, 1)$ and $c_F = (0.\overline{3}, 0.25, 0)$, corresponding to the encoding, say,

$$[010 \star (01)^\omega,\ 111 \star 010^\omega,\ 01 \star 0^\omega]$$

or $c_I = (2, -3, 0)$ and $c_F = (0.\overline{3}, 0.25, 1)$, corresponding to, say,

$$[010 \star (01)^\omega,\ 11111 \star 001^\omega,\ 0 \star 1^\omega]\ .$$

Let α^+ (α^-) be the sum of the positive (negative) components of a; for instance, if $a = (1, -2, 0, 3, -1)$ then $\alpha^+ = 4$ and $\alpha^- = -3$. Since $c_F \in [0, 1]^n$, we have

$$\alpha^- \leq a \cdot c_F \leq \alpha^+ \tag{15.3}$$

and therefore, if c is a solution of φ, then

$$\begin{aligned} a \cdot c_I + a \cdot c_F &\leq b & (15.4)\\ a \cdot c_I &\leq b - \alpha^- & (15.5) \end{aligned}$$

Putting together 15.3-15.5, we get that $c_I + c_F$ is a solution of φ iff:

- $a \cdot c_I \leq b - \alpha^+$; or
- $a \cdot c_I \leq \beta$ for some integer $\beta \in [b - \alpha^+ + 1, b - \alpha^-]$ and $a \cdot c_F \leq b - \beta$.

If we denote $\beta^+ = b - \alpha^+ + 1$ and $\beta^- = b - \alpha^-$, then we can decompose the solution space of φ as follows:

$$\begin{aligned} Sol(\varphi) \quad = \quad & \{c_I + c_F \mid a \cdot c_I < \beta^+\} \\ \cup \quad & \bigcup_{\beta^+ \leq \beta \leq \beta^-} \{\, c_I + c_F \ \mid \ a \cdot c_I = \beta \text{ and } ; a \cdot c_F \leq b - \beta \,\} \end{aligned}$$

Example 15.5 We use $\varphi = 2x - y \leq 0$ as running example. We have

$$\begin{array}{ll} \alpha^+ = 2 & \alpha^- = 0 \\ \beta^+ = -1 & \beta^- = 0 \ . \end{array}$$

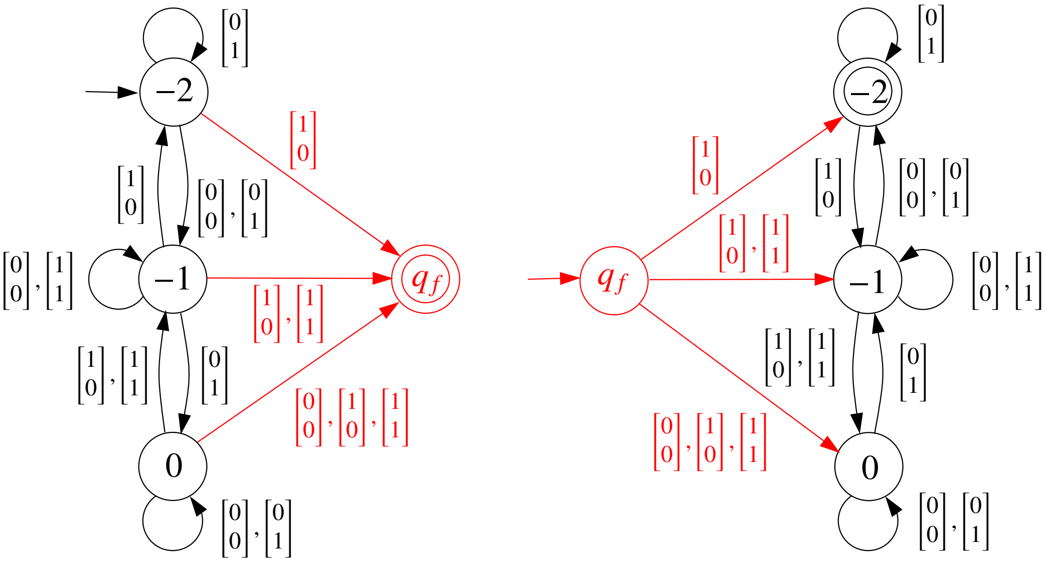
So $x, y \in \mathbb{R}$ is a solution of φ iff:

- $2x_I - y_I \leq -2$; or
- $2x_I - y_I = -1$ and $2x_F - y_F \leq 1$; or
- $2x_I - y_I = 0$ and $2x_F - y_F \leq 0$.

□

The solutions of $a \cdot c_I < \beta^+$ and $a \cdot c_I = \beta$ can be computed using algorithms *IneqZNFA* and *EqZNFA* of Section 10.3. Recall that both algorithms use the *lsbf* encoding, but it is easy to transform their output into NFAs for the *msbf* encoding: since the algorithms deliver NFAs with exactly one final state, it suffices to *reverse* the transitions of the NFA, and exchange the initial and final states: the new automaton recognizes a word w iff the old one recognizes its reverse w^{-1}, and so it recognizes exactly the *msbf*-encodings.

Example 15.6 Figure 15.1 shows NFAs for the solutions of $2x_I - y_I \leq -2$ in *lsbf* (left) and *msbf* encoding (right). The NFA on the right is obtained by reversing the transition, and exchanging the initial and final state. Figure 15.2 shows NFAs for the solutions of $2x_I - y_I = -1$, also in *lsbf* and *msbf* encoding. □



-2
-1
0
q_f
$\begin{bmatrix}0\\1\end{bmatrix}$
$\begin{bmatrix}1\\0\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}0\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}$

Figure 15.1: NFAs for the solutions of $2x - y \leq -2$ over $\mathbb{Z}$ with *lbsf* (left) and *msbf* (right) encodings.

15.3.1 A NBA for the Solutions of $a \cdot x_F \leq \beta$

We construct a DBA recognizing the solutions of $a \cdot x_F \leq \beta$. The algorithm is similar to *AFtoNFA* in Section 10.2. The states of the DBA are integers. We choose transitions and final states so that the following property holds:

$$\text{State } q \in \mathbb{Z} \text{ recognizes the encodings of the tuples } c_F \in [0,1]^n \text{ such that } a \cdot c_F \leq q. \qquad (15.6)$$

However, recall that $\alpha^- \leq a \cdot c_F \leq \alpha^+$ for every $c_F \in [0,1]^n$, and therefore:

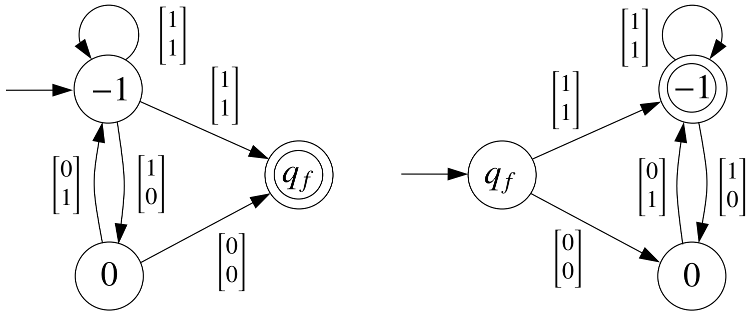

−1
q_f
0
$\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}0\\1\end{bmatrix}$
$\begin{bmatrix}1\\0\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}$
q_f
−1
0
$\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}0\\1\end{bmatrix}$
$\begin{bmatrix}1\\0\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}$

Figure 15.2: NFAs for the solutions of $2x - y = -1$ over $\mathbb{Z}$ with *lbsf* (left) and *msbf* (right) encodings.

- all states $q \geq \alpha^+$ accept all tuples of reals in $[0, 1]^n$, and can be merged with the state α^+;
- all states $q < \alpha^-$ accept no tuples in $[0, 1]^n$, and can be merged with the state $\alpha^- - 1$.

Calling these two merged states *all* and *none*, the possible states of the DBA (not all of them may be reachable from the initial state) are

$$\mathit{all}\ , \ \mathit{none}\ , \quad \text{and} \ \{q \in \mathbb{Z} \mid \alpha^- \leq q \leq \alpha^+ - 1\} \ .$$

All these states but *none* are final, and the initial state is β. Let us now define the set of transitions. Given a state q and a letter $\zeta \in \{0, 1\}^n$, let us determine the target state q' of the unique transition $q \xrightarrow{\zeta} q'$. Clearly, if $q = \mathit{all}$, then $q' = \mathit{all}$, and if $q = \mathit{none}$, then $q' = \mathit{none}$. If $q \in \mathbb{Z}$, we compute the value v that q' must have in order to satisfy property 15.6, and then: If $v \in [\alpha^-, \alpha^+ - 1]$, we set $q' = v$; if $v < \alpha^-$, we set $q' = \mathit{none}$, and if $q > \alpha^+ - 1$, we set $q' = \mathit{all}$. To compute v, recall that a word $w \in (\{0, 1\}^n)^*$ is accepted from q' iff the word ζw is accepted from q. So the tuple $c' \in \mathbb{R}^n$ encoded by w and the tuple $c \in \mathbb{R}^n$ of real numbers encoded by ζw are linked by the equation

$$c = \frac{1}{2}\zeta + \frac{1}{2}c' \tag{15.7}$$

Since c' is accepted from q' iff c is accepted by q, to fulfil property 15.6 we must choose v so that $a \cdot (\frac{1}{2}\zeta + \frac{1}{2}c') \leq q$ holds iff $a \cdot c' \leq v$ holds. We get $v = q - a \cdot \zeta$, and so we define the transition function of the DBA as follows:

$$\delta(q, \zeta) = \begin{cases} \mathit{none} & \text{if } q = \mathit{none} \ \text{ or } \ q - a \cdot \zeta < \alpha^- \\ q - a \cdot \zeta & \text{if } \alpha^- \leq q - a \cdot \zeta \leq \alpha^+ - 1 \\ \mathit{all} & \text{if } q = \mathit{all} \ \text{ or } \ \alpha^+ - 1 < q - a \cdot \zeta \end{cases}$$

Example 15.7 Figure 15.3 shows the DBA for the solutions of $2x_F - y_F \leq 1$ (the state *none* has been omitted). Since $\alpha^+ = 2$ and $\alpha^- = -1$, the possible states of the DBA are *all*, *none*, and

$-1, 0, 1$. The initial state is 1. Let us determine the target state of the transitions leaving state 1. We instantiate the definition of $\delta(q, \zeta)$ with $q = 1$, $\alpha^+ = 2$ and $\alpha^- = -1$, and get

$$\delta(1,\zeta) = \begin{cases} \textit{none} & \text{if } \quad 2\zeta_x + \zeta_y < -2 \\ 1 - 2\zeta_x + \zeta_y & \text{if } \quad -2 \leq -2\zeta_x + \zeta_y \leq 0 \\ \textit{all} & \text{if } \quad 0 < -2\zeta_x + \zeta_y \end{cases}$$

which leads to

$$\delta(1,\zeta) = \begin{cases} 1 & \text{if } \zeta_x = 0 \text{ and } \zeta_y = 0 \\ \textit{all} & \text{if } \zeta_x = 0 \text{ and } \zeta_y = 1 \\ -1 & \text{if } \zeta_x = 1 \text{ and } \zeta_y = 0 \\ 0 & \text{if } \zeta_x = 1 \text{ and } \zeta_y = 1 \end{cases}$$

Recall that, by property 15.6, a state $q \in \mathbb{Z}$ accepts the encodings of the pairs $(x_F, y_F) \in [0,1]^n$ such that $2x_F - y_F \leq q$. This allows us to immediately derive the DBAs for $2x_F - y_F \leq 0$ or $2x_F - y_F \leq -1$: it is the DBA of Figure 15.3, with 0 or -1 as initial state, respectively.

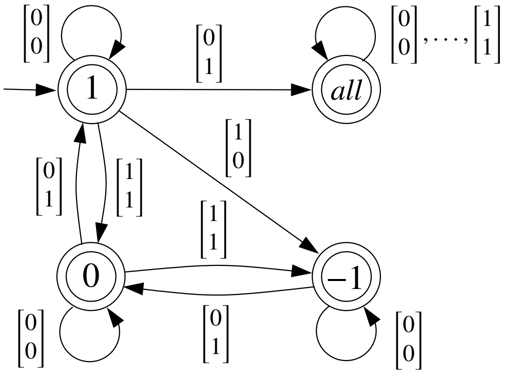


Figure 15.3: DBA for the solutions of $2x - y \leq 1$ over $[0, 1] \times [0, 1]$.

□

Example 15.8 Consider again $\varphi = 2x - y \leq 0$. Recall that $(x, y) \in \mathbb{R}^2$ is a solution of φ iff:

(i) $2x_I - y_I \leq -2$; or

(ii) $2x_I - y_I = -1$ and $2x_F - y_F \leq 1$; or

(iii) $2x_I - y_I = 0$ and $2x_F - y_F \leq 0$.

Figure 15.4 shows at the top a DBA for the x, y satisfying (i). It is easily obtained from the NFA for the solutions of $2x_I - y_I \leq -2$ shown on the right of Figure 15.1.

The DBA at the bottom of Figure 15.4 recognizes the $x, y \in R$ satisfying (ii) or (iii). To construct it, we "concatenate" the DFA on the right of Figure 15.2, and the DBA of Figure 15.3. The DFA recognizes the integer solutions of $2x_I - y_I = -1$, which is adequate for (ii), but changing the final state to 0 we get a DFA for the integer solutions of $2x_I - y_I = 0$, adequate for (iii). Simarly with the DBA, and so it suffices to link state -1 of the DFA to state 1 of the DBA, and state 0 of the DFA to state 0 of the DBA.

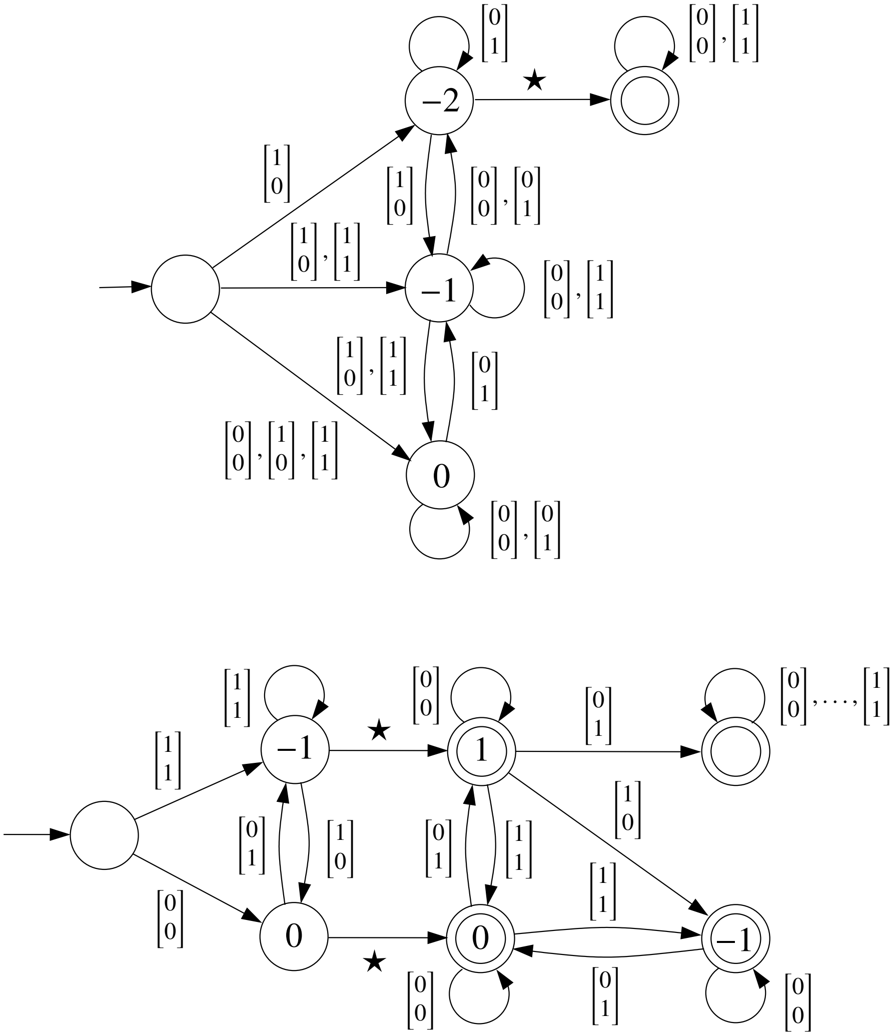

−2
★
−1
0
$\begin{bmatrix}0\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}1\\0\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}0\\1\end{bmatrix}$
$\begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}$
−1
1
0
0
−1
$\begin{bmatrix}1\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}$
$\begin{bmatrix}0\\1\end{bmatrix}$
$\begin{bmatrix}1\\0\end{bmatrix}$
$\begin{bmatrix}0\\0\end{bmatrix}, \ldots, \begin{bmatrix}1\\1\end{bmatrix}$

Figure 15.4: DBA for the real solutions of $2x - y \leq 0$ satisfying (i) (top) and (ii) or (iii) (bottom).

□