CHAPTER 1

Algorithms on Words

1.0. Introduction

This chapter is an introductory chapter to the book. It gives general notions, notation, and technical background. It covers, in a tutorial style, the main notions in use in algorithms on words. In this sense, it is a comprehensive exposition of basic elements concerning algorithms on words, automata and transducers, and probability on words.

The general goal of "stringology" we pursue here is to manipulate strings of symbols, to compare them, to count them, to check some properties, and perform simple transformations in an effective and efficient way.

A typical illustrative example of our approach is the action of circular permutations on words, because several of the aspects we mentioned above are present in this example. First, the operation of circular shift is a transduction which can be realized by a transducer. We include in this chapter a section (Section 1.5) on transducers. Transducers will be used in Chapter 3. The orbits of the transformation induced by the circular permutation are the so-called conjugacy classes. Conjugacy classes are a basic notion in combinatorics on words. The minimal element in a conjugacy class is a good representative of a class. It can be computed by an efficient algorithm (actually in linear time). This is one of the algorithms which appear in Section 1.2. Algorithms for conjugacy are again considered in Chapter 2. These words give rise to Lyndon words which have remarkable combinatorial properties already emphasized in Lothaire (1997). We describe in Section 1.2.5 the Lyndon factorization algorithm.

The family of algorithms on words has features which make it a specific field within algorithmics. Indeed, algorithms on words are often of low complexity but intricate and difficult to prove. Many algorithms have even

Applied Combinatorics on Words, eds. Jean Berstel and Dominique Perrin.
Published by Cambridge University Press.

a linear time complexity corresponding to a single pass scanning of the input word. This contrasts with the fact that correctness proofs of these algorithms are frequently complex. A well-known example of this situation is the Knuth–Morris–Pratt string searching algorithm (see Section 1.2.3). This algorithm is compact, and apparently simple but the correctness proof requires a sophisticated loop invariant.

The field of algorithms on words still has challenging open problems. One of them is the minimal complexity of the computation of a longest common subword of two words which is still unknown. We present in Section 1.2.4 the classic quadratic dynamic programming algorithm. A more efficient algorithm is mentioned in the Notes.

The field of algorithms on words is intimately related to formal models of computation. Among those models, finite automata and context-free grammars are the most used in practice. This is why we devote a section (Section 1.3) to finite automata and another one to grammars and syntax analysis (Section 1.6). These models, and especially finite automata, regular expressions, and transducers, are ubiquitous in the applications. They appear in almost all chapters.

The relationship between words and probability theory is an old one. Indeed, one of the basic aspects of probability and statistics is the study of sequences of events. In the elementary case of a finite sample space, such as in tossing a coin, the sequence of outcomes is a word. More generally, a partition of an arbitrary probability space into a finite number of classes produces sequences over a finite set. Section 1.8 is devoted to an introduction to these aspects. They are developed later in Chapters 6 and 7.

We have chosen to present the algorithms and the related properties in a direct style. This means that there are no formal statements of theorems, and consequently no formal proofs. Nevertheless, we give precise assertions and enough arguments to show the correctness of algorithms and to evaluate their complexity. In some cases, we use results without proof and we give bibliographic indications in the Notes.

For the description of algorithms, we use a kind of programming language that is close to the usual programming languages. Doing this, rather than relying on a precise programming language, gives more flexibility and improves readability.

The syntactic features of our programs concerning the control structure and the elementary instructions, make the language similar to a language such as Pascal. We take some liberty with real programs. In particular, we often omit declarations and initializations of variables. The parameter handling is C–like (no call by reference). In addition to arrays, we also use implicitly data structures such as sets and stacks and pairs or triples of

variables to simplify notation. All functions are global, and there is nothing resembling classes or other features of object-oriented programming. However, we use overloading for parsimony. The functions are referenced in the text and in the index by their name, like LONGESTCOMMONPREFIX() for example.

1.1. Words

We briefly introduce the basic terminology on words. Let $\mathcal{A}$ be a finite set usually called the *alphabet*. In practice, the elements of the alphabet may be characters from some concrete alphabet, but also more complex objects. They may themselves be words on another alphabet, as in the case of syllables in natural language processing (presented in Chapter 3). In information processing, any kind of record can be viewed as a symbol in some huge alphabet. This has the consequence that some apparently elementary operations on symbols, like the test for equality, often need a careful definition and may require a delicate implementation.

We denote as usual by $\mathcal{A}^*$ the set of words over $\mathcal{A}$ and by ε the empty word. For a word w, we denote by $|w|$ the length of w. We use the notation $\mathcal{A}^+ = \mathcal{A}^* - \{\varepsilon\}$. The set $\mathcal{A}^*$ is a monoid. Indeed, the concatenation of words is associative, and the empty word is a neutral element for concatenation. The set $\mathcal{A}^+$ is sometimes called the *free semigroup* over $\mathcal{A}$, while $\mathcal{A}^*$ is called the *free monoid*.

A word w is called a *factor* (resp. a *prefix*, resp. a *suffix*) of a word u if there exist words x, y such that $u = xwy$ (resp. $u = wy$, resp. $u = xw$). The factor (resp. the prefix, resp. the suffix) is *proper* if $xy \neq \varepsilon$ (resp. $y \neq \varepsilon$, resp. $x \neq \varepsilon$). The prefix of length k of a word w is also denoted by $w[0\,..\,k-1]$.

The set of words over a finite alphabet $\mathcal{A}$ can be conveniently seen as a tree. Figure 1.1 represents $\{a, b\}^*$ as a binary tree. The vertices are the elements of $\mathcal{A}^*$. The root is the empty word ε. The sons of a node x are the words xa for $a \in \mathcal{A}$. Every word x can also be viewed as the path leading from the root to the node x. A word x is a prefix of a word y if it is an ancestor in the tree. Given two words x and y, the longest common prefix of x and y is the nearest common ancestor of x and y in the tree.

A word x is a *subword* of a word y if there are words $u_1, \ldots, u_n$ and $v_0, v_1, \ldots, v_n$ such that $x = u_1 \cdots u_n$ and $y = v_0 u_1 v_1 \cdots u_n v_n$. Thus, x is obtained from y by erasing some factors in y.

Given two words x and y, a *longest common subword* is a word z of maximal length that is both a subword of x and y. There may exist several

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

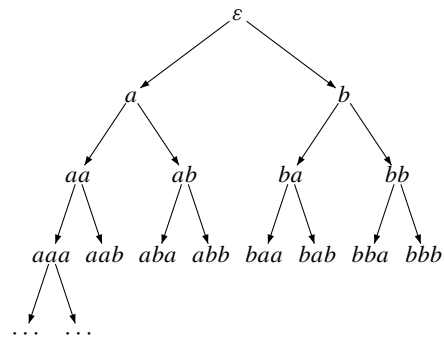


Figure 1.1. The tree of the free monoid on two letters.

longest common subwords for two words x and y. For instance, the words abc and acb have the common subwords ab and ac.

We denote by alph w the set of letters having at least one occurrence in the word w.

The set of factors of a word x is denoted $F(x)$. We denote by $F(\mathcal{X})$ the set of factors of words in a set $\mathcal{X} \subset \mathcal{A}^*$. The *reversal* of a word $w = a_1 a_2 \cdots a_n$, where $a_1, \ldots, a_n$ are letters, is the word $\tilde{w} = a_n a_{n-1} \cdots a_1$. Similarly, for $\mathcal{X} \subset \mathcal{A}^*$, we denote $\widetilde{\mathcal{X}} = \{\tilde{x} \mid x \in \mathcal{X}\}$. A *palindrome word* is a word w such that $w = \tilde{w}$. If $|w|$ is even, then w is a palindrome if and only if $w = x\tilde{x}$ for some word x. Otherwise w is a palindrome if and only if $w = xa\tilde{x}$ for some word x and some letter a.

An integer $p \geq 1$ is a *period* of a word $w = a_1 a_2 \cdots a_n$ where $a_i \in \mathcal{A}$ if $a_i = a_{i+p}$ for $i = 1, \ldots, n - p$. The smallest period of w is called *the* period or the *minimal* period of w.

A word $w \in \mathcal{A}^+$ is *primitive* if $w = u^n$ for $u \in \mathcal{A}^+$ implies $n = 1$.

Two words x, y are *conjugate* if there exist words u, v such that $x = uv$ and $y = vu$. Thus conjugate words are just cyclic shifts of one another. Conjugacy is thus an equivalence relation. The conjugacy class of a word of length n and period p has p elements if p divides n and has n elements otherwise. In particular, a primitive word of length n has n distinct conjugates.

1.1.1. Ordering

There are three order relations frequently used on words. We give the definition of each of them.

The *prefix order* is the partial order defined by $x \leq y$ if x is a prefix of y.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

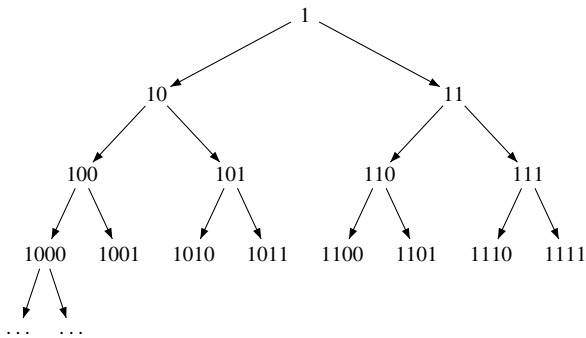


Figure 1.2. The tree of integers in binary notation.

Two other orders, the *radix order* and the *lexicographic order* are refinements of the prefix order which are defined for words over an ordered alphabet $\mathcal{A}$. Both are total orders.

The *radix order* is defined by $x \leq y$ if $|x| < |y|$ or $|x| = |y|$ and $x = uax'$ and $y = uby'$ with a, b letters and $a \leq b$. If integers are represented in base k without leading zeroes, then the radix order on their representations corresponds to the natural ordering of the integers. If we allow leading zeroes, the same holds provided the words have the same length (which can always be achieved by padding).

For $k = 2$, the tree of words without leading zeroes is given in Figure 1.2. The radix order corresponds to the order in which the vertices are met in a breadth-first traversal. The index of a word in the radix order is equal to the number represented by the word in base 2.

The *lexicographic order*, also called *alphabetic order*, is defined as follows. Given two words x, y, we have $x < y$ if x is a proper prefix of y or if there exist factorizations $x = uax'$ and $y = uby'$ with a, b letters and $a < b$. This is the usual order in a dictionary. Note that $x < y$ in the radix order if $|x| < |y|$ or if $|x| = |y|$ and $x < y$ in the lexicographic order.

1.1.2. Distances

A *distance* over a set E is a function d that assigns to each element of E a nonnegative number such that:

(i) $d(u, v) = d(v, u)$,
(ii) $d(u, w) \leq d(u, v) + d(v, w)$ (triangular inequality)
(iii) $d(u, v) = 0$ if and only if $u = v$.

Several distances between words are used (see Figures 1.3 and 1.4). The most common is the *Hamming distance*. It is only defined on words of equal

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

(a) Hamming distance (b) Subword distance

Figure 1.3. The Hamming distance is 3 and the subword distance is 2.

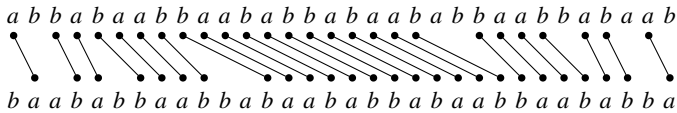


Figure 1.4. The Hamming distance of these two Thue–Morse blocks of length 32 is equal to their length; their subword distance is only 6.

length. For two words $u = a_0 \cdots a_{n-1}$ and $v = b_0 \cdots b_{n-1}$, where a_i, b_i are letters, it is the number $d_H(u, v)$ of indices i with $0 \leq i \leq n-1$ such that $a_i \neq b_i$. In other terms

$$d_H(u, v) = \mathrm{Card}\{i \mid 0 \leq i \leq n-1 \text{ and } a_i \neq b_i\}.$$

Thus the Hamming distance is the number of *mismatches* between u and v. It can be verified that d_H is indeed a distance. Observe that $d_H(u, v) = n - p$ where p is the number of positions at which u and v coincide. In a more general setting, a distance between letters is used instead of just counting each mismatch as 1.

The Hamming distance takes into account the differences at the same position. In this way, it can be used as a measure of modifications or errors caused by a modification of a symbol by another one, but not of a deletion or an insertion. Another distance is the subword distance which is defined as follows. Let u be a word of length n and v be a word of length m, and p be the length of a longest common subword of u and v. The *subword distance* between u and v is defined as $d_S(u, v) = n + m - 2p$. It can be verified that $d_S(u, v)$ is the minimal number of insertions and suppressions that change u into v. The name *indel* (for *in*sertion and *del*etion) is used to qualify a transformation that is either an insertion or a deletion.

A common generalization of the Hamming distance and the subword distance is the *edit distance*. It takes into account the substitutions of a symbol by another in additions to indels (see Problem 1.1.2).

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

A related distance is the *prefix distance*. It is defined as $d(u, v) = n + m - 2p$ where $n = |u|$, $m = |v|$ and p is the length of the longest common prefix of u and v. It can be verified that the prefix distance is actually the length of the shortest path from u to v in the tree of the free monoid.

1.2. Elementary algorithms

In this section, we treat algorithmic problems related to the basic notions on words: prefixes, suffixes, factors.

1.2.1. Prefixes and suffixes

Recall that a word x is a *prefix* of a word y if there is a word u such that $y = xu$. It is said to be *proper* if u is nonempty. Checking whether x is a prefix of y is straightforward. Algorithm LongestCommonPrefix below returns the length of the longest common prefix of two words x and y.

```
LongestCommonPrefix(x, y)
1 ▷ x has length m, y has length n
2 i ← 0
3 while i < m and i < n and x[i] = y[i] do
4       i ← i + 1
5 return i
```

In the tree of a free monoid, the length of the longest common prefix of two words is the height of the least common ancestor.

As mentioned earlier, the conceptual simplicity of the above algorithm hides implementation details such as the computation of equality between letters.

1.2.2. Overlaps and borders

We introduce first the notion of overlap of two words x and y. It captures the amount of possible overlap between the end of x and the beginning of y. To avoid trivial cases, we rule out the case where the overlap would be the whole word x or y. Formally, the *overlap* of x and y is the longest proper suffix of x that is also a proper prefix of y. For example, the overlap of *abacaba* and *acabaca* has length 5. The *border* of a nonempty word w is the overlap of w and itself. Thus it is the longest word u which is both a proper prefix and

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

a proper suffix of w. The overlap of x and y is denoted by overlap(x, y), and the border of x by border(x). Thus border(x) = overlap(x, x).

As we shall see, the computation of the overlap of x and y is intimately related to the computation of the border. This is due to the fact that the overlap of x and y involves the computation of the overlaps of the prefixes of x and y. Actually, one has overlap(xa, y) = border(xa) whenever x is a prefix of y and a is a letter. Next, the following formula allows the computation of the overlap of xa and y, where x, y are words and a is a letter. Let z = overlap(x, y).

$$\text{overlap}(xa, y) = \begin{cases} za & \text{if } za \text{ is a prefix of } y, \\ \text{border}(za) & \text{otherwise.} \end{cases}$$

Observe that border(za) = overlap(za, y) because z is a prefix of y. The computation of the border is an interesting example of a nontrivial algorithm on words. A naive algorithm would check, for each prefix of w, whether it is also a suffix of w, and select the longest such prefix. This would obviously require a time proportional to $|w|^2$. We will see that it can be done in time proportional to the length of the word. This relies on the following recursive formula allowing the computation of the border of xa in terms of the border of x, where x is a word and a is a letter. Let u = border(x) be the border of x. Then for each letter a,

$$\text{border}(xa) = \begin{cases} ua & \text{if } ua \text{ is a prefix of } x, \\ \text{border}(ua) & \text{otherwise.} \end{cases} \tag{1.2.1}$$

The following algorithm (Algorithm BORDER) computes the length of the border of a word x of length m. It outputs an array b of $m + 1$ integers such that $b[j]$ is the length of the border of $x[0\,..\,j-1]$. In particular, the length of border(x) is $b[m]$. It is convenient to set $b[0] = -1$. For example, if $x = abaababa$, the array b is

	0	1	2	3	4	5	6	7	8
b :	−1	0	0	1	1	2	3	2	3

BORDER(x)

```
1 ▷ x has length m, b has size m + 1
2 i ← 0
3 b[0] ← −1
4 for j ← 1 to m − 1 do
5       b[j] ← i
```

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

```
 6        ▷ Here x[0 . . i − 1] = border(x[0 . . j − 1])
 7        while i ≥ 0 and x[j] ≠ x[i] do
 8              i ← b[i]
 9        i ← i + 1
10  b[m] ← i
11  return b
```

This algorithm is an implementation of Formula (1.2.1). Indeed, the body of the loop on j computes, in the variable i, the length of the border of $x[0\,..\,j]$. This value will be assigned to $b[j]$ at the next increase of j. The inner loop is a translation of the recursive formula.

The algorithm computes the border of x (or the table b itself) in time $O(|x|)$. Indeed, the execution time is proportional to the number of comparisons of symbols performed at line 7. Each time a comparison is done, the expression $2j - i$ increases strictly. In fact, either $x[j] = x[i]$ and i, j both increase by 1, or $x[j] \neq x[i]$, and j remains constant while i decreases strictly (since $b[i] < i$). Since the value of the expression is initially 0 and is bounded by $2|x|$, the number of comparisons is at most $2|x|$.

The computation of the overlap of two words x, y will be done in the next section.

1.2.3. Factors

In this section, we consider the problem of checking whether a word x is a factor of a word y. This problem is usually referred to as a *string matching problem*. The word x is called the *pattern* and y is the *text*. A more general problem, referred to as *pattern matching*, occurs when x is replaced by a *regular expression* $\mathcal{X}$ (see Section 1.4). The evaluation of the efficiency of string matching or pattern matching algorithms depends on which parameters are considered. In particular, one may consider the pattern to be fixed (because several occurrences of the same pattern are looked for in an unknown text), or the text to be fixed (because several different patterns will be looked for in this text). When the pattern or the text is fixed, it may be subject to a preprocessing. Moreover, the evaluation of the complexity can take into account either only the computation time, or both time and space. This may make a significant difference on very large texts and patterns.

We begin with a naive quadratic string searching algorithm. To check whether a word x is a factor of a word y, it is clearly enough to test for each index $j = 0, \ldots, n - 1$ if x is a prefix of the word $y[j\,..\,n - 1]$.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

```
NaiveStringMatching(x, y)
1 ▷ x has length m, y has length n
2 (i, j) ← (0, 0)
3 while i < m and j < n do
4       if x[i] = y[j] then
5             (i, j) ← (i + 1, j + 1)
6       else  j ← j − i + 1
7             i ← 0
8 return i = m
```

The number of comparisons required in the worst case is $O(|x||y|)$. The worst case is reached for $x = a^m b$ and $y = a^n$. The number of comparisons performed is in this case $m(n - m - 1)$.

We shall see now that it is possible to search a word x inside another word y in linear time, that is in time $O(|x| + |y|)$. The basic idea is to use a finite automaton recognizing the words ending with x. If we can compute some representation of it in time $O(|x|)$, then it will be straightforward to process the word y in time $O(|y|)$.

The wonderfully simple solution presented below uses the notion of border of a word. Suppose that we are in the process of identifying x inside y, the position i in x being placed in front of position j in y, as in the naive algorithm. We can then set $x = ubt$ where $b = x[i]$ and $y = wuaz$ where $a = y[j]$. If $a = b$, the process goes on with $i + 1$, $j + 1$. Otherwise, instead of just moving x one place to the right (i.e. $j = j - i + 1$, $i = 0$), we can take into account that the next possible position for x is determined by the border of u. Indeed, we must have $y = w'u'az$ and $x = u'ct'$ with u' both a prefix of u and a suffix of u since $w'u' = wu$ (see Figure 1.5). Hence the next comparison to perform is between $y[j]$ and $x[k]$ where $k - 1$ is the length of the border of u.

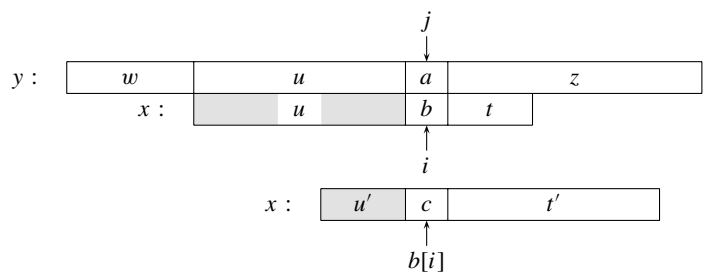


Figure 1.5. Checking $y[j]$ against $x[i]$: if they are different, $y[j]$ is checked against $x[b[i]]$.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

The algorithm is realized by the following program (Algorithm SEARCHFACTOR). It returns the starting position of the first occurrence of the word x inside the word y, and $|y|$ if x is not a factor of y. It uses an array b of $|x|+1$ integers such that $b[i]$ is the length of the border of $x[0\,..\,i-1]$.

```
SEARCHFACTOR(x, y)
 1 ▷ x has length m, y has length n
 2 ▷ b is the array of length of borders of the prefixes of x
 3 b ← BORDER(x)
 4 (i, j) ← (0, 0)
 5 while i < m and j < n do
 6       while i ≥ 0 and x[i] ≠ y[j] do
 7             i ← b[i]
 8       (i, j) ← (i + 1, j + 1)
 9 return i = m
```

The time complexity is $O(|x|+|y|)$. Indeed, the computation of the array b can be done in time $O(|x|)$ as in Section 1.2.2. Further, the analysis of the algorithm given by the function SEARCHFACTOR is the same as that for the function BORDER. The expression $2j-i$ increases strictly at each comparison of two letters, and thus the number of comparisons is bounded by $2|y|$. Thus, the complete time required to check whether x is a factor of y is $O(|x|+|y|)$ as announced.

Computing the overlap of two word x, y can be done as follows. We may suppose $|x| < |y|$. The value of overlap(x, y) is the final value of the variable i in the algorithm SEARCHFACTOR applied to the pair (y, x).

1.2.4. Subwords

We now consider the problem of looking for subwords. The following algorithm checks whether x is a subword of y. In contrast to the case of factors, a greedy algorithm suffices to perform the check in linear time.

```
ISSUBWORD(x, y)
 1 ▷ x has length m, y has length n
 2 (i, j) ← (0, 0)
 3 while i < m and j < n do
 4       if x[i] = y[j] then
 5             i ← i + 1
 6       j ← j + 1
 7 return i = m
```

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

We denote by $\mathrm{lcs}(x, y)$ the set of *longest common subwords* (also called longest common subsequences) of two words x and y. The computation of the longest common subwords is a classical algorithm with many practical uses. We present below a quadratic algorithm. It is based on the following formula.

$$\mathrm{lcs}(xa, yb) = \begin{cases} \mathrm{lcs}(x, y)a & \text{if } a = b, \\ \max(\mathrm{lcs}(xa, y), \mathrm{lcs}(x, yb)) & \text{otherwise.} \end{cases}$$

where max() stands for the union of the sets if their elements have equal length, and for the set with the longer words otherwise.

In practice, one computes the length of the words in $\mathrm{lcs}(x, y)$. For this, define an array $M[i, j]$ by $M[i, j] = k$ if the longest common subwords to the prefixes of length i of x and j of y have length k. The previous formula then translates into

$$M[i+1, j+1] = \begin{cases} M[i, j] + 1 & \text{if } a = b, \\ \max(M[i+1, j], M[i, j+1]) & \text{otherwise.} \end{cases}$$

For instance, if $x = abba$ and $y = abab$, the array M is the following.

		a	b	a	b
	0	0	0	0	0
a	0	1	1	1	1
b	0	1	2	2	2
b	0	1	2	2	3
a	0	1	2	3	3

The first row and the first column of the array M are initialized at 0. The following function computes the array M.

```
LcsLengthArray(x, y)
 1 ▷ x has length m and y has length n
 2 for i ← 0 to m − 1 do
 3       for j ← 0 to n − 1 do
 4             if x[i] = y[j] then
 5                   M[i + 1, j + 1] ← M[i, j] + 1
 6             else  M[i + 1, j + 1] ← max(M[i + 1, j], M[i, j + 1])
 7 return M
```

The above algorithm has quadratic time and space complexity. Observe that the length of the longest common subwords, namely the value $M[m, n]$,

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

can be computed in linear space (but quadratic time) by computing the matrix M row by row or column by column. To recover a word in the set $\mathrm{lcs}(x, y)$, it is enough to walk backwards through the array M.

```
LCS(x, y)
 1 ▷ result is a longest common word w
 2 M ← LCSLENGTHARRAY(x, y)
 3 (i, j, k) ← (m − 1, n − 1, M[m, n] − 1)
 4 while k ≥ 0 do
 5     if x[i] = y[j] then
 6           w[k] ← x[i]
 7           (i, j, k) ← (i − 1, j − 1, k − 1)
 8      else if M[i + 1, j] < M[i, j + 1] then
 9                 i ← i − 1
10           else  j ← j − 1
11      return w
```

1.2.5. Conjugacy and Lyndon words

Two words x, y are said to be *conjugate* if $x = uv$, $y = vu$, for some words u, v. Thus two words are conjugate if they differ only by a cyclic permutation of their letters.

To check whether x and y are conjugate, we can compare all possible cyclic permutations of x with y. This requires $O(|x||y|)$ operations. Actually we can do much better as follows. Indeed, x and y are conjugate if and only if $|x| = |y|$ and if x is a factor of yy. Indeed, if $|x| = |y|$ and $yy = uxv$, we have $|y| \leq |ux|, |xv|$ and thus there are words u', v' such that $x = v'u'$ and $y = uv' = u'v$. Since $|x| = |y|$, we have $|u'| = |u|$, whence $u = u'$ and $v = v'$. This shows that $x = vu$, $y = uv$.

Hence, using the linear time algorithm SEARCHFACTOR of Section 1.2.3, we can check in $O(|x| + |y|)$ whether two words x, y are conjugate.

Recall that a *Lyndon word* is a word which is strictly smaller than any of its conjugates for the alphabetic ordering. In other terms, a word x is a Lyndon word if for any factorization $x = uv$ with u, v nonempty, one has $uv < vu$. A Lyndon word is primitive.

Any primitive word has a conjugate which is a Lyndon word, namely its least conjugate. Computing the smallest conjugate of a word is a practical way to compute a standard representative of the conjugacy class of a word (this is sometimes called *canonization*). This can be done in linear time by the following algorithm, which is a modification of the algorithm BORDER of Section 1.2.2. It is applied to a word x of length m. We actually use an

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

array containing x^2, and call this array x. Of course, an array of length m would suffice provided the indices are computed mod m.

```
CircularMin(x)
 1  (i, j, k) ← (0, 1, 0)
 2  b[0] ← −1
 3  while k + j < 2m do
 4        ▷ Here x[k . . k + i − 1] = border(x[k . . k + j − 1])
 5        if j − i = m then
 6              return k
 7        b[j] ← i
 8        while i ≥ 0 and x[k + j] ≠ x[k + i] do
 9              if x[k + j] < x[k + i] then
10                    (k, j) ← (k + j − i, i)
11              i ← b[i]
12        (i, j) ← (i + 1, j + 1)
```

Algorithm CircularMin looks like Algorithm Border. Indeed, if we discard lines 5–6 and lines 9–10 in algorithm CircularMin, the variable k remains 0 and we obtain an essentially equivalent algorithm (with a **while** loop replacing the **for** loop). The key assertion of this algorithm is that $x[k \,..\, k + i - 1] = \text{border}(x[k \,..\, k + j - 1])$, as indicated at line 4. This is the same as the assertion in Algorithm Border for $k = 0$. The array b contains the information on borders, in the sense that $b[j]$ is the length of $\text{border}(x[k \,..\, k + j - 1])$.

The value of k is the index of the beginning of a candidate for a least conjugate of x (see Figure 1.6). If the condition at line 9 holds, a new candidate has been found. The assignment at line 10 shifts the value of k by $j - i$, and j is adjusted in such a way that the value of $k + j$ is not modified. The modifications of the value of k do not require the entire recomputation of the array b. Indeed, the values $b[j']$ for $0 \le j' < i$ serve both for the old and the new candidate. For the same reason as for Algorithm Border, the time complexity is linear in the size of x.

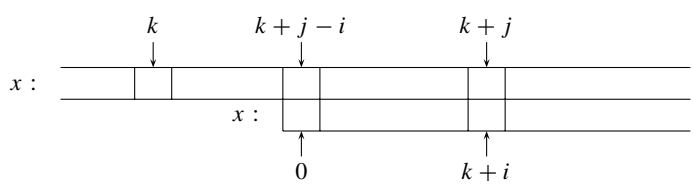


Figure 1.6. Checking whether $x[k \,..\, k + m - 1]$ is the least circular conjugate of x.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

Any word admits a unique factorization as a nonincreasing product of Lyndon words. In other words, for any word x, there is a factorization

$$x = \ell_1^{n_1} \cdots \ell_r^{n_r}$$

where $r \geq 0$, $n_1, \ldots, n_r \geq 1$, and $\ell_1 > \cdots > \ell_r$ are Lyndon words. We discuss now an algorithm to compute this factorization.

The following program computes the pair (ℓ_1, n_1) for x in linear time. By iteration, it allows us to compute the Lyndon factorization in linear time.

```
LyndonFactorization(x)
1 ▷ x has length m
2 (i, j) ← (0, 1)
3 while j < m and x[i] ≤ x[j] do
4       if x[i] < x[j] then
5             i ← 0
6       else  i ← i + 1
7       j ← j + 1
8 return (j − i, ⌊j/(j − i)⌋)
```

The idea of the algorithm is the following. Assume that at some step, we have $x = \ell^n p y$, where ℓ is a Lyndon word, $n \geq 1$ and p is a proper prefix of ℓ. The pair (ℓ, n) is a candidate for the value (ℓ_1, n_1) of the factorization. The relation with the values i, j of the program is given by $j = |\ell^n p|$, $j - i = |\ell|$, $n = \lfloor j/(j-i) \rfloor$. Let $a = x[i]$, $b = x[j]$. Then $\ell = paq$ for some word q, and $x = \ell^n pbz$. If $a < b$, then $\ell' = \ell^n pb$ is a Lyndon word. The pair $(\ell', 1)$ becomes the new candidate. If $a = b$, then pb replaces p. Finally, if $a > b$ the pair (ℓ, n) is the correct value of (ℓ_1, n_1).

The above algorithm can also be used to compute the Lyndon word ℓ in the conjugacy class of a primitive word x. Indeed, ℓ is the only Lyndon word of length $|x|$ that appears in the Lyndon factorization of xx. Thus, Algorithm LyndonFactorization gives an alternative to Algorithm CircularMin.

1.3. Tries and automata

In this section, we consider sets of words. These sets arise in a natural way in applications. Dictionaries in natural language processing, or more generally text processing in databases are typical examples. Another situation is when one considers properties of words, and the sets satisfying such a property, for example the set of all words containing a given pattern. We are interested in the practical representation for retrieval and manipulation of sets of words.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

The simplest case is that of finite, but possibly very large sets. General methods for manipulation of sets may be used. This includes hash functions, bit vectors, and various families of search trees. These general methods are sometimes available in programming packages. We will be interested here in methods that apply specifically to sets of words.

Infinite sets arise naturally in pattern matching. The natural way to handle them is by means of two equivalent notions: regular expressions and finite automata. We describe here in some detail these approaches.

1.3.1. Tries

A *trie* is the simplest nontrivial structure allowing the representation of a finite set $\mathcal{X}$ of words. It has both the advantage of reducing the space required to store the set of words and to allow a fast access to each element.

A trie R is a rooted tree. Each edge is labelled with a letter. The labels have the property that two distinct edges starting in the same vertex have distinct labels. A subset T of the set of vertices is called the set of *terminal vertices*. The set $\mathcal{X}$ of words *represented* by the trie R is the set of labels of paths from the root to a vertex in T. It is convenient to assume that every vertex is on a path from the root to some vertex in T (since otherwise the vertex could be removed). In particular, every leaf of the tree is a terminal vertex.

Example 1.3.1. The trie represented on Figure 1.7 represents the set

$$\mathcal{X} = \{\text{leader, let, letter, sent}\}.$$

The terminal vertices are doubly circled.

To implement a trie, we use a partial function NEXT(p, a) which gives the vertex q such that the edge (p, q) is labelled a. The root of the tree is the value of ROOT().

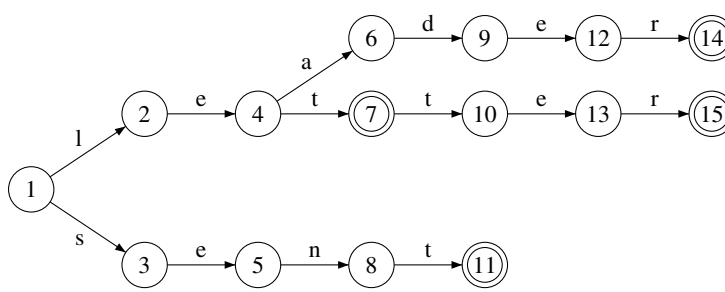


Figure 1.7. A trie.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

```
IsInTrie(w)
1 ▷ checks if the word w of length n is in the trie
2 (i, p) ← LongestPrefixInTrie(w)
3 return i = n and p is a terminal vertex
```

Function IsInTrie returns true if the word w is in the set represented by the trie. It uses the function LongestPrefixInTrie() to compute the pair (i, p) where i is the length of the longest prefix of w which is the label of a path in the trie, and p is the vertex reached by this prefix. For future use, we give a slightly more general version of this function. It computes the pair (i, p) where i is the length of the longest prefix of the suffix of w starting in position j.

```
LongestPrefixInTrie(w, j)
1 ▷ returns the length of the longest prefix of w[j..n − 1]
2 ▷ in the trie, and the vertex reached by this prefix
3 q ← Root()
4 for i ← j to n − 1 do
5       p ← q
6       q ← Next(q, w[i])
7       if q is undefined then
8             return (i − j, p)
9 return (n − j, q)
```

Searching for a word in a trie is done in linear time with respect to the length of the word. It does not depend on the size of the trie. This is the main advantage of this data structure. However, this is only true under the assumption that the function Next can be computed in constant time. In practice, if the alphabet is of large size, this might no longer be true.

To add a word to a trie amounts to the following simple function.

```
AddToTrie(w)
1 ▷ adds the word w to the trie
2 (i, p) ← LongestPrefixInTrie(w, 0)
3 for j ← i to n − 1 do
4       q ← NewVertex()
5       Next(p, w[j]) ← q
6       p ← q
7 Add q to the set of terminal vertices
```

We use a function NewVertex() to create a new vertex of the trie. The function AddToTrie() is linear in the length of w, provided Next() is in constant time.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

To remove a word from a trie is easy if we have a function FATHER() giving the father of a vertex. The function can be tabulated during the construction of the trie (by adding the instruction $\text{FATHER}(q) \leftarrow p$ just after line 5 in Algorithm ADDTOTRIE(). The function FATHER() can also be computed on the fly during the computation of LONGESTPREFIXINTRIE() at line 2 of Algorithm REMOVEFROMTRIE(). Another possibility, avoiding the use of the function FATHER(), is to write the function REMOVEFROMTRIE() recursively. We also use a Boolean function ISLEAF() to test whether a vertex is a leaf or not.

```
REMOVEFROMTRIE(w)
 1 ▷ removes the word w of length n from the trie
 2 (i, p) ← LONGESTPREFIXINTRIE(w, 0)
 3 ▷ i should be equal to n
 4 Remove p from the set of terminal vertices
 5 while ISLEAF(p) and p is not terminal do
 6       (i, p) ← (i − 1, FATHER(p))
 7       NEXT(p, w[i]) ← −1
```

The use of a trie structure reduces the space needed to represent a set of words, compared with a naive representation. If one wishes to further reduce the size, it is possible to use an acyclic graph instead of a tree. The result is an acyclic graph with labelled edges, an initial vertex, and a set of terminal vertices. This is sometimes called a *directed acyclic word graph* abbreviated as DAWG.

Example 1.3.2. We represent below (see Figure 1.8) a DAWG for the set

$$\mathcal{X} = \{\text{leader, let, letter, sent}\}$$

of Example 1.3.1.

For a given finite set $\mathcal{X}$ of words, there is a unique minimal DAWG representing $\mathcal{X}$. This is a particular case of a statement concerning finite

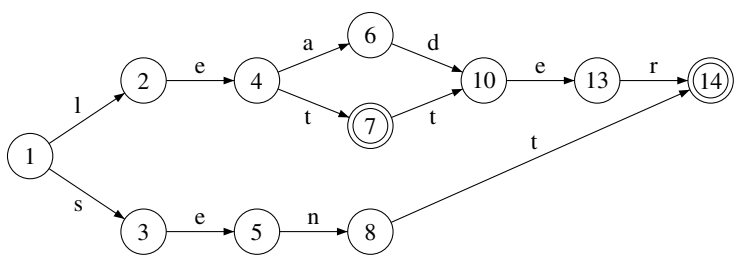


Figure 1.8. A directed acyclic word graph (DAWG).

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

automata that we shall see in the next section. The minimal DAWG is actually the minimal deterministic automaton recognizing $\mathcal{X}$, and standard algorithms exist to compute it.

1.3.2. Automata

An *automaton* over an alphabet $\mathcal{A}$ is composed of a set Q of *states*, a finite set $E \subset Q \times \mathcal{A}^* \times Q$ of *edges* or *transitions* and two sets $I, T \subset Q$ of *initial* and *terminal* states. For an edge $e = (p, w, q)$, the state p is the *origin*, w is the *label*, and q is the *end*.

The automaton is often denoted $\mathfrak{A} = (Q, E, I, T)$, or also (Q, I, T) when E is understood, or even $\mathfrak{A} = (Q, E)$ if $Q = I = T$.

A *path* in the automaton $\mathfrak{A}$ is a sequence

$$(p_0, w_1, p_1), (p_1, w_2, p_2), \ldots, (p_{n-1}, w_n, p_n)$$

of consecutive edges. Its label is the word $x = w_1 w_2 \cdots w_n$. The path *starts* at p_0 and *ends* at p_n. The path is often denoted

$$p_0 \xrightarrow{x} p_n.$$

A path is *successful* if it starts in an initial state and ends in a terminal state. The set *recognized* by the automaton is the set of labels of its successful paths.

A set is *recognizable* or *regular* if it is the set of words recognized by some automaton.

The family of regular sets is both the simplest family of sets that admits an algorithmic description. It is also the most widely used one, because of its numerous closure properties.

A state p is *accessible* if there is a path starting in an initial state and ending in p. It is *coaccessible* if there is a path starting in p and ending in a terminal state. An automaton is *trim* if every state is accessible and coaccessible.

An automaton is *unambiguous* if, for each pair of states p, q, and for each word w, there is at most one path from p to q labelled with w. An automaton is represented as a labelled graph. Initial (final) states are distinguished by an incoming (outgoing) arrow.

Example 1.3.3. The automaton given in Figure 1.9 recognizes the set of words on the alphabet $\{a, b\}$ ending with aba. It is unambiguous and trim.

The definition of an automaton given above is actually an abstraction which went up from circuits and sequential processes. In this context, an automaton is frequently called a *state diagram* to mean that the states represent possible values of time changing variables.

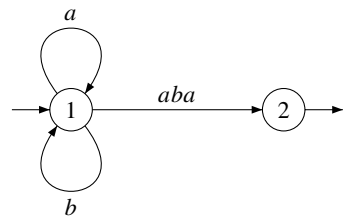


Figure 1.9. A nondeterministic automaton.

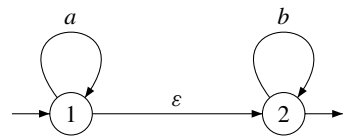


Figure 1.10. A literal automaton for the set a^*b^*.

In some situations, this representation is not adequate. In particular, the number of states can easily become too large. Indeed, the number of states is in general exponential in the number of variables. A typical example is the automaton which memorizes the n last input symbols. It has 2^n states on a binary alphabet but can be represented simply with n binary variables. Observe however that this situation is not general. In particular, automata occurring in linguistics or in bioinformatics cannot in general be represented with such parsimony.

We have introduced here a general model of automata which allows edges labelled by words. This allows in particular edges labelled by the empty word. Such an edge is usually called an *ε-transition*. We will use here two particular cases of this general definition. The first is that of a *synchronous* automaton in which all edges are labelled by letters. In this case, the length of a path equals the length of its label.

An automaton which is not synchronous is called *asynchronous*. Among asynchronous automata, we use *literal* automata as a second class. These have labels that are either letters or the empty word. In this case, the length of a path is always at least equal to the length of its label.

Example 1.3.4. The automaton $\mathfrak{A}$ of Figure 1.10 is asynchronous but literal. It recognizes the set a^*b^*.

An automaton is *deterministic* if it is synchronous, it has a unique initial state, and if, for each state p and each letter a, there is at most one edge which starts at p and is labelled by a. The end state of the edge is denoted by $p \cdot a$. Clearly, a deterministic automaton is unambiguous. For any word w, there is at most one path starting in p and labelled w. The end state of

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

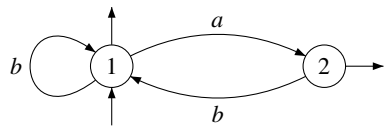


Figure 1.11. The Golden mean automaton.

this is denoted $p \cdot w$. Clearly, for any state p and any words u, v, one has

$$p \cdot uv = (p \cdot u) \cdot v$$

provided the paths exist.

An automaton is *complete* if for any state p and any letter a there exists an edge starting in p and labelled with a. Any automaton can be *completed*, that is transformed into a complete automaton by adding one state (frequently called the sink) and by adding transitions to this state whenever they do not exist in the original automaton.

Example 1.3.5. The automaton given in Figure 1.11 is deterministic. It recognizes the set of words having no occurrence of the factor aa. It is frequently called the *Golden mean* automaton, because the number of words of length n it recognizes is the Fibonacci number F_n (with the convention $F_0 = 0$ and $F_1 = 1$).

An automaton is *finite* if its set of states is finite. Since the alphabet is usually assumed to be finite, this means that the set of edges is finite.

A set of words $\mathcal{X}$ over $\mathcal{A}$ is *recognizable* if it can be recognized by a finite automaton.

The implementation of a deterministic automaton with a finite set of states Q, and over a finite alphabet $\mathcal{A}$, uses the *next-state* function which is the partial function $\text{NEXT}(p, a) = p \cdot a$. In practice, the states are identified with integers, and the next-state function is given either by an array or by a set of edges (a, q) for each state p. The set may be either hashed, or listed, or represented in some balanced tree structure. Other representations exist with the aim of reducing the space while preserving the efficiency of the access.

The next-state function is extended to a function again called NEXT and defined by $\text{NEXT}(p, w) = p \cdot w$, for a word w. A practical implementation has to choose a convenient way to represent the case where the function is undefined.

```
NEXT(p, w)
1 ▷ w has length n
2 for i ← 0 to n − 1 do
3         p ← NEXT(p, w[i])
```

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

```
4       if p is undefined then
5           break
6 return p
```

Example 1.3.6. For the Golden mean automaton, the next-state function is represented by the following table (observe that $2 \cdot a$ is undefined)

	a	b
1	2	1
2		1

For the implementation of nondeterministic automata, we restrict ourselves to the case of a literal automaton which is the most frequent one. For each state, the set of outgoing edges is represented by sets $\text{NEXT}(p, a)$ for each letter a, and $\text{NEXT}(p, \varepsilon)$ for the ε-transitions. By definition $\text{NEXT}(p, a) = \{q \mid (p, a, q) \in E\}$, and $\text{NEXT}(p, \varepsilon) = \{q \mid (p, \varepsilon, q) \in E\}$, where E denotes the set of edges. We denote by INITIAL the set of initial states, and by TERMINAL the set of terminal states.

In order to check whether a word is accepted by a nondeterministic automaton, one performs a search in the graph controlled by the word to be processed. We treat this search in a breadth-first manner in the sense that, for each prefix p of the word, we compute the set of states reachable by p.

For this, we start with the implementation of the next-state function for a set of states. We give a function $\text{NEXT}(S, a)$ that computes the set of states reachable from a state in S by a path consisting of an edge labelled by the letter a followed by a path labelled ε. Another possible option groups the ε-transitions before the edge labelled by a. This will be seen in the treatment of the computation of a word.

```
NEXT(S, a)
1 ▷ S is a set of states, and a is a letter
2 T ← ∅
3 for q ∈ S do
4       T ← T ∪ NEXT(q, a)
5 return CLOSURE(T)
```

The function $\text{CLOSURE}(T)$ computes the set of states accessible from states in T by paths labelled ε. This is just a search in a graph, and it can be performed either depth-first or breadth-first. The time complexity of the function $\text{NEXT}(S, a)$ is $O(d \cdot \text{Card}(S))$, where d is the maximal out-degree of a state.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

The function NEXT() extends to words as follows.

```
NEXT(S, w)
1 ▷ S is a set of states, and w is a word of length n
2 T ← CLOSURE(S)
3 for i ← 0 to n − 1 do
4       T ← NEXT(T, w[i])
5 return T
```

In order to test whether a word w is accepted by an automaton, it suffices to compute the set $S = \text{NEXT}(\text{INITIAL}, w)$, and to check whether S meets the set of final states. This is done by the following function.

```
ISACCEPTED(w)
1 ▷ S is a set of states
2 S ← NEXT(INITIAL, w)
3 return S ∩ TERMINAL ≠ ∅
```

The time complexity of the function ACCEPT(w) is $O(nmd)$, where m is the number of states and d is the maximal out-degree of a state. Thus, in all cases, the time complexity is $O(nm^2)$.

1.3.3. Determinization algorithm

Instead of exploring a nondeterministic automaton, one may compute an equivalent deterministic automaton and perform the acceptance test on the resulting deterministic automaton. This preprocessing is especially interesting when the same automaton is going to be used on several inputs. However, the size of the deterministic automaton may be exponential in the size of the original, nondeterministic one, and the direct search may be the unique realistic option.

We now show how to compute an equivalent deterministic automaton. The states of the deterministic automaton are sets of states, namely the sets computed by the function NEXT(). A practical implementation of the algorithm will use an appropriate data structure for a collection of sets of states. This can be a linked list or an array of sets. We only need to be able to add elements, and to test membership.

The function EXPLORE() essentially searches for the states that are accessible in the automaton $\mathfrak{B}$ under construction. As for any exploration, several strategies are possible. We use a depth-first search realized by recursive calls of the function EXPLORE().

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

```
EXPLORE(𝒯, S, 𝔅)
 1 ▷ 𝒯 is a collection of sets of states of 𝔄
 2 ▷ 𝒯 is also the set of states of 𝔅
 3 ▷ S is an element of 𝒯
 4 for each letter c do
 5       U ← NEXT_𝔄(S, c)
 6       NEXT_𝔅(S, c) ← U
 7       if U ≠ ∅ and U ∉ 𝒯 then
 8             𝒯 ← 𝒯 ∪ U
 9             (𝒯, 𝔅) ← EXPLORE(𝒯, U, 𝔅)
10 return (𝒯, 𝔅)
```

We can now write the determinization algorithm.

```
NFATODFA(𝔄)
 1 ▷ 𝔄 is a nondeterministic automaton
 2 𝔅 ← NEWDFA()
 3 I ← CLOSURE(INITIAL_𝔄)
 4 INITIAL_𝔅 ← I
 5 ▷ 𝒯 is a collection of sets of states of 𝔄
 6 𝒯 ← I
 7 (𝒯, 𝔅) ← EXPLORE(𝒯, I, 𝔅)
 8 TERMINAL_𝔅 ← {U ∈ 𝒯 | U ∩ TERMINAL_𝔄 ≠ ∅}
 9 return 𝔅
```

The result of Algorithm NFATODFA is the deterministic automaton $\mathfrak{B}$. Its set of states is the set $\mathcal{T}$. In practice, it can be represented by a set of integers coding the elements of $\mathcal{T}$, as the collection $\mathcal{T}$ itself is no longer needed. The complexity of Algorithm NFATODFA is proportional to the size of the resulting deterministic automaton times the complexity of testing membership in line 7.

Example 1.3.7. We show in a first example the computation of a deterministic automaton equivalent to a nondeterministic one. We start with the automaton $\mathfrak{A}$ given in Figure 1.12. We have $\text{INITIAL}_{\mathfrak{A}} = \{1, 2\}$ and

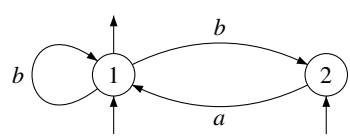


Figure 1.12. The nondeterministic automaton $\mathfrak{A}$.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

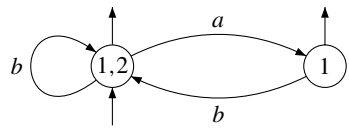


Figure 1.13. The deterministic version $\mathfrak{B}$ of $\mathfrak{A}$.

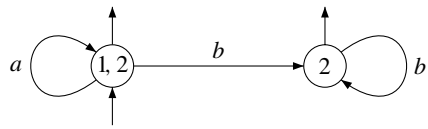


Figure 1.14. A deterministic automaton for the set a^*b^*.

$\text{TERMINAL}_{\mathfrak{A}} = \{1\}$. The next-state function is given by the following table

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

The collection T of sets of states of the resulting automaton computed by Algorithm NFAtoDFA is $T = \{\{1, 2\}, \{1\}\}$. The automaton is represented in Figure 1.13. It is actually the Golden mean automaton of Example 1.3.5.

Example 1.3.8. As a second example, we consider the automaton $\mathfrak{A}$ of Example 1.3.4. We have $\text{INITIAL}_{\mathfrak{A}} = \{1\}$, and $\text{CLOSURE}(\text{INITIAL}_{\mathfrak{A}}) = \{1, 2\}$. The resulting deterministic automaton is given in Figure 1.14

Example 1.3.9. For any set $\mathcal{K}$ of words, let $F(\mathcal{K})$ denote the set of factors of the words in $\mathcal{K}$. We are going to verify a formula involving the shuffle of two sets of words. Formally, the *shuffle operator* ш is defined inductively on words by $u \text{ ш } \varepsilon = \varepsilon \text{ ш } u = u$ and

$$ua \text{ ш } vb = \begin{cases} (u \text{ ш } v)a & \text{if } a = b \\ (ua \text{ ш } v)b \cup (u \text{ ш } vb)a & \text{otherwise.} \end{cases}$$

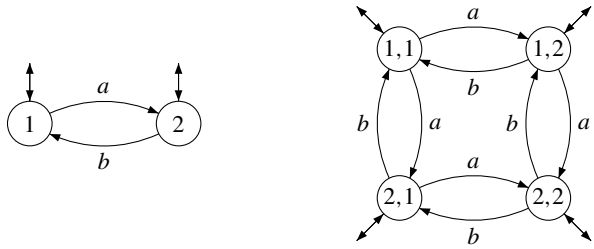
The shuffle of two sets is the union of the shuffles of the words in the sets.

The formula is the following

$$F((ab)^*) \text{ ш } F((ab)^*) = F((ab + ba)^*)\,. \tag{1.3.1}$$

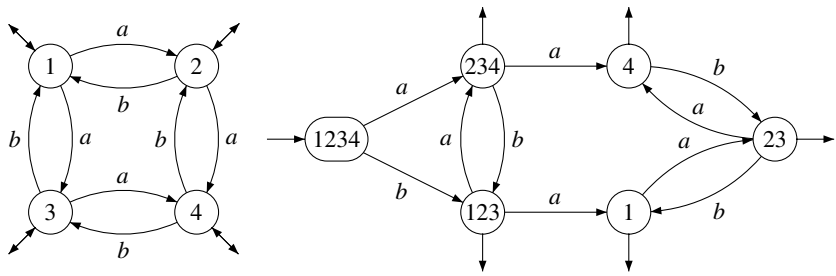
This equality is the basis of a card trick known as Gilbreath's card trick (see Notes).

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press



(a) $F((ab)^*)$ (b) $F((ab)^*) \sqcup\!\sqcup F((ab)^*)$

Figure 1.15. Two automata, recognizing $F((ab)^*)$ and $F((ab)^*) \sqcup\!\sqcup F((ab)^*)$.



(a) After renaming states. (b) Deterministic automaton.

Figure 1.16. On the right, a deterministic automaton recognizing the set $F((ab)^*) \sqcup\!\sqcup F((ab)^*)$ which is recognized by the automaton on the left.

In order to prove this formula, we apply a general principle that is valid for regular sets: to compute deterministic automata for each side of the equation and to check that they are equivalent.

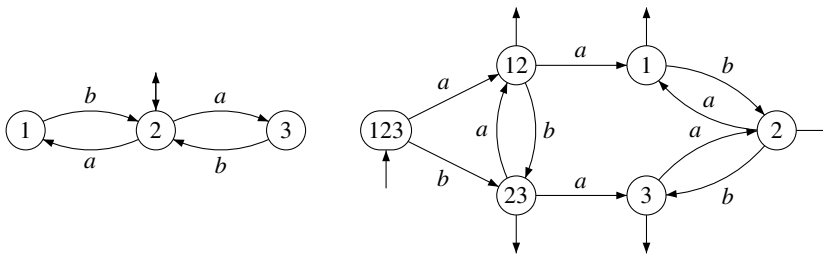
The set $F((ab)^*)$ is recognized by the automaton on the left of Figure 1.15. It is easy to see that the set $F((ab)^*) \sqcup\!\sqcup F((ab)^*)$ is recognized by the nondeterministic automaton on the right of Figure 1.15, realized by forming pairs of states of the first automaton with action on either component.

To compute a deterministic automaton, we first renumber the states as indicated on the left of Figure 1.16. The result of the determinization is shown on the right.

Next, an automaton recognizing $(ab + ba)^*$ is shown on the left of Figure 1.17.

To recognize the set $F((ab + ba)^*)$, we make all states initial and terminal in this automaton. The determinization algorithm is then applied

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press



(a) An automaton recognizing the set $F((ab + ba)^*)$.

(b) A deterministic automaton for this set.

Figure 1.17. Two automata recognizing the set $F((ab + ba)^*)$.

with the new initial state $\{1, 2, 3\}$. The result is shown on the right of Figure 1.17. This automaton is clearly equivalent to the automaton of Figure 1.16. This proves Formula 1.3.1.

1.3.4. Minimization algorithms

A given regular language $\mathcal{S} \subset \mathcal{A}^*$ may be recognized by several different automata. There is however a unique one with a minimal number of states, called the *minimal automaton* of S. We will give a description of the minimal automaton and several algorithms allowing it to be computed.

The abstract definition is quite simple: the states are the nonempty sets of the form $x^{-1}\mathcal{S}$ for $x \in \mathcal{A}^*$ where

$$x^{-1}\mathcal{S} = \{y \in \mathcal{A}^* \mid xy \in \mathcal{S}\}.$$

The initial state is the set $\mathcal{S}$ itself (corresponding to $x = \varepsilon$) and the final states are the sets $x^{-1}\mathcal{S}$ with $x \in \mathcal{S}$ (or, equivalently, such that $\varepsilon \in x^{-1}\mathcal{S}$). There is a transition from the state $x^{-1}\mathcal{S}$ by letter $a \in \mathcal{A}$ to the state $(xa)^{-1}\mathcal{S}$.

Example 1.3.10. Let us consider the set $\mathcal{S}_n$ of words over $\mathcal{A} = \{a, b\}$ that have a symbol a at the $(n + 1)$th position before the end for some $n \geq 0$. Formally, $\mathcal{S}_n = \mathcal{A}^* a \mathcal{A}^n$. For any $x = a_0 a_1 \cdots a_m \in \mathcal{A}^*$, one has

$$x^{-1}\mathcal{S}_n = \mathcal{S}_n U \bigcup_{i \in P(x)} \mathcal{A}^{n-i}$$

where $P(x) = \{i \mid 0 \leq i \leq n \text{ and } a_{m-i} = a\}$. Thus the minimal automaton of $\mathcal{S}_n$ has 2^{n+1} states since its set of states is the set of all subsets of $\{0, 1, \ldots, n\}$. The set $\mathcal{S}$ is also recognized by the nondeterministic automaton of Figure 1.18. This example shows that the size of the minimal automaton can be exponential, compared with the size of a nondeterministic one.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

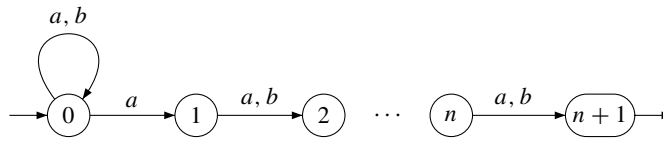


Figure 1.18. Recognizing the words which have the letter a at the $n+1$th position before the end.

A general method for computing the minimal automaton has three steps.

(i) Compute a nondeterministic automaton (e.g., by the method explained in the next section).

(ii) Apply the determinization algorithm of the preceding Section 1.3.3 and remove all states that are not accessible or coaccessible. The resulting automaton is deterministic and trim.

(iii) Apply a minimization algorithm, as described below.

To minimize a deterministic automaton, one uses a sequence of refinements of equivalence relations $\pi_0 \geq \pi_1 \geq \cdots \geq \pi_n$ in such a way that the classes of π_n are the states of the minimal automaton.

The equivalence relation π_n is called the *Nerode equivalence* of the automaton. It is characterized by

$$p \sim q \text{ if and only if } \mathcal{L}_p = \mathcal{L}_q,$$

where $\mathcal{L}_p$ is the set of words recognized by the automaton with initial state p.

The sequence starts with the partition π_0 in two classes separating the terminal states from the other ones. Further, one has $p \equiv q \bmod \pi_{k+1}$ if and only if

$$p \equiv q \bmod \pi_k \quad \text{and} \quad p \cdot a \equiv q \cdot a \bmod \pi_k \text{ for all } a \in A.$$

In the above condition, it is understood that $p \cdot a = \emptyset$ if and only if $q \cdot a = \emptyset$. A partition of a set with n elements can be simply represented by a function assigning to each element x its class $c(x)$.

The computation of the final partition is realized by the following algorithm known as Moore's algorithm.

```
MooreMinimization()
 1  f ← InitialPartition()
 2  do   e ← f
 3       ▷ e is the old partition, f is the new one
 4       f ← Refine(f)
 5  while e ≠ f
 6  return e
```

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

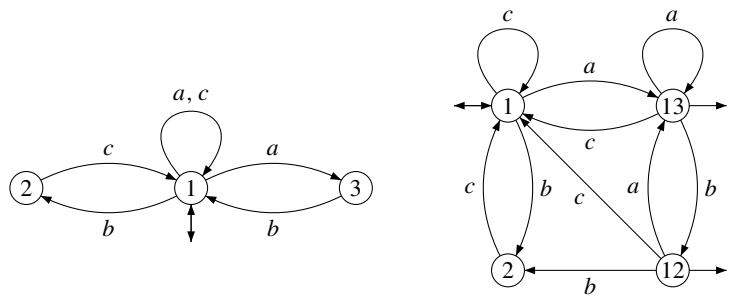
The refinement is realized by the following function in which we denote by $a^{-1}e$ the equivalence $p \equiv q \bmod a^{-1}e$ if and only if $p \cdot a \equiv q \cdot a \bmod e$. Again, it is understood that $p \cdot a$ is defined if and only if $q \cdot a$ is defined.

```
REFINE(e)
1 for a ∈ A do
2       g ← a^{-1}e
3       e ← INTERSECTION(e, g)
4 return e
```

The computation of the intersection of two equivalence relations on an n-element set can be done in time $O(n^2)$ by brute force. A refinement using a radix sort of the pairs of classes improves the running time to $O(n)$. Thus, the function REFINE() runs in time $O(nk)$ on an automaton with n states on an alphabet with k symbols. The loop in the function PARTITION() is executed at most n times since the sequence of successive partitions is strictly decreasing. Moore's algorithm itself thus computes in time $O(n^2k)$ the minimal automaton equivalent to a given automaton with n states and k letters.

Example 1.3.11. Let us consider the set $\mathcal{S} = (a + bc + ab + c)^*$. A nondeterministic automaton recognizing $\mathcal{S}$ is represented on the left of Figure 1.19. The determinization algorithm produces the automaton on the right of the figure.

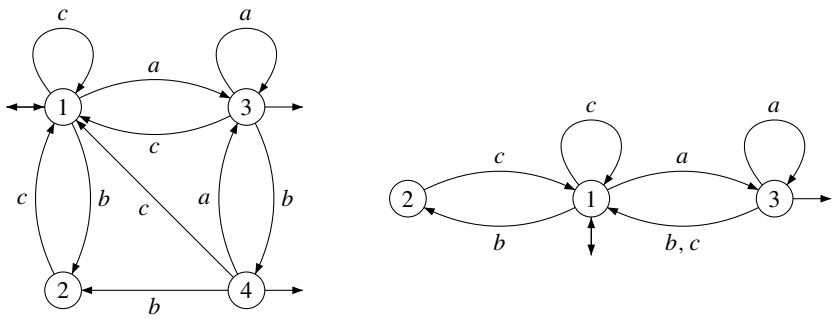
Applying a renumbering of the states, we obtain the automaton on the left of Figure 1.20. The minimization procedure starts with the partition



(a) A nondeterministic automaton. (b) The determinized version.

Figure 1.19. Recognizing the set $(a + bc + ab + c)^*$

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press



(a) Renaming the states. (b) The minimal automaton.

Figure 1.20. The minimization algorithm

$e = \{1, 3, 4\}\{2\}$. Since $a^{-1}e = e$, the action of letter a does not refine e. On the contrary, $b^{-1}e = \{1, 4\}\{2\}\{3\}$ and thus e is refined to $f = \{1, 4\}\{2\}\{3\}$ which is found to be stable. Thus we obtain the minimal automaton represented on Figure 1.20 on the right.

There is a more complicated but more efficient algorithm, known as Hopcroft's algorithm, which can be used to minimize deterministic automata. We assume that the automaton is complete.

The idea is to replace the global operation of intersection of two partitions by the refinement of a partition by a single block. Let P be a set of states, and let a be a letter. Let $a^{-1}P = \{q \mid q \cdot a \in P\}$. A set B of states is *refined* into B' and B'' by the pair (P, a) if the sets $B' = B \cap a^{-1}P$ and $B'' = B \setminus B'$ are both nonempty. Otherwise, B is said to be *stable* by the pair (P, a).

The algorithm starts with the partition composed of the set T of terminal states and its complement T^c. It maintains a set S of pairs (P, a) formed of a set of states and a letter.

The main loop selects a pair (P, a) from the set S. Then for each block B of the current partition which is refined by (P, a) into B', B'', one performs the following steps

1. replace B by B' and B'' in the current partition,
2. for each letter b,
   (a) if (B, b) is in S, then replace (B, b) by (B', b) and (B'', b) in S,
   (b) otherwise add to S the pair (C, b) where C is the smaller of the sets B' and B''.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

If, instead of choosing the smaller of the sets B' and B'', one adds both sets (B', b) and (B'', b) to S, the algorithm becomes a complicated version of Moore's algorithm. The reason why one may dispense with one of the two sets is that when a block B is stable by (P, a) and when P is partitioned into P' and P'', then the refinement of B by (P', a) is the same as the refinement by (P'', a). The choice of the smaller one is the essential ingredient to the improvement of the time complexity from $O(n^2)$ to $O(n \log n)$.

This is described in the following algorithm.

```
HopcroftMinimization()
 1  e ← {T, T^c}
 2  C ← the smaller of T and T^c
 3  for a ∈ A do
 4        Add((C, a), S)
 5  while S ≠ ∅ do
 6        (P, a) ← First(S)
 7        for B ∈ e such that B is refined by (P, a) do
 8              B', B'' ← Refine(B, P, a)
 9              BreakBlock(B, B', B'', e)
10              ▷ breaks B into B', B'' in the partition e
11              Update(S, B, B', B'')
```

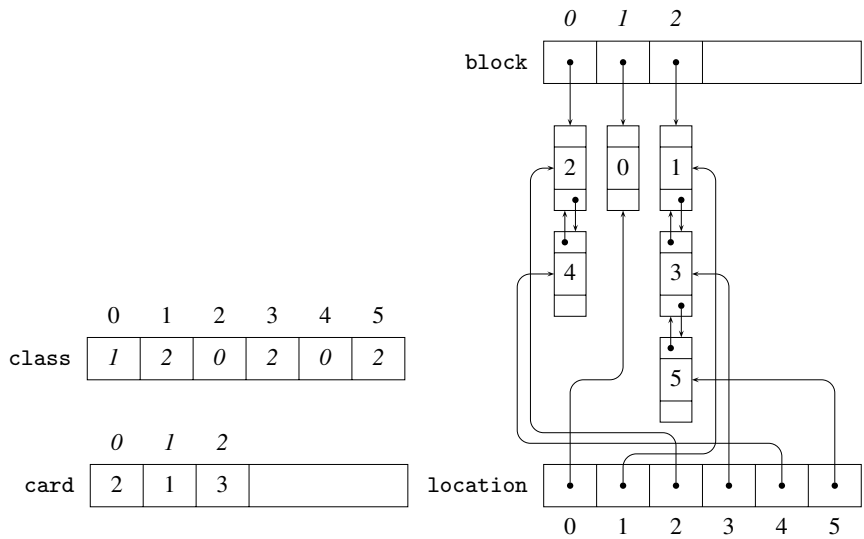
where Update() is the function that updates the set of pairs used to refine the partition, defined as follows.

```
Update(S, B, B', B'')
1  C ← the smaller of B' and B''
2  for b ∈ A do
3        if (B, b) ∈ S then
4              Replace((B, b), S, (B', b), (B'', b))
5        else  Add((C, b), S)
```

A careful implementation of the algorithm leads to a time complexity in $O(kn \log n)$ on an automaton with n states over k letters. One of the key points is the implementation of the function BreakBlock(B, B', B'', e) which has to be implemented so as to run in time $O(\text{Card}(B))$. The function actually replaces B by B'' and adds a new block B'. For this, one traverses B (in linear time) and removes each element which is in B' from B in constant time and adds it to the new block, also in constant time.

The states of a class are represented by a doubly linked list, one list for each class of the partition (see Figure 1.21). This representation allows the

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press



(a) The classes and their size.

(b) The blocks of the partition.

Figure 1.21. A partition of $Q = \{0, \ldots, 5\}$. The class of a state is the integer in the array `class`. The size of a class is given in the array `card`. The elements of a block are chained in a doubly-linked list pointed to by the entry in the array `block`. Each cell in these lists can be retrieved in constant time by its state using the pointer in the array `location`.

element to be removed from the list, and so also from the class, in constant time. An array of pointers indexed by the states allows the retrieval of the location of a state in its block in the partition.

In order to be able to check whether a block B is refined by a pair (P, a), one maintains an array that counts, for each block B, the number of states of $a^{-1}P$ that are found to be in B. The test of whether B is actually refined checks whether this number is both nonzero and strictly less than Card B. This requires that a table be maintained containing the number of elements of the blocks in the current partition.

To summarize, an arbitrary deterministic finite automaton with n states can be minimized in time $O(n \log n)$.

A trim automaton recognizing a *finite set* of words can be minimized in linear time with respect to the size of the automaton. Let $\mathfrak{A}$ be a finite automaton with set of states Q recognizing a finite set of words. Since the automaton is trim, it is acyclic. Thus we are faced again with DAWGs already seen in Section 1.3.1.

The *height* $h(q)$ of a state q is the length of the longest path in $\mathfrak{A}$ starting in q. Equivalently, it is the length of a longest word in the language $\mathcal{L}_q$ of

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

words recognized by the automaton with initial state q. Of course, for any edge (p, a, q) one has $h(p) > h(q)$. Since the automaton is trim, its initial state is the unique state of maximal height. The heights satisfy the formula

$$h(p) = \begin{cases} 0 & \text{if } p \text{ has no outgoing edge,} \\ 1 + \max_{(p,a,q)} h(q) & \text{otherwise.} \end{cases}$$

In the second case, the maximum is taken over all edges starting in p. Observe that this formula leads to an effective algorithm for computing heights because the automaton has no cycle.

The parameters in the algorithm are the number n of states of $\mathfrak{A}$, the number m of transitions, and the size k of the underlying alphabet. Of course, $m \leq n \cdot k$. In practical situations like large dictionaries, the number m is much smaller than the product. As we will see, the minimization algorithm can be implemented in time $O(n + m + k)$.

A word about the representation of $\mathfrak{A}$. Since there are few edges, a convenient representation is to have, for each state p, a list of outgoing edges, each represented by the pair (a, q) such that (p, a, q) is a transition. States are numbered, so traversal, marking, copying, and sorting are done by integers. Also, terminal states are represented in such a way that one knows in constant time whether a state is terminal.

It is easily seen that two states q and q' can be merged into a single state in the minimal automaton only if they have the same height. Therefore, the Nerode equivalence is a refinement of the partition into states of equal height.

Recall that the Nerode equivalence is defined by

$$p \sim q \text{ if and only if } \mathcal{L}_p = \mathcal{L}_q.$$

Recall also that

$$p \sim q \text{ if and only if } (p \in T \Leftrightarrow q \in T) \text{ and } p \cdot a \sim q \cdot a \text{ for all } a \in \mathcal{A} \tag{1.3.2}$$

This formula shows that if the equivalence is known for all states up to some height $h - 1$, it can be computed, by this formula, for states of height h. To describe this in more detail, we associate, to each state q, a sequence of data called its *signature*. It has the form

$$\sigma(q) = (s, a_1, \nu(q_1), a_2, \nu(q_2), \ldots, a_r, \nu(q_r))$$

where $s = 0$ if q is a nonterminal state and $s = 1$ if q is a terminal state, where $(q, a_1, q_1), \ldots, (q, a_r, q_r)$ are the edges starting in q, and where $\nu(p)$ is the class of the state p. We consider that classes of states are represented by integers. We assume moreover that $a_1, \ldots, a_r$ are in increasing order. This can be realized by a bucket sort in time $O(n + m + k)$.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

Then Equation 1.3.2 means that

$$p \sim q \text{ if and only if } \sigma(p) = \sigma(q)\,.$$

Thus, a signature is a sequence of integers of length at most $1 + 2k$, $(k = \operatorname{Card} \mathcal{A})$ and each element in this sequence has a value bounded by $\max(2, k, n)$. Observe that the sum of the lengths of all signatures is bounded by $2m + n$, where m is the number of transitions. In fact, the signature of state p is merely a representation of the transitions in the minimal automaton starting in the state $\nu(p)$.

For computing the Nerode equivalence of the set Q_h of states of height h, one computes the set of signatures of states in Q_h. This set is sorted by a radix sort according to their signatures, viewed as vectors over integers. Then states with equal signatures are consecutive in the sorted list and the test $\sigma(p) = \sigma(q)$ for equivalence can be done in linear time.

Here is the algorithm

```
AcyclicMinimization()
 1 ▷ ν[p] is the state corresponding to p in the minimal automaton
 2 (Q_0, ..., Q_H) ← PartitionByHeight(Q)
 3 for p in Q_0 do
 4       ν[p] ← 0
 5 k ← 0
 6 for h ← 1 to H do
 7       S ← Signatures(Q_h, ν)
 8       P ← RadixSort(Q_h, S)  ▷ P is the sorted sequence Q_h
 9       p ← first state in P
10       ν[p] ← k
11       k ← k + 1
12       for each q in P \ p in increasing order do
13             if σ(q) = σ(p) then
14                   ν[q] ← ν[p]
15             else  ν[q] ← k
16                   (k, p) ← (k + 1, q)
17 return ν
```

A usual topological sort can implement PartitionByHeight(Q) in time $O(n + m)$.

Each signature is then computed in time proportional to its size, so the whole set of signatures is computed in time $O(n + m)$. Each radix sort can be done in time proportional to the sum of the sizes of the signatures, with an overhead of one $O(k)$ initialization of the buckets. So the total time for the sort is also $O(n + m + k)$.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

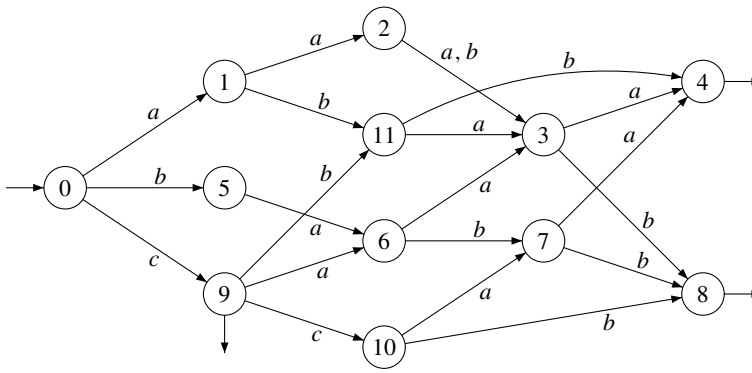


Figure 1.22. A trim automaton recognizing a finite set.

Observe that the test at line 13 is linear in the length of the signatures, so the whole algorithm is in time $O(k+n+m)$.

Example 1.3.12. Consider the automaton of Figure 1.22. The computation of the heights gives the follow partition:

$$Q_0 = \{4, 8\},\ Q_1 = \{3, 7\},\ Q_2 = \{2, 6, 10, 11\},\ Q_3 = \{1, 5, 9\},\ Q_4 = \{0\}.$$

States of height 0 are always final states, and are merged into a class numbered *0*.

	0	1	2	3	4	5	6	7	8	9	10	11
v				*1*	*0*			*1*	*0*			

3 : 0*a*0*b*0
7 : 0*a*0*b*0

(a) Signatures of states of height 1.

(b) The corresponding states of the minimal automaton.

The states of height 1 have the signatures given above. Observe that in a signature, the next state appearing in an edge is replaced by its class. This can be done because the algorithm works by increasing height. These states are merged into a class numbered *1*.

The radix sort of the four states of height 2 gives the sequence $(10, 11, 2, 6)$, so $10, 11$ are grouped into a class *2* and $2, 6$ are grouped into a class *3*.

2 : 0*a*1*b*1
6 : 0*a*1*b*1
10 : 0*a*1*b*0
11 : 0*a*1*b*0

	0	1	2	3	4	5	6	7	8	9	10	11
v			*3*	*1*	*0*		*3*	*1*	*0*		*2*	*2*

(c) Signatures of states of height 2.

(d) The corresponding states of the minimal automaton.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

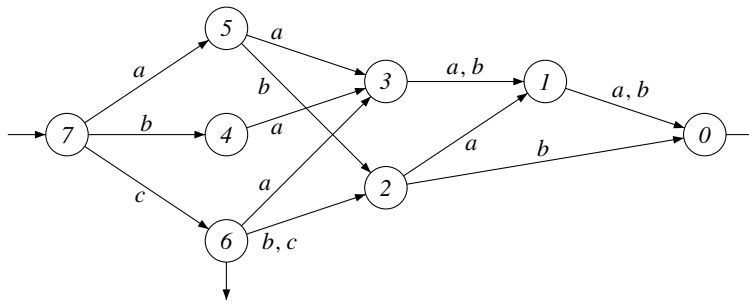


Figure 1.23. The corresponding minimal automaton.

The states of height 3 all give singleton classes, because the signatures are different. This is already clear because they have distinct lengths. In other terms, a refinement of the algorithm could partition the states of the same height into subclasses according to their *width*, that is the number of edges starting in each state.

Thus, the minimal automaton has 8 states. It is given in Figure 1.23.

1 : 0*a*3*b*2
5 : 0*a*3
9 : 1*a*3*b*2*c*2

(a) Signatures of states of height 3

	0	1	2	3	4	5	6	7	8	9	10	11
v	*7*	*5*	*3*	*1*	*0*	*4*	*3*	*1*	*0*	*6*	*2*	*2*

(b) The final state vector of the minimal automaton.

1.4. Pattern matching

The specification of simple patterns on words uses the notion of a *regular expression*. It is an expression built using letters and a symbol representing the empty word, and three operators:

- *union*, denoted by the symbol '+',
- *product*, denoted by mere concatenation,
- *star* denoted by '*'.

These operators are used to denote the usual operations on sets of words. The operations are the set union, set product

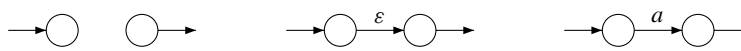
$$\mathcal{X}\mathcal{Y} = \{xy \mid x \in \mathcal{X}, y \in \mathcal{Y}\}$$

and the star operation

$$\mathcal{X}^* = \{x_1 \cdots x_n \mid n \geq 0, x_1, \ldots, x_n \in \mathcal{X}\}\,.$$

A regular expression defines a set of words $W(e)$, by using recursively the

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press



(a) Empty set. (b) Empty word. (c) Letter a.

Figure 1.24. Automata for the empty set, for the empty word, and for a letter.

operations of union, product, and star.

$$W(e+f) = W(e) \cup W(f), \quad W(ef) = W(e)W(f), \quad W(e^*) = W(e)^*.$$

Words in $W(e)$ are said to *match* the expression e. The problem of checking whether a word matches a regular expression is called a *pattern matching problem*.

For instance, $e = (a+b)^*abaab(a+b)^*$ is a regular expression. The set $W(e)$ is the set of words on $\mathcal{A} = \{a, b\}$ having $abaab$ as a factor. More generally, for any word w, the words matching the regular expression $\mathcal{A}^* w \mathcal{A}^*$ are those having w as a factor. Thus, the problem of checking whether a word is a factor of another is a particular case of a pattern matching problem. The same holds for subwords.

For each regular expression e, there exists a finite automaton recognizing the set of words $W(e)$. In other terms, $W(e)$ is a regular set. A proof of this assertion uses an algorithm for building such a finite automaton, inductively on the structure of the expression. Several constructions exist that use slightly different normalizations of automata or of expressions. The main variations concern the use of ε-transitions. We now present a construction which makes extensive use of ε-transitions. The main advantage is its simplicity, and the small size of the resulting automaton.

One starts with simple automata recognizing respectively ε and a, for any letter a. They are represented in Figure 1.24. One further uses a recursive construction on automata with three constructs implementing union, product, and star. The construction is indicated in Figure 1.25. It constructs finite automata with several particular properties. First, each state has at most two edges leaving it. If there are two edges, they have each an empty label. Also, there is a unique initial state i and a unique terminal state t. Finally, there is no edge entering i and no edge leaving t. We call such an automaton a *pattern matching automaton*.

We use a specific representation of nondeterministic automata tailored to the particular automata constructed by the algorithm. A conversion to the representation described above is straightforward. First, an automaton $\mathfrak{A}$ has a state INITIAL (the initial state) and a state TERMINAL (the terminal state). Then, there are two functions NEXT1() and NEXT2(). For each state p, NEXT1$(p) = (a, q)$ if there is an edge (p, a, q). If there is an edge

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

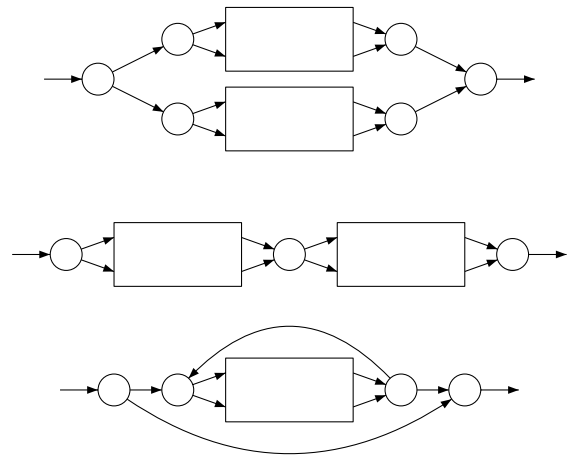


Figure 1.25. Automata for union, product, and star.

(p, ε, q), then $\text{NEXT1}(p) = (\varepsilon, q)$. If there is a second edge (p, ε, q'), then $\text{NEXT2}(p) = (\varepsilon, q')$. If no edge starts from p, then $\text{NEXT}(p)$ is undefined.

We use a function NEWAUTOMATON() to create an automaton with just one initial state and one terminal state and no edges. The function creating an automaton recognizing a is given in Algorithm AUTOMATONLETTER.

```
AUTOMATONLETTER(a)
1 𝔄 ← NEWAUTOMATON()
2 NEXT1(INITIAL_𝔄) ← (a, TERMINAL_𝔄)
3 return 𝔄
```

The automata recognizing the union, the product, and the star are depicted in Figure 1.25. Boxes represent automata, up to their initial and terminal state, which are drawn separately. All drawn edges are ε-transitions. The implementation of the corresponding three functions AUTOMATAUNION(), AUTOMATAPRODUCT(), and AUTOMATONSTAR() is straightforward.

```
AUTOMATAUNION(𝔄, 𝔅)
1 ℭ ← NEWAUTOMATON()
2 NEXT1(INITIAL_ℭ) ← (ε, INITIAL_𝔄)
3 NEXT2(INITIAL_ℭ) ← (ε, INITIAL_𝔅)
4 NEXT1(TERMINAL_𝔄) ← (ε, TERMINAL_ℭ)
5 NEXT1(TERMINAL_𝔅) ← (ε, TERMINAL_ℭ)
6 return ℭ
```

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

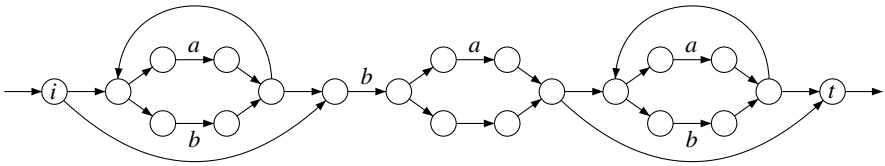


Figure 1.26. The automaton for the expression $(a + b)^*b(a + 1)(a + b)^*$.

The function AUTOMATAPRODUCT() uses a function MERGE() that merges two states into a single one.

```
AUTOMATAPRODUCT(𝔄, 𝔅)
1 ℭ ← NEWAUTOMATON()
2 INITIAL_ℭ ← INITIAL_𝔄
3 TERMINAL_ℭ ← TERMINAL_𝔅
4 MERGE(TERMINAL_𝔄, INITIAL_𝔅)
5 return ℭ
```

```
AUTOMATONSTAR(𝔄)
1 𝔅 ← NEWAUTOMATON()
2 NEXT1(INITIAL_𝔅) ← (ε, INITIAL_𝔄)
3 NEXT2(INITIAL_𝔅) ← (ε, TERMINAL_𝔅)
4 NEXT1(TERMINAL_𝔄) ← (ε, INITIAL_𝔄)
5 NEXT1(TERMINAL_𝔄) ← (ε, TERMINAL_𝔅)
6 return ℭ
```

The practical implementation of these algorithms on a regular expression is postponed to the next section. As an example, consider the automaton in Figure 1.26. It has 21 states and 27 edges. The size of the pattern matching automaton recognizing the set of words matching a regular expression is linear in the size of the expression. Indeed, denote by $n(e)$ the number of states of the pattern matching automaton corresponding to the expression e. Then

$$\begin{aligned} n(a) &= 2 \quad \text{for each letter } a \\ n(\varepsilon) &= 2 \\ n(e + f) &= n(e) + n(f) + 2 \\ n(ef) &= n(e) + n(f) - 1 \\ n(e^*) &= n(e) + 2 \end{aligned}$$

Thus $n(e) \leq 2|e|$, where $|e|$ is the length of the expression e (discarding the left and right parentheses). The number of edges is at most twice the number of states. Thus the space complexity of the resulting algorithm is linear in the size of the expression.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

To realize the run of such an automaton on a word w, one uses Algorithm IsAccepted. We observe that in a pattern matching automaton, the out-degree of a state is at most 2. Therefore, the time complexity of a call IsAccepted(w) is $O(nm)$, where n is the size of the regular expression and $m = |w|$.

In some particular cases, the quadratic complexity $O(nm)$ can be replaced by $O(n + m)$. This is the case in particular for the string matching problem treated in Algorithm SearchFactor.

1.5. Transducers

Beyond formal languages, *relations* between words are a very natural concept. We consider relations over words, but most of the general notions work for relations over arbitrary sets.

Formally, a relation ρ between words over the alphabet $\mathcal{A}$ and words over the alphabet $\mathcal{B}$ is just a subset of the Cartesian product $\mathcal{A}^* \times B^*$. We call it a relation from $\mathcal{A}^*$ to $\mathcal{B}^*$. Actually, such a relation can be viewed as a partial function f_ρ from $\mathcal{A}^*$ to the set $\mathfrak{P}(\mathcal{B}^*)$ of subsets of $\mathcal{B}^*$ defined by

$$f_\rho(x) = \{y \in \mathcal{B}^* \mid (x, y) \in \rho\}, \qquad x \in \mathcal{A}^* .$$

The *inverse* of a relation σ from $\mathcal{A}^*$ to $\mathcal{B}^*$ is the relation σ^{-1} from $\mathcal{B}^*$ to $\mathcal{A}^*$ defined by

$$\sigma^{-1} = \{(v, u) \mid (u, v) \in \sigma\}.$$

The *composition* of a relation σ from $\mathcal{A}^*$ to $\mathcal{B}^*$ and a relation τ from $\mathcal{B}^*$ to $\mathcal{C}^*$ is the relation from $\mathcal{A}^*$ to $\mathcal{C}^*$ defined by $(x, z) \in \sigma \circ \tau$ if and only if there exists $y \in \mathcal{B}^*$ such that $(x, y) \in \sigma$ and $(y, z) \in \tau$. The reader should be aware that the composition of relations goes the other way round compared to the usual composition of functions. The function $f_{\sigma\circ\tau}$ defined by the relation $\sigma \circ \tau$ is $f_{\sigma\circ\tau}(x) = f_\tau(f_\sigma(x))$. One can overcome this unpleasant aspect by writing the function symbol on the right of the argument.

A particular case of a relation ρ from $\mathcal{A}^*$ to $\mathcal{B}^*$ is that of a partial function from $\mathcal{A}^*$ to $\mathcal{B}^*$. In this case, f_ρ is a (partial) function from $\mathcal{A}^*$ into $\mathcal{B}^*$.

Example 1.5.1. Consider the relation $\gamma \subset \mathcal{A}^* \times \mathcal{A}^*$ defined by $(x, y) \in \gamma$ if and only if x and y are conjugate. Clearly, $\gamma = \gamma^{-1}$. The image of a word x is the set of conjugates of x.

Example 1.5.2. Consider the relation $\mu \subset \mathcal{A}^* \times \mathcal{A}^*$ defined by

$$\mu = \{(a_1 a_2 \cdots a_n, a_n a_{n-1} \cdots a_1) \mid a_1, \ldots, a_n \in \mathcal{A}\} .$$

Clearly, $\mu = \mu^{-1}$ and $\mu \circ \mu$ is the identity relation.

Example 1.5.3. For the relation $\rho \subset \mathcal{A}^* \times \mathcal{A}^*$ defined by doubling each letter:

$$\rho = \{(a_1 a_2 \cdots a_n, a_1^2 a_2^2 \cdots a_n^2) \mid a_1, \ldots, a_n \in \mathcal{A}\}$$

the image of a word $x = a_1 a_2 \cdots a_n$ is $a_1^2 a_2^2 \cdots a_n^2$. The inverse is only defined on words of the form $a_1^2 a_2^2 \cdots a_n^2$.

The set of relations on words is subject to several additional operations. The *union* of two relations $\rho, \sigma \subset \mathcal{A}^* \times \mathcal{B}^*$ is the set union $\rho \cup \sigma$. The *product* of ρ and $\sigma \subset \mathcal{A}^* \times \mathcal{B}^*$ is the relation

$$\rho\sigma = \{(ur, vs) \mid (u, v) \in \rho, (r, s) \in \sigma\}.$$

The *star* of $\sigma \subset \mathcal{A}^* \times \mathcal{B}^*$ is the relation

$$\sigma^* = \{(u_1 u_2 \cdots u_n, v_1 v_2 \cdots v_n) \mid (u_i, v_i) \in \sigma, n \geq 0\}.$$

A relation from $\mathcal{A}^*$ to $\mathcal{B}^*$ is *rational* if it can be obtained from subsets of $(\mathcal{A} \cup \{\varepsilon\}) \times (\mathcal{B} \cup \{\varepsilon\})$ by a finite number of operations of union, product, and star.

A rational relation that is a (partial) function is called a *rational function*.

Example 1.5.4. The doubling relation is rational since it can be written, e.g., on the alphabet $\{a, b\}$ as $((a, aa) \cup (b, bb))^*$. More generally, for any morphism f from $\mathcal{A}^*$ to $\mathcal{B}^*$, the relation $\rho = \{(x, f(x) \mid x \in \mathcal{A}^*\}$ is rational. Indeed, $\rho = (\cup_{a \in \mathcal{A}}(a, f(a)))^*$. Thus morphisms are rational functions.

In the same way that regular expressions correspond to automata, rational relations correspond to a kind of automata called *transducers* which are just automata with output. Formally, a *transducer* over the alphabets $\mathcal{A}$, $\mathcal{B}$ is an automaton in which the edges are elements of $Q \times \mathcal{A}^* \times \mathcal{B}^* \times Q$. Thus each edge (p, u, v, q) has an *input label* u which is a word over the alphabet $\mathcal{A}$ and an *output label* v which is a word over the output alphabet $\mathcal{B}$. The transducer is denoted (Q, E, I, T) where Q is the set of states, E the set of edges, I the set of initial states, and T the set of final states.

There are two "ordinary" automata corresponding to a given transducer. The *input automaton* is obtained by using only the input label of each edge. The *output automaton* is obtained by using only the output labels.

The terminology introduced for automata extends naturally to transducers. In particular, a path is labelled by a pair (x, y) formed of its input label x and its output label y. Such a path from p to q is often denoted $p \xrightarrow{x|y} q$. Just as a finite automaton recognizes a set of words, a transducer recognizes or *realizes* a relation. The algorithms of Section 1.4 can easily be adapted to build a transducer corresponding to a given rational relation.

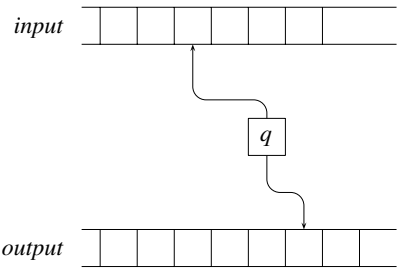


Figure 1.27. A transducer reads the input and writes the output.

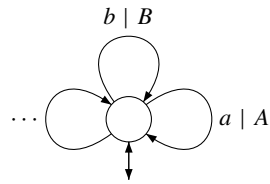


Figure 1.28. From lowercase to uppercase.

As for automata, we allow in the definition of transducers the input and output labels to be arbitrary, possibly empty, words. The behaviour of the transducer can be viewed as a machine reading an input word and writing an output word through two "heads" (see Figure 1.27). The mechanism is asynchronous in the sense that the two heads may move at different speeds.

The particular case of *synchronous* transducers is important. A transducer is said to be synchronous if, for each edge, the input label and the output label are letters. Not every rational relation can be realized by a synchronous transducer. Indeed, if ρ is realized by a synchronous transducer, then ρ is length-preserving. This means that whenever $(x, y) \in \rho$, then $|x| = |y|$.

A transducer is *literal* if for each edge the input label and the output label are letters or the empty word. It is not difficult to show that any transducer can be replaced by a literal one.

Example 1.5.5. The relation between a word written in lower-case letters $a, b, c, \ldots$ and the corresponding upper-case letters $A, B, C, \ldots$ is rational. Indeed, it is described by the expression $((a, A) \cup (b, B) \cup \ldots)^*$. This relation is realized by the transducer of Figure 1.28. This transducer is both literal and synchronous.

Example 1.5.6. The Fibonacci morphism defined by $a \to ab$, $b \to a$ is realized by the transducer on the left of Figure 1.29. The transducer on the right of Figure 1.29 realizes the same morphism. It is literal.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

Figure 1.29. The Fibonacci morphism.

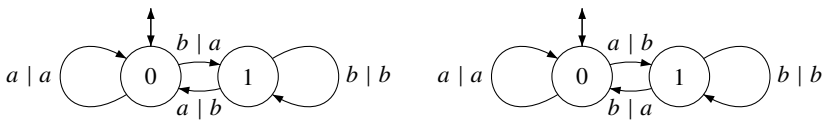


Figure 1.30. The circular right shift on words ending with a and its inverse.

Example 1.5.7. The transducer represented on the left of Figure 1.30 realizes the *circular right shift* on a word on the alphabet $\{a, b\}$ ending with the letter a. The transformation consists in shifting cyclically each symbol one place to the right. For example

$$\begin{array}{c} a\ b\ b\ a\ b\ a \\ a\ a\ b\ b\ a\ b \end{array}$$

The restriction to words ending with the letter a is for simplicity (and corresponds to the choice of state 0 as initial and final state in the automaton on the left of Figure 1.30). The inverse of the right shift is the left shift which shifts all symbols cyclically one place to the left. Its restriction to words beginning with a is represented on the right of Figure 1.30. The composition of both transformations is the identity restricted to words ending with the letter a plus the empty word.

An important property of rational relations is that *the composition of two rational relations is again a rational relation*. The construction of a transducer realizing the composition is the following. We start with a transducer $\mathfrak{S} = (Q, E, I, T)$ over $\mathcal{A}, \mathcal{B}$ and a transducer $\mathfrak{S}' = (Q', E', I', T')$ over $\mathcal{B}, \mathcal{C}$. We suppose that $\mathfrak{S}$ and $\mathfrak{S}'$ are literal (actually we shall only need that the output automaton of $\mathfrak{S}$ is literal and that the input automaton of $\mathfrak{S}'$ is literal). We build a new transducer $\mathfrak{U}$ as follows. The set of states of $\mathfrak{U}$ is $Q \times Q'$. The set of edges is formed of three kinds of edges:

1. The set of edges $(p, p') \xrightarrow{a|c} (q, q')$ for all edges $p \xrightarrow{a|b} q$ in E and $p' \xrightarrow{b|c} q'$ in E'.
2. The set of edges $(p, p') \xrightarrow[a|\varepsilon]{\varepsilon|c} (p, q')$ for $p' \xrightarrow[a|\varepsilon]{\varepsilon|c} q'$ in E'.
3. The set of edges $(p, p') \longrightarrow (q, p')$ for $(p \longrightarrow q)$ in E.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

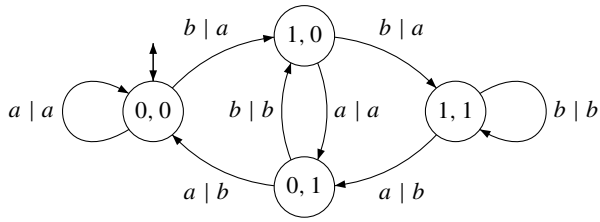


Figure 1.31. The right 2-shift.

The set of initial states of $\mathfrak{U}$ is $I \times I'$ and the set of terminal states is $T \times T'$. The definition of the edges implies that

$$(p, r) \xrightarrow{x|z} (q, s) \iff \exists y : p \xrightarrow{x|y} q \text{ and } r \xrightarrow{y|z} s.$$

This allows us to prove that the composed transducer realizes the composition of the relations.

Example 1.5.8. The composition of the circular right shift of Example 1.5.7 with itself produces the circular right 2-shift which consists in cyclically shifting the letters two places to the right for words ending with *aa* (see Figure 1.31).

For the implementation of transducers we use a function NEXT(p) which associates to each state p the set of edges beginning at p, and two sets INITIAL and TERMINAL to represent the initial and terminal states.

The algorithm computing the composition of two transducers is easy to write.

```
COMPOSETRANSDUCERS(𝔖, 𝔗)
 1 ▷ 𝔖 and 𝔗 are literal transducers
 2 𝔘 ← NEWTRANSDUCER()
 3 for each edge (p, a, b, q) of 𝔖 do
 4       for each edge (r, b, c, s) of 𝔗 do
 5             add ((p, r), a, c, (q, s)) to the edges of 𝔘
 6 for each edge (p, a, ε, q) of 𝔖 do
 7       for each state r of 𝔗 do
 8             add ((p, r), a, ε, (q, r)) to the edges of 𝔘
 9 for each edge (r, ε, c, s) of 𝔗 do
10       for each state p of 𝔖 do
11             add ((p, r), ε, c, (p, s)) to the edges of 𝔘
12 INITIAL_𝔘 ← INITIAL_𝔖 × INITIAL_𝔗
13 TERMINAL_𝔘 ← TERMINAL_𝔖 × TERMINAL_𝔗
14 return 𝔘
```

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

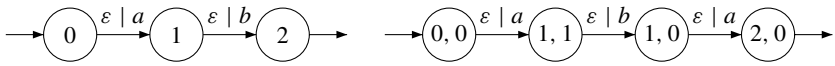


Figure 1.32. The image $f(x) = aba$ of $x = ab$ by the Fibonacci morphism.

The composition can be used to compute an automaton that recognizes the image of a word (and more generally of a regular set) by a rational relation. Indeed, let ρ be a rational relation from $\mathcal{A}^*$ to $\mathcal{B}^*$, let x be a word over $\mathcal{A}$. Let $\mathfrak{R}$ be a literal transducer realizing ρ, and let $\mathfrak{A}$ be a literal transducer realizing the relation $\{(\varepsilon, x)\}$. Let $\mathfrak{T} = \textsc{Compose}(\mathfrak{A}, \mathfrak{R})$ be the composition of $\mathfrak{A}$ and $\mathfrak{R}$. The image $f(x) = \{y \in \mathcal{B}^* \mid (x, y) \in \rho\}$ is recognized by the output automaton of $\mathfrak{T}$.

Example 1.5.9. Consider the word $x = ab$ and the Fibonacci morphism of Example 1.5.6. On the left of Figure 1.32 is a transducer realizing $\{(\varepsilon, x)\}$, and on the right the transducer obtained by composing it with the literal transducer of Figure 1.29. The composition of the transducers actually contains an additional edge $(0, 1) \xrightarrow{\varepsilon|b} (0, 0)$ which is useless because the state $(0, 1)$ is inaccessible from the initial state.

A *sequential transducer* over $\mathcal{A}$, $\mathcal{B}$ is a triple (Q, i, T) together with a partial function

$$Q \times \mathcal{A} \to \mathcal{B}^* \times Q$$

which breaks up into a *next state* function $Q \times \mathcal{A} \to Q$ and an output function $Q \times \mathcal{A} \to \mathcal{B}^*$. As usual, the next state function is denoted $(q, a) \mapsto q \cdot a$ and the output function $(q, a) \mapsto q * a$. In addition, the initial state $i \in Q$ has a word λ attached to it called the *initial prefix* and T is actually a (partial) function $T : Q \to B^*$ called the *terminal function*. Thus, an initial prefix and additional suffix can be added to all outputs.

The next state and the output functions are extended to words by $p \cdot (xa) = (p \cdot x) \cdot a$ and $p * (xa) = (p * x)(p \cdot x) * a$. The second formula means that the output $p * (xa)$ is actually the product of the words $p * x$ and $q * a$ where $q = p \cdot x$. The (partial) function f from $\mathcal{A}^*$ to $\mathcal{B}^*$ *realized* by the sequential transducer is defined by $f(x) = \lambda v \tau$ where u is the initial prefix, $v = i * x$ and $\tau = T(i \cdot x)$. A function from $\mathcal{A}^*$ to $\mathcal{B}^*$ that is realized by a sequential transducer is called a *sequential function*.

Example 1.5.10. The circular left shift on words over $\{a, b\}$ beginning with a is realized, on the right of Figure 1.30, by a transducer which is not sequential (two edges with input label a leave state 0). It can also be computed by the sequential transducer of Figure 1.33 with the initial pair $(\varepsilon, 0)$ and the terminal function $T(1) = a$.

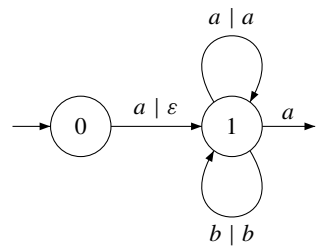


Figure 1.33. A sequential transducer for the circular left shift on words beginning with a.

The composition of two sequential functions is again a sequential function. This is actually a particular case of the composition of rational functions. The same construction is used to compose sequential transducers and it happens to produce a sequential transducer. We give explicitly the form of the composed transducer.

Let $\mathfrak{S} = (Q, i, T)$ be a sequential transducer over $\mathcal{A}$, $\mathcal{B}$ and let $\mathfrak{S}' = (Q', i', T')$ be a sequential transducer over $\mathcal{B}, \mathcal{C}$. The *composition* of $\mathfrak{S}$ and $\mathfrak{S}'$ is the sequential transducer $\mathfrak{S} \circ \mathfrak{S}'$ with set of states $Q' \times Q$, initial state (i', i) and terminal states $T'' = T' \times T$. Observe that we reverse the order for notational convenience. The next state function and the output function are given by

$$\begin{aligned}(p', p) \cdot x &= (p' \cdot (p * x), p \cdot x)\\ (p', p) * x &= p' * (p * x).\end{aligned}$$

The initial prefix of the composed transducer is the word $\lambda'' = \lambda'(i' * \lambda)$, and the terminal function T'' is defined by

$$T''(q', q) = (q' * T(q))T'(q' \cdot T(q))\,.$$

The value of the terminal function T'' on (q', q) is indeed obtained by first computing the value of the terminal function $T(q)$ and then fitting this word in the transducer $\mathfrak{S}'$ at state q'.

For the implementation of sequential transducers we use a partial function $\textsc{Next}(p, a) = (p * a, p \cdot a)$ grouping the output function and the next state function. There is also a pair $\textsc{Initial} = (\lambda, i) \in B^* \times Q$ for the initial prefix and the initial state and a partial function $\textsc{Terminal}(q)$ returning the terminal suffix for each terminal state $q \in T$.

1.5.1. Determinization of transducers

Contrary to ordinary automata, it is not true that any finite transducer is equivalent to a finite sequential one. It can be verified that a transducer is

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

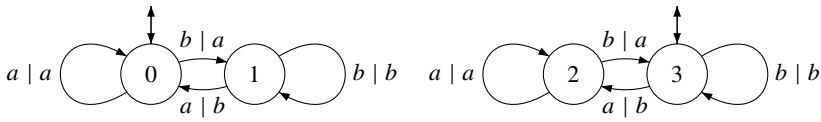


Figure 1.34. The circular right shift.

equivalent to a sequential one if and only if it realizes a partial function and if it satisfies a condition called the twinning property defined as follows. Consider a pair of paths with the same input label and of the form

$$\begin{aligned} i &\xrightarrow{u|u'} q \xrightarrow{v|v'} q \\ i' &\xrightarrow{u|u''} q' \xrightarrow{v|v''} q' \end{aligned}$$

where i and i' are initial states. Two paths such as these are called *twin*. The *twinning property* is that for any pair of twin paths, the output is such that v', v'' are conjugate and $u'v'v' \cdots = u''v''v'' \cdots$.

Example 1.5.11. The circular right shift on all words over $\{a, b\}$ is realized by the transducer of Figure 1.34. It is not a sequential function because the last letter cannot be guessed before the end of the input. Formally, this is visible because of the twin paths

$$0 \xrightarrow{b|a} 1 \xrightarrow{ab|ba} 1$$

and

$$3 \xrightarrow{b|b} 3 \xrightarrow{ab|ba} 3,$$

with distinct outputs $ababa \cdots$ and $bbababa \cdots$.

The computation of an equivalent sequential transducer is a variant of the determinization algorithm of automata. The main difference is that it may fail to terminate since, as we have seen before, it cannot always be performed successfully. We start with a transducer $\mathfrak{A}$ which is supposed to be equivalent to a sequential one. We suppose that $\mathfrak{A}$ is literal (or, at least, that its input automaton is literal) and trim. The states of the equivalent sequential transducer $\mathfrak{B}$ are sets of pairs $(u, q) \in B^* \times Q$. A pair $(u, q) \in B^* \times Q$ is called a *half-edge*. The states are computed by using in a first step a function NEXT() represented below. The value of NEXT(S, a) on a set S of half-edges and a letter a is the union, for $(u, p) \in S$ of the set of half-edges (uvw, r) such that there are, in $\mathfrak{A}$,

(i) an edge $p \xrightarrow{a|v} q$,

(ii) and a path $q \xrightarrow{\varepsilon|w} r$.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

We use a function $\text{NEXT}_{\mathfrak{A}}(p, a)$ returning the set of half-edges (v, q) such that (p, a, v, q) is an edge of the transducer $\mathfrak{A}$.

```
NEXT(S, a)
1 ▷ S is a set of half-edges (u, q) ∈ B* × Q, and a is a letter
2 T ← ∅
3 for (u, p) ∈ S do
4       for (v, q) ∈ NEXT_A(p, a) do
5             T ← T ∪ (uv, q)
6 return CLOSURE(T)
```

The set $\text{CLOSURE}(T)$ is the set of half-edges (uw, r) such that there is a path $q \xrightarrow{\varepsilon|w} r$ in $\mathfrak{A}$ for some half-edge $(u, q) \in T$. If the transducer is equivalent to a deterministic one, this set is finite. The computation of $\text{CLOSURE}(T)$ uses as usual an exploration of the graph composed of the edges of the form (q, ε, v, r). A test can be added to check that this graph has no loop whose label is a nonempty word, that is that the set $\text{CLOSURE}(T)$ is finite.

As an auxiliary step, we compute the following function

```
LCP(U)
1 ▷ U is a set of half-edges
2 v ← LONGESTCOMMONPREFIX(U)
3 U' ← ERASE(v, U)
4 return (v, U')
```

The function $\text{LONGESTCOMMONPREFIX}(U)$ returns the longest common prefix of the words u such that there is a pair $(u, q) \in U$. The function $\text{ERASE}(v, U)$ returns the set of half-edges obtained by erasing the prefix v of the words u appearing in the half-edges $(u, q) \in U$.

In a second step, we build the set of states and the next state function of the resulting sequential transducer $\mathfrak{B}$. As for automata, we use a function EXPLORE() which operates on the fly.

```
EXPLORE(T, S, B)
1 ▷ T is a collection of sets of half-edges
2 ▷ S is an element of T
3 for each letter a do
4       (v, U) ← LCP(NEXT(S, a))
5       NEXT_B(S, a) ← (v, U)
6       if U ≠ ∅ and U ∉ T then
7             T ← T ∪ U
8             (T, B) ← EXPLORE(T, U, B)
9 return (T, B)
```

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

We can finally write the function realizing the determinization of a transducer into a sequential one.

```
ToSequentialTransducer(𝔄)
 1 ▷ 𝔄 is a transducer
 2 𝔅 ← NewSequentialTransducer()
 3 I ← Closure({ε} × Initial_𝔄)
 4 Initial_𝔅 ← I
 5 ▷ 𝒯 is a collection of sets of half-edges
 6 𝒯 ← I
 7 (𝒯, 𝔅) ← Explore(𝒯, I, 𝔅)
 8 for S ∈ 𝒯 do
 9       for (u, q) ∈ S do
10             if q ∈ Terminal_𝔄 then
11                   Terminal_𝔅(S) ← u
12 return 𝔅
```

Example 1.5.12. The application of the determinization algorithm to the transducer on the right of Figure 1.30 produces the sequential transducer of Figure 1.33 as obtained on Figure 1.35.

A test can be added to the determinization algorithm to stop the computation in case of failure, that is if one of the following situations occurs, implying that the transducer $\mathfrak{A}$ is not equivalent to a sequential one. First, one may check at line 4 in algorithm Explore() that the half edges appearing in a state of $\mathfrak{B}$ have a label of bounded length. Indeed, it can be shown that there exists a constant K, depending on $\mathfrak{A}$ such that for each half-edge (u, q) appearing in a state of $\mathfrak{B}$, the length of u is bounded by K (otherwise $\mathfrak{A}$ does not satisfy the twinning property, see Problem 1.5.1). Second, a test can be added at line 10 of algorithm ToSequentialTransducer() to check that if a state of $\mathfrak{B}$ contains two half-edges (u, q) and (v, r) with q, r

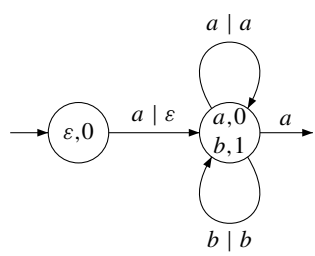


Figure 1.35. A sequential transducer for the circular left shift on words beginning with a obtained by the determinization algorithm.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

terminal, then $u = v$ (if this condition fails to hold, then $\mathfrak{A}$ does not realize a function).

1.5.2. Minimization of transducers

Just as there is a unique minimal deterministic automaton equivalent to a given one, there is also a unique minimal sequential transducer equivalent to a given one. The minimization of sequential transducers consists of two steps. A preliminary one, called *normalization*, allows output to be produced as soon as possible. The second step is quite similar to the minimization of finite automata.

Let $\mathfrak{A} = (Q, i, T)$ be a sequential transducer. For each state $p \in Q$, let us denote by $\mathcal{X}_p$ the subset of $\mathcal{B}^*$ recognized by the output automaton corresponding to $\mathfrak{A}$ with p as initial state. The normalization consists in computing for each state $p \in Q$, the longest common prefix π_p of all words in $\mathcal{X}_p$.

The normalized transducer is obtained by modifying the output function and terminal function of $\mathfrak{A}$. We set

$$\lambda' = \lambda\pi_i, \quad p *' a = \pi_p^{-1}(p * a)\pi_{p.a}, \quad T'(p) = \pi_p^{-1}T(p).$$

The computation of the words π_p can be performed as follows. It uses the binary operation associating to two words their longest common prefix. This operation is associative and commutative and will be denoted in this section by a $+$, like a sum. We consider the set $K = \mathcal{B}^* \cup 0$ formed of $\mathcal{B}^*$ augmented with 0 as ordered by the relation $x \leq y$ if x is a prefix of y or $y = 0$. For $p, q \in Q$, we denote by $M_{p,q}$ the element of K which is the longest common prefix of all words v such that there is an edge $p \xrightarrow{a|v} q$ (and $M_{p,q} = 0$ if this set is empty). We also consider the Q-vector N defined by $N_p = T(p)$, where T is the terminal function, and $N_p = 0$ if $T(p)$ is empty. For a Q-vector X of elements of K, we consider the vector $Y = MX + N$ which is defined for $p \in Q$ by

$$Y_p = \sum_{q \in Q} M_{p,q} X_q + N_p.$$

Recall that all sums are in fact longest common prefixes and that the right-hand side of the equation above is the longest common prefix of the words $M_{p,q}X_q$, for $q \in Q$, and N_p. It can be checked that the function f defined by $f(X) = MX + N$ is order preserving for the partial order considered on the set K. Thus, there is a unique maximal fix-point which satisfies $X = MX + N$. This is precisely the vector of words $P = (\pi_p)$ we are looking for. It can be computed as the limit of the decreasing sequence $f^k(0)$ for $k = 1, 2, \ldots$

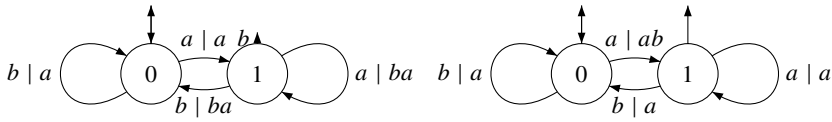


Figure 1.36. The normalization algorithm.

Example 1.5.13. Consider the transducer realizing the Fibonacci morphism represented on the right of Figure 1.29. The determinization of this transducer produces the sequential transducer on the left of Figure 1.36.

The computation of the vector P uses the transformation $Y = MX + N$ with

$$Y_0 = aX_1 + aX_0 + \varepsilon$$
$$Y_1 = baX_0 + baX_1 + b$$

The successive values of the vector P are $P = \begin{bmatrix} 0 & 0 \end{bmatrix}$, $P = \begin{bmatrix} \varepsilon & b \end{bmatrix}$. The last value satisfies $P = MP + N$ and thus it is the final one. The normalized transducer is shown on the right of Figure 1.36.

The algorithm to compute the array P is easy to write.

LongestCommonPrefixArray($\mathfrak{A}$)
1 ▷ P, P' are arrays of strings initially null
2 ▷ M is the matrix of transitions of $\mathfrak{A}$ and N the vector of terminals
3 **do** $P \leftarrow P'$
4 $P' \leftarrow MP + N$
5 **while** $P \neq P'$
6 **return** P

The expression $MP + N$ should be evaluated using the longest common prefix for the sum, including those appearing in the product MP. The normalized transducer can now be computed by the following function.

NormalizeTransducer($\mathfrak{A}$)
1 $P \leftarrow$ LongestCommonPrefixArray($\mathfrak{A}$)
2 $(\lambda, i) \leftarrow$ Initial
3 Initial $\leftarrow (\lambda P[i], i)$
4 **for** $(p, a) \in Q \times A$ **do**
5 $(u, q) \leftarrow$ Next(p, a)
6 Next$(p, a) \leftarrow P[p]^{-1} u P[q]$
7 **for** $p \in Q$ **do**
8 $T[p] \leftarrow P[p]^{-1} T[p]$

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

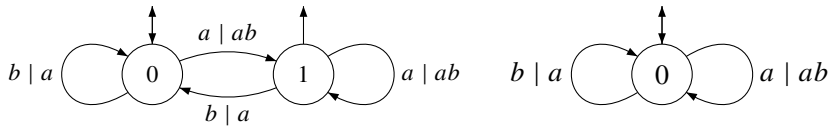


Figure 1.37. The minimization algorithm.

The last step of the minimization algorithm minimizes the input automaton, starting from the initial partition which is defined by $p \equiv q$ if $T(p) = T(q)$ and if $p * a = q * a$ for each $a \in A$. Any one of the minimization algorithms presented in Section 1.3.4 applies.

Example 1.5.14. We apply the minimization algorithm to the transducer obtained after normalization on the right of Figure 1.37. The two states are found to be equivalent.

The result is the sequential transducer on the right of Figure 1.37 which is of course identical to the transducer on the left of Figure 1.29.

1.6. Parsing

There are other ways, beyond regular expressions, to specify properties of words. In particular, context-free grammars offer a popular way to describe words satisfying constraints. These constraints often appear as the syntactic constraints defining programming languages or also natural languages. The patterns specified by regular expressions can also be expressed in this way, but grammars are strictly more powerful.

The problem of parsing or syntax analysis is that of computing the derivation tree of a word, given a grammar.

A *grammar* $\mathfrak{G}$ on an alphabet $\mathcal{A}$ is given by a finite set $\mathcal{V}$ and a finite set $\mathcal{R} \subset \mathcal{V} \times (\mathcal{A} \cup \mathcal{V})^*$. The elements of $\mathcal{V}$ are called *variables* and the elements of $\mathcal{R}$ are called the *productions* of the grammar. A production (v, w) is often written $v \to w$. One fixes moreover a particular variable $i \in \mathcal{V}$ called the *axiom*. The grammar is denoted by $\mathfrak{G} = (\mathcal{A}, \mathcal{V}, \mathcal{R}, i)$.

Given two words $x, y \in (\mathcal{A} \cup \mathcal{V})^*$, one writes $x \to y$ if y is obtained from x by replacing some occurrence of v by w for some production (v, w) in $\mathcal{R}$, that is if $x = pvq$, $y = pwq$. One denotes by $\overset{*}{\to}$ the reflexive and transitive closure of the relation $\to$. Thus $x \overset{*}{\to} y$ if there exists a sequence $w_0 = x, w_1, \ldots, w_n = y$ of words $w_h \in (\mathcal{A} \cup \mathcal{V})^*$ such that $w_h \to w_{h+1}$ for $0 \leq h < n$. Such a sequence is called a *derivation* from x to y. The *language generated* by the grammar $\mathfrak{G}$ is the set

$$L(\mathfrak{G}) = \{x \in \mathcal{A}^* \mid i \overset{*}{\to} x\}.$$

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

One may more generally consider the language generated by any variable v, denoted by $L(\mathfrak{G}, v) = \{x \in \mathcal{A}^* \mid v \overset{*}{\to} x\}$.

A grammar $\mathfrak{G} = (\mathcal{A}, \mathcal{V}, \mathcal{R}, i)$ can usefully be viewed as a system of equations, where the unknowns are the variables. Consider indeed the system of equations

$$v = W_v \quad (v \in \mathcal{V}) \tag{1.6.1}$$

where $W_v = \{w \mid (v, w) \in \mathcal{R}\}$. If each variable v is replaced by the set $L(\mathfrak{G}, v)$, one obtains a solution of the system of equations which is always the smallest solution (with respect to set inclusion) of the system.

A variant of the definition of a grammar is often used, where the sets W_v of Equation (1.6.1) are regular sets. In this case, these sets are usually described by regular expressions. This is equivalent to the first definition but often more compact. We give two fundamental examples of languages generated by a grammar.

Example 1.6.1. As a first example, let $\mathcal{A} = \{a, b\}$, $\mathcal{V} = \{v\}$, and $\mathcal{R}$ be composed of the two productions

$$v \to avv, \quad v \to b.$$

The language generated by the grammar $\mathfrak{G} = (\mathcal{A}, \mathcal{V}, \mathcal{R}, v)$ is known as the *Łukasiewicz language*. Its elements can be interpreted as arithmetic expressions in prefix notation, with a as an operator symbol and b as an operand symbol. The first words of $L(\mathfrak{G})$ in radix order are $b, abb, aabbb, ababb, aaabbbb, aababbb, aabbabb, abaabbb, \ldots$ In alphabetic order (with $a < b$) the last words are $\ldots, abb, b$.

Example 1.6.2. The second fundamental example is the *Dyck language* generated by the grammar $\mathfrak{G}$ with the same sets $\mathcal{A}, \mathcal{V}$ as above and the productions

$$v \to avbv, \quad v \to \varepsilon.$$

Let $\mathcal{M}$ be the language generated by this grammar. Then $\mathcal{M} = a\mathcal{M}b\mathcal{M} + \varepsilon$. Set $\mathcal{D} = a\mathcal{M}b$. Then $\mathcal{M} = \mathcal{D}\mathcal{M} + \varepsilon$. This shows that $\mathcal{M} = \mathcal{D}^*$, and thus $\mathcal{D} = a\mathcal{D}^*b$. The set $\mathcal{M}$ is called the *Dyck language*, and $\mathcal{D}$ is the set of *Dyck primes*. The words in $\mathcal{M}$ can be viewed as well-formed sequences of parentheses with a as left parenthesis and b as right parenthesis. The words of $\mathcal{D}$ are the words in $\mathcal{M}$ which are not products of two nonempty words of $\mathcal{M}$. The first words in radix order in $\mathcal{D}$ and in $\mathcal{D}^*$ are respectively $ab, aabb, aababb, \ldots,$ and $\varepsilon, ab, aabb, abab, aabbab$. A basic relation between the Łukasiewicz set $\mathcal{L}$ and the Dyck language $\mathcal{M}$ is the equation

$$\mathcal{L} = \mathcal{M}b\,.$$

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

This is easy to verify, provided one uses the equational form of the grammar. The set $\mathcal{L}$ is indeed uniquely defined as the solution of the equation

$$\mathcal{L} = a\mathcal{L}\mathcal{L} + b \tag{1.6.2}$$

Since $\mathcal{M} = a\mathcal{M}b\mathcal{M} + \varepsilon$, multiplying both sides by b on the right, we obtain

$$\mathcal{M}b = a\mathcal{M}b\mathcal{M}b + b\,.$$

which is Equation (1.6.2), whence $\mathcal{M}b = \mathcal{L}$. There is a simple combinatorial interpretation of this identity. Let $\delta(x)$ denote the difference of the number of occurrences of a and of b in the word x. One can verify that a word x is in $\mathcal{M}$ if and only if $\delta(x) = 0$ and $\delta(p) \geq 0$ for each prefix p of x. Similarly, a word x is in $\mathcal{L}$ if and only if $\delta(x) = -1$ and $\delta(p) \geq 0$ for each proper prefix p of x.

A *derivation tree* for a word w is a tree T labelled by elements of $\mathcal{A} \cup \mathcal{V} \cup \{\varepsilon\}$ such that:

1. The root of T is labelled by i.
2. For each interior node n, the pair (v, x) formed by the label v of n and the word x obtained by concatenating the labels of the children of n in left to right order is an element of $\mathcal{R}$.
3. A leaf is labelled ε only if it is the unique child of its parent.
4. The word w is obtained by concatenating the labels of the leaves of T in increasing order.

A derivation tree is a useful shorthand for representing a set of derivations. Indeed, any traversal of the derivation tree produces a derivation represented by this tree, and conversely (see Figure 1.38 for a derivation tree in the Dyck grammar).

We now present in an informal manner two strategies for syntax analysis. Given a grammar $\mathfrak{G}$ and a word x, we want to be able to check whether x is in $L(\mathfrak{G})$. This amounts to building a derivation $i \xrightarrow{*} x$ from the axiom i

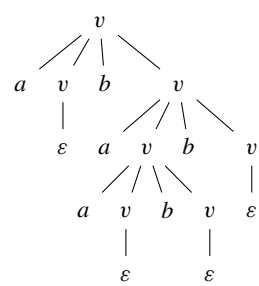


Figure 1.38. A derivation tree for the word *abaabb* in the Dyck grammar.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

of $\mathfrak{G}$ to x. There are two main options for doing this. The first one, called top-down parsing, builds the derivation from left to right (from i to x). This corresponds to constructing the derivation tree from the root to the leaves. The second one, called bottom-up parsing, builds the derivation from right to left (from x backwards to i). This corresponds to constructing the derivation tree from the leaves to the root.

1.6.1. Top-down parsing

The idea of top-down parsing is to build the derivation tree from the root. This is done by trying to build a derivation $i \overset{*}{\to} x$ and from left to right. The current situation in a top-down parsing is as follows (see Figure 1.39). A derivation $i \overset{*}{\to} yw$ has already been constructed. It has produced the prefix y of $x = yz$. It remains to build the derivation $w \overset{*}{\to} z$. We may assume that w starts with a variable v, that is $w = vs$. The key point for top-down parsing to work is that the grammar fulfils the following requirement. The pair (v, a), where a is the first letter of z, uniquely determines the production $v \to \alpha$ to be used, which is such that there exists a derivation $\alpha s \overset{*}{\to} z$. Grammars having this property for all x usually are called $LL(1)$ grammars.

We illustrate this method on two examples. The first one is the example of arithmetic expressions, and the second one concerns regular expressions already considered in Section 1.4. We consider the following grammar defining arithmetic expressions with operators $+$ and $*$ and parenthesis. The grammar allows unambiguous parsing of these expressions by introducing a hierarchy (expressions > terms > factors) reflecting the usual precedence of arithmetic operators ($* > +$).

$$\begin{aligned} E &\to E + T \mid T \\ T &\to T * F \mid F \\ F &\to (E) \mid c \end{aligned} \tag{1.6.3}$$

where c is any simple character.

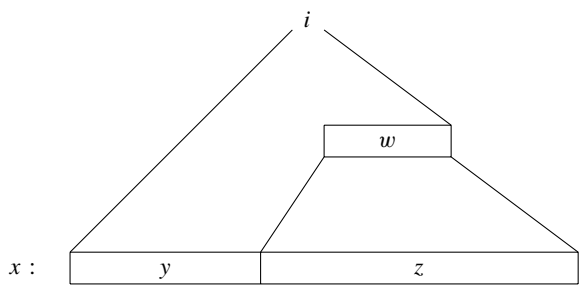


Figure 1.39. Top-down parsing.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

We want to write a program to evaluate such an expression using top-down parsing. The idea is to associate to each variable of the grammar a function which acts according to the right side of the corresponding production in the grammar. To manage the word to be analysed, a function CURRENT() gives the first letter of the suffix of the input word that remains to be analysed. In syntax analysis, the value of the function CURRENT() is called the *lookahead* symbol.

A function ADVANCE() allows one to progress on the input word. The value of CURRENT() allows one to choose the production of the grammar that should be used.

As already said, this method will work provided one may uniquely select, with the help of the value of CURRENT(), which production should be applied. However, we are already faced with this problem with the productions $E \to E + T$ and $E \to T$, because the first letter of the input word does not allow us to know whether there is a $+$ sign following the first term. This phenomenon is called *left recursion*. To eliminate this feature, we transform the grammar and replace the two rules above by the equivalent form $E = T(+T)^*$. This shows that every expression starts with a term, and the continuation of the derivation is postponed to the end of the analysis of the first term.

The function corresponding to the variables E is given in Algorithm EVALEXP. It returns the numerical value of the expression.

```
EVALEXP()
 1 v ← EVALTERM()
 2 while CURRENT() = '+' do
 3       ADVANCE()
 4       v ← v + EVALTERM()
 5 return v
```

The functions EVALTERM() and EVALFACT() corresponding to T and F are similar.

```
EVALTERM()
 1 v ← EVALFACT()
 2 while CURRENT() = '*' do
 3       ADVANCE()
 4       v ← v * EVALFACT()
 5 return v
EVALFACT()
 1 if CURRENT() = '(' then
 2       ADVANCE()
```

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

```
3       v ← EvalExp()
4       Advance()
5  else v ← Current()
6       Advance()
7  return v
```

The instruction at line 5 of the function EvalFact() assigns to v the numerical value corresponding to the current symbol.

The evaluation of an expression, involving the parsing of its structure, is realized by calling EvalExp().

As a second example, we show that the syntax of regular expressions can also be defined by a grammar. This is quite similar to the previously seen grammar of arithmetic expressions.

$$\begin{aligned} E &\to E + T \mid T \\ T &\to TF \mid F \\ F &\to G \mid G^* \\ G &\to (E) \mid c \end{aligned} \tag{1.6.4}$$

The symbol c stands for a letter or the symbol representing the empty word. A top-down parser for this grammar allows the implementation of the constructions of the previous section that produce a finite automaton from a regular expression.

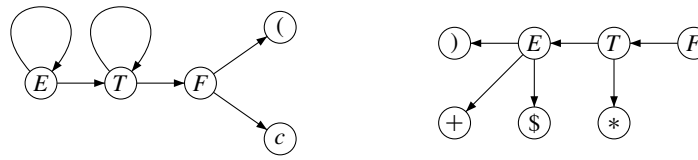
We have just seen top-down parsing developed on two examples. These examples show how easy it is to write a top-down analyser. The drawback of this method is that it assumes that the grammar defining the language has a rather restricted form. In particular, it should not be left recursive, although there exist standard procedures to eliminate left recursion. However, there exist grammars that cannot be transformed into equivalent $LL(1)$ grammars that allow top-down parsing. The letters L in the acronym $LL(1)$ refer to left to right processing (on both the text and the derivation), and the number 1 refers to the number of lookahead symbols.

The precise definition of $LL(1)$ grammars uses two functions called First() and Follow() that associate to each variable a set of terminal symbols. For a variable $x \in \mathcal{V}$, First(x) is the set of terminal symbols $a \in \mathcal{A}$ such that there is a derivation of the form $x \overset{*}{\to} au$. The function First() is extended to words in a natural way: First(w) is the set of terminal symbols a such that $w \overset{*}{\to} au$.

For each variable $x \in \mathcal{V}$, Follow(x) is the set of terminal symbols $a \in \mathcal{A}$ such that there is a derivation $u \overset{*}{\to} vxaw$ with a "following" x.

To compute First(), we build a graph with vertices $\mathcal{A} \cup \mathcal{V}$ and with edges the pairs $(x, y) \in \mathcal{V} \times (\mathcal{A} \cup \mathcal{V})$ such that there is a production of the form $x \to uyw$ with $u \overset{*}{\to} \varepsilon$. Then $a \in$ First(x) if and only if there is a

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press



(a) FIRST(). (b) FOLLOW().

Figure 1.40. The graphs of FIRST() and of FOLLOW().

path from x to a in this graph. The graph corresponding to the grammar of arithmetic expressions is shown on Figure 1.40(a).

The algorithm used to compute FIRST() is given more precisely below. We begin with an algorithm (EPSILON()) which computes a boolean array *epsilon* indicating whether a symbol v is nullable, that is whether $v \stackrel{*}{\rightarrow} \varepsilon$. The array *epsilon* has size $n + k$, where n is the number of variables in the grammar and k is the number of terminals.

```
EPSILON()
 1 for each production v → ε do
 2       epsilon[v] ← true
 3 for i ← 0 to n − 1 do
 4       for each production v → x_1 ··· x_m do
 5           epsilon[v] ← epsilon[v] ∨ (epsilon[x_1] ∧ ··· ∧ epsilon[x_m])
 6 return epsilon
```

It is easy to compute a function ISNULLABLE(w) for $w = x_1 \cdots x_n$ as the conjunction of the Boolean values $epsilon[x_i]$. The computation of FIRST() consists of several steps. We first compute the graph defined above. The graph is represented by the set FIRSTCHILD(v) of successors of each variable v. The function FIRST() is computed after a depth-first exploration of the graph has been performed.

```
FIRSTCHILD(v)
 1 ▷ S is the set of successors of v
 2 S ← ∅
 3 for each production v → x_1 ··· x_m do
 4       for i ← 1 to m do
 5           S ← S ∪ x_i
 6           if epsilon[x_i] = false then
 7               break
 8 return S
```

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

We mark vertices in the graph by a standard depth-first exploration.

```
EXPLOREFIRSTCHILD(v)
1 firstmark[v] ← true
2 for each x ∈ FIRSTCHILD(v) do
3       if firstmark[x] = false then
4             EXPLOREFIRSTCHILD(x)
```

The array *firstmark* is used for exploration of the graph of FIRST(). Finally, we compute FIRST().

```
FIRST(v)
1 ▷ firstmark is an array initialized to false
2 EXPLOREFIRSTCHILD(v)
3 S ← ∅
4 for each terminal c do
5       if firstmark[c] then
6             S ← S ∪ c
7 return S
```

The values of the function FIRST() could of course be stored in an array *first*. The extension of FIRST to words is straightforward.

```
FIRST(w)
1 S ← ∅
2 for i ← 1 to n do                ▷ w has length n
3       S ← S ∪ FIRST(w[i])
4       if epsilon[w[i]] = false then
5             break
6 return S
```

There is an alternative way to present the computation of FIRST(), by means of a system of mutually recursive equations. For this, observe that for each variable x, FIRST(x) is the union of the sets FIRST(y) over the set $S(x)$ of successors of x in the graph of FIRST(). Thus, the function FIRST() is the least solution of the system of equations

$$\text{FIRST}(x) = \cup_{y \in S(x)} \text{FIRST}(y) \qquad (x \in V)$$

such that FIRST(a) = a for each letter $a \in \mathcal{A}$. For example, the equations for Grammar (1.6.3) are

$$\begin{aligned} \text{FIRST}(E) &= \text{FIRST}(E) \cup \text{FIRST}(T) \\ \text{FIRST}(T) &= \text{FIRST}(T) \cup \text{FIRST}(F) \\ \text{FIRST}(F) &= \{(, c\} \end{aligned}$$

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

To compute the function FOLLOW(), we build a similar graph. There are two rules to define the edges.

1. If there is a production $x \to uvw$ with a terminal symbol a in FIRST(w), then (v, a) is an edge.
2. If there is a production $z \to uxw$ with $w \xrightarrow{*} \varepsilon$, then (x, z) is an edge (notice that we use the productions backwards).

The graph of FOLLOW() for the grammar of arithmetic expressions is shown on Figure 1.40(b). The computation of the function FOLLOW is analogous. It begins with the computation of the graph SIBLING(x).

```
SIBLING(x)
1 S ← ∅
2 for each production z → uxw do
3       S ← S ∪ FIRST(w)
4       if ISNULLABLE(w) then
5             S ← S ∪ z
6 return S
```

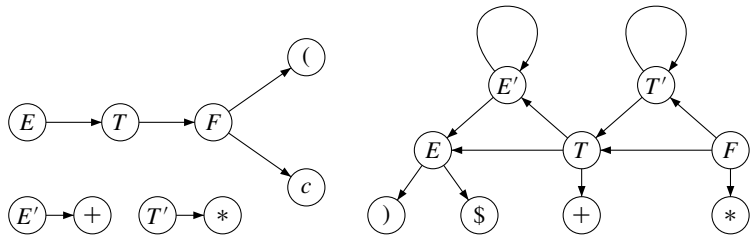
The depth-first exploration EXPLORESIBLING(v) is then performed as before. It produces an array *followmark* which is used to compute the function FOLLOW().

```
FOLLOW(v)
1 ▷ followmark is an array initialized to false
2 EXPLORESIBLING(v)
3 S ← ∅
4 for each terminal c do
5       if followmark[c] then
6             S ← S ∪ c
7 return S
```

As for the function FIRST(), the function FOLLOW() can also be computed by solving a system of equations. The precise definition of an $LL(1)$ grammar can now be formulated. It is a grammar such that:

1. For each pair of distinct productions $x \to u$, $x \to v$, with the same left-side and $u, v \neq \varepsilon$, one has
$$\text{FIRST}(u) \cap \text{FIRST}(v) = \emptyset.$$
2. For each pair of distinct productions of the form $x \to u$, $x \to \varepsilon$, one has
$$\text{FIRST}(u) \cap \text{FOLLOW}(x) = \emptyset.$$

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press



(a) FIRST(). (b) FOLLOW().

Figure 1.41. The graphs of FIRST() and FOLLOW() for Grammar (1.6.5).

Observe that our grammar for arithmetic expressions violates the first condition, since for instance $\text{FIRST}(E) = \text{FIRST}(T)$, although we have two productions $E \to E + T$ and $E \to T$ with the same left-hand side. We have already met this problem of left recursion, and solved it by transforming the grammar. The solution that we described is actually equivalent to considering the grammar

$$\begin{aligned} E &\to TE' \\ E' &\to +TE' \mid \varepsilon \\ T &\to FT' \\ T' &\to *FT' \mid \varepsilon \\ F &\to (E) \mid c. \end{aligned} \tag{1.6.5}$$

This grammar is equivalent to Grammar (1.6.3). It meets the two conditions for being *LL*(1). Indeed, the functions FIRST() and FOLLOW() are given in Figure 1.41.

For example, consider the productions $E' \to \varepsilon$ and $E' \to +TE'$. The symbol $+$ is not in $\text{FOLLOW}(E')$, and thus the second condition is satisfied for this pair of productions. The characterization allows us to fill the entries of a table called the *parsing table* given in Table 1.1. This is an

Table 1.1. The parsing table of Grammar (1.6.5).

	c	$+$	$*$	$($	$)$	$\$$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to c$			$F \to (E)$		

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

equivalent way to define the mutually recursive functions we defined above (for the wise: this is also a way to convince oneself that the programs are correct!)

The computation of the $LL(1)$ parsing table uses the following algorithm.

```
LLTable()
 1 ▷ computes the LL(1) parsing table M
 2 for each production p : v → w do
 3       for each terminal c ∈ First(w) do
 4             M[v][c] ← p
 5       if w = ε then
 6             for each terminal c ∈ Follow(v) do
 7                   M[v][c] ← p
 8 return M
```

The above algorithm as written supposes the grammar to be $LL(1)$. Error messages to inform that the grammar is not $LL(1)$ can easily be added.

1.6.2. Bottom-up parsing

We now describe bottom-up parsing which is a more complicated but more powerful method for syntax analysis.

The idea of bottom-up parsing is to build the derivation tree from the leaves to the root. This method is more complicated to program, but is more powerful than top-down parsing.

The current situation of bottom-up parsing is pictured in Figure 1.42. The left part of the text which has already been analysed has been reduced, using the productions backwards, to a string that is kept in a

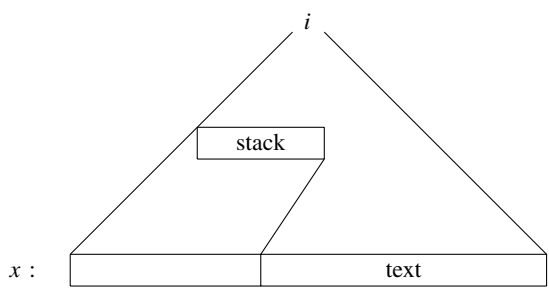


Figure 1.42. Bottom up parsing.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

stack. We will see below that this actually corresponds to a last-in first-out strategy.

We present bottom-up parsing on the example of arithmetic expressions already used.

$$\begin{array}{l} 1 : E \to E + T \\ 2 : E \to T \\ 3 : T \to T * F \\ 4 : T \to F \\ 5 : F \to (E) \\ 6 : F \to c \end{array} \tag{1.6.6}$$

We reproduce Grammar (1.6.3) with productions numbered from 1 to 6.

Programming a bottom-up analyser involves the management of a stack containing the part of the text that has already been analysed. The evolution of the stack and of the text is pictured below (Figure 1.43) to be read from top to bottom.

	Stack	Text
1		$(1+2)*3$
2	$($	$1+2)*3$
3	$(c$	$+2)*3$
4	$(F$	$+2)*3$
5	$(T$	$+2)*3$
6	$(E$	$+2)*3$
7	$(E+$	$2)*3$
8	$(E+c$	$)*3$
9	$(E+F$	$)*3$
10	$(E+T$	$)*3$
11	$(E$	$)*3$
12	(E)	$*3$
13	F	$*3$
14	T	$*3$
15	$T*$	3
16	$T*c$	
17	$T*F$	
18	T	
19	E	

Figure 1.43. Evolution of the stack and of the text during the bottom-up analysis of the expression $(1+2)*3$.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

At the beginning, the stack is empty. Each step either

1. transfers a new symbol from the text to the stack (this operation is called a *shift*);
2. reduces the top part of the stack according to a rule of the grammar (this is a *reduction*).

As an example, the second and third rows in Figure 1.43 are the results of shifts, while the three following rows are the results of reductions by rules 6, 4, and 2 respectively.

To be able to choose between shift and reduction, one uses a finite automaton called *LR automaton*. This automaton keeps track of the information concerning the presence of the right side of a rule at the top of the stack. In our example, the automaton is given in Figure 1.44.

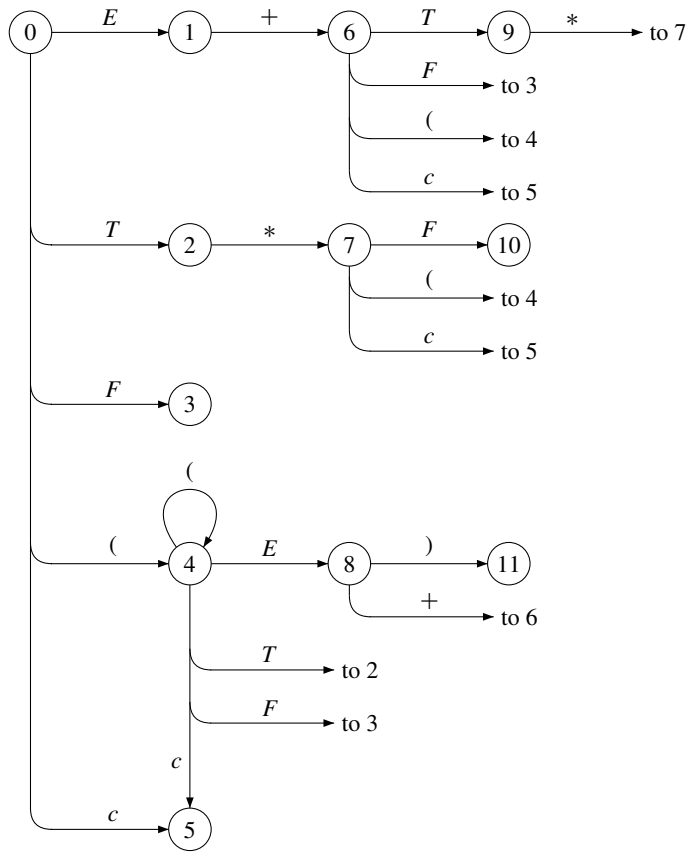


Figure 1.44. The LR automaton.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

Stack	Text
0	$(1 + 2) * 3\$$
0 4	$1 + 2) * 3\$$
0 4 5	$+2) * 3\$$
0 4 3	$+2) * 3\$$
0 4 2	$+2) * 3\$$
0 4 8	$+2) * 3\$$
0 4 8 6	$2) * 3\$$
0 4 8 6 5	$) * 3\$$
0 4 8 6 3	$) * 3\$$
0 4 8 6 9	$) * 3\$$
0 4 8	$) * 3\$$
0 4 8 11	$*3\$$
0 3	$*3\$$
0 2	$*3\$$
0 2 7	$3\$$
0 2 7 5	$\$$
0 2 7 10	$\$$
0 2	$\$$
0 1	$\$$

Figure 1.45. The stack of states of the *LR* automaton during the bottom-up analysis of the expression $(1 + 2) * 3$.

The input to the *LR* automaton is the content of the stack. According to the state reached, and to the lookahead symbol, the decision can be made whether to shift or to reduce, and in the latter case by which rule. The fact that this is possible is a property of the grammar. These grammars are called *SLR*–grammars.

In practice, instead of pushing the symbols on the stack, one rather pushes the states of the *LR* automaton. The result on the expression $(1 + 2) * 3$ is shown on Figure 1.45.

The decision made at each step uses two arrays S and R, represented on Figure 1.46.

The array S is the transition table of the *LR* automaton. Thus $S[p][c]$ is the state reached from state p by reading c. The table R indicates which reduction to perform. The value $R[p][c]$ indicates the number of the production to be used backwards to perform a reduction when the state p is on top of the stack and the symbol c is the lookahead symbol. Empty entries in tables S and R correspond to nonexisting transitions. A special state *Accept*, abbreviated as *Acc* is the accepting state ending the computation

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

	c	+	∗	(	)	$	E	T	F
0	5			4			1	2	3
1		6				*Acc*			
2			7						
3									
4	5			4			8	2	3
5									
6	5			4				9	3
7	5			4					10
8		6			11				
9			7						
10									
11									

(a) The array S.

	c	+	∗	(	)	$	E	T	F
0									
1									
2		2			2	2			
3		4	4		4	4			
4									
5		6	6		6	6			
6									
7									
8									
9		1			1	1			
10		3	3		3	3			
11		5	5		5	5			

(b) The array R.

Figure 1.46. The arrays S and R.

with a successful analysis. The tables S and R could be superposed because their nonempty entries are disjoint. Actually, this is necessary for the LR-algorithm to work!

The implementation of the algorithm is given in the function LR-PARSE(). It uses, on the input, the two functions CURRENT(), ADVANCE() already described earlier, and the symbol '$' to mark the end of the text. The functions TOP() and PUSH() are the usual functions on stacks. The function REDUCE() operates in three steps. The call REDUCE(n), where n is the index of the production $r \to u$, consists of the following:

1. It erases from the stack the number of states equal to the length of u.
2. It computes the new value $p =$ TOP() and the state $q = T[p][r]$.
3. It pushes q on the stack.

In the implementation, the value -1 represents nonexisting transitions. The function returns the Boolean value **true** if the analysis was successful, and **false** otherwise. There are three cases of failure:

1. There is no legal shift nor legal reduction, this is checked at lines 5 and 9. This happens for instance if the input is $x =$ ')'.
2. The text has not been exhausted at the end of the analysis, for instance if $x =$ '('; this leads to the same situation as above, because the state *Accept* can only be accessed by the end marker.
3. The text has been exhausted before the end of the analysis; in this case, the end marker leads to an empty entry in the tables.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

```
LRPARSE(x)
 1 while TOP() ≠ Accept do
 2       p ← TOP()
 3       c ← CURRENT()
 4       q ← T[p][c]
 5       if q ≠ −1 then
 6             PUSH(q)
 7             ADVANCE()
 8       else  n ← R[p][c]
 9             if n ≠ −1 then
10                   REDUCE(n)
11             else return false
12 return true
```

It remains to explain how to compute the *LR* automaton and the corresponding tables from the grammar. We work with an end marker '$'. Accordingly, we add to the grammar an additional rule which, in our running example, is $E' \to E\$$. The *LR* automaton recognizes the content of the stack and its state allows one to tell whether the right side of some production is present on the top of the stack. The set of possible stack contents (sometimes called the *viable prefixes*) is the set

$$\mathcal{X} = \{p_1 p_2 \cdots p_n \mid p_i \in P, n \geq 0\}$$

where P is the set of prefixes of the right sides of the productions and where, for each i, $1 \leq i \leq n-1$, there is a production (x_i, v_i) such that $p_i x_{i+1}$ is a prefix of v_i, and x_1 is the axiom of the grammar. One may verify this description of $\mathcal{X}$ by working on the bottom-up analysis backwards. It is easy to build a nondeterministic automaton recognizing this set $\mathcal{X}$. It is built from the automata recognizing the right sides of the productions and adding ε-transitions from each position before a variable y to the initial positions of the productions with left side y.

The result is represented on Figure 1.47. The circled states correspond to full right-hand sides and thus to productions of the grammar.

To be complete, we should add the transitions corresponding to the rule $E' \to E\$$. The states 0 and 4 correspond to the productions with left side E. The automaton of Figure 1.44 is just the result of the determinization algorithm applied to the nondeterministic automaton obtained. This explains how the *LR* automaton and thus the table S, which is just its transition table, are built. It still remains to explain how table R is built. We have $R[p][c] = n$ if and only if the reduction by production $n : x \to v$ is

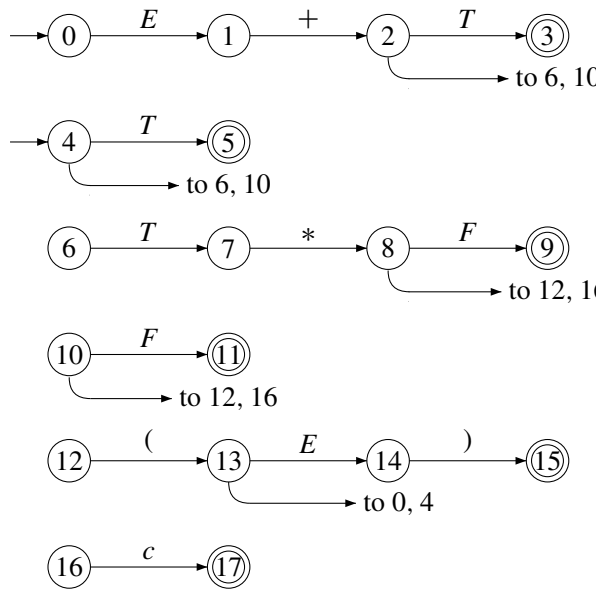


Figure 1.47. A non deterministic LR automaton

possible in state p, and provided the lookahead symbol c is in FOLLOW(x). This solves the conflicts between shift and reduce.

Suppose for example that the variable T is on top of the stack, as at lines 5, 14, 18 of Figure 1.43. At each of these lines, we can either reduce by production 2 or shift. Similarly, at line 10 we can either reduce by production 1 or 2, or shift. We should reduce only if the lookahead symbol is in FOLLOW(E). This is why we choose to reduce by production 2 at lines 5 and 18. At line 14, we choose to shift, because the symbol $*$ is not in FOLLOW(E). At line 10, we reduce by production 1 because the corresponding state 9 allows this reduction and the lookahead symbol ')' is in FOLLOW(E).

A grammar for which this method works is called *SLR*(1). A word on this terminology. The acronym *LR* refers to a left to right analysis of the text and a rightmost derivation (corresponding to a bottom-up analysis). A grammar is said to be *LR*(0) if no shift-reduce conflict appears on the *LR* automaton. The 0 means that no lookahead is needed to make the decisions. This is not the case of Grammar (1.6.6), as we have seen. The acronym *SLR* means 'simple LR' and the integer 1 refers to the length of the lookahead. Formally, a grammar is said to be *SLR*(1) if for any state p of the *LR* automaton and each terminal symbol c, at most one of the two following cases arises.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

1. There is a transition from p by c in the automaton.
2. There is a possible reduction in state p by production $n : x \to v$ such that $c \in \text{FOLLOW}(x)$.

In practice, this condition is equivalent to the property that the sets of nonempty entries of the tables S and R are disjoint.

More complicated methods exist, either with lookahead 1 or with a larger lookahead, although a lookahead of size larger than 1 is rarely used in practice. With lookahead 1, the class of $LR(1)$ grammars uses an automaton called the $LR(1)$ automaton to keep track of the pair (s, c) of the stack content s and the lookahead symbol c to be expected at the next reduction. The main drawback is that the number of states is much larger than with the $LR(0)$ automaton.

1.7. Word enumeration

One often has to compute the number of words satisfying some property. This can be done using finite automata or grammars as illustrated in the following examples.

1.7.1. Two illustrative examples

The first example illustrates the case of a property defined by a finite automaton.

Example 1.7.1. The number u_n of words of length n on the binary alphabet $\{a, b\}$ which do not contain two consecutive letters a satisfies the recurrence formula $u_{n+1} = u_n + u_{n-1}$. Indeed, a nonempty word of length n can terminate with either a or b. In the first case, it has to terminate with ba unless it is the word a. Since $u_0 = 1$ and $u_1 = 2$, the number u_n is the Fibonacci number F_{n+2}.

This argument can be used quite generally when the corresponding set of words is recognized by a finite automaton. In the present case, the set $\mathcal{S}$ without factor bb is recognized by the Golden mean automaton of Figure 1.11. Let $\mathcal{S}_q$ be the set of words recognized by the automaton with initial state 1 and final state q. We derive from the automaton the following set of equations

$$\begin{aligned} \mathcal{S}_1 &= \mathcal{S}_1 b + \mathcal{S}_2 b + \varepsilon \\ \mathcal{S}_2 &= \mathcal{S}_1 a \end{aligned}$$

Since $\mathcal{S} = \mathcal{S}_1 + \mathcal{S}_2$, summing up the equations gives

$$\mathcal{S} = \mathcal{S}b + \mathcal{S}_1 a + \varepsilon = \mathcal{S}(b + ab) + \varepsilon.$$

This gives the expected recurrence relation.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

A second example concerns the Dyck language.

Example 1.7.2. Recall from Example 1.6.2 that the Dyck language $\mathcal{D}^*$ is related to the Łukasiewicz language $\mathcal{L}$ by the relation $\mathcal{D}^*b = \mathcal{L}$. Let f_n be the number of words of length n in $\mathcal{D}$ and let u_n be the number of words of length n in $\mathcal{D}^*$.

It can be verified, using the function δ of Example 1.6.2, that each word x of length $2n+1$ with $\delta(x) = -1$ is primitive and has exactly one conjugate in $\mathcal{L}$. Since u_{2n} is also the number of words of length $2n+1$ in $\mathcal{L}$, one gets

$$u_{2n} = \frac{1}{2n+1}\binom{2n+1}{n} = \frac{1}{n+1}\binom{2n}{n}.$$

Since $\mathcal{D} = a\mathcal{D}^*b$, it follows that

$$f_{2n} = \frac{1}{n}\binom{2n-2}{n-1}.$$

The sequence (u_{2n}) is the sequence of *Catalan numbers*.

The combinatorial method used to compute the numbers f_n and u_n can be frequently generalized in the case of more complicated grammars (see Chapter 9). In the present case, the relation is the following.

We start with the relation $\mathcal{D} = a\mathcal{D}^*b$. This implies that the generating function $D(z) = \sum_{n\geq 0} f_n z^n$ satisfies the equation

$$D^2 - D + z^2 = 0\,.$$

It follows that

$$D(z) = \frac{1 - \sqrt{1-4z^2}}{2}\,.$$

An elementary application of the binomial formula gives the expected expression for the coefficient f_n.

1.7.2. The Perron–Frobenius theorem

Several enumeration problems on words involve the spectral properties of nonnegative matrices. The *Perron–Frobenius theorem* describes some of these properties and constitutes a very important tool in this framework. We shall see in the next section several applications of this theorem.

Let Q be a set of indices (we have of course in mind the set of states of a finite automaton). For two Q-vectors v, w with real coordinates, one writes $v \leq w$ if $v_q \leq w_q$ for all $q \in Q$ and $v < w$ if $v_q < w_q$ for all $q \in Q$. A vector v is said to be *nonnegative* (resp.*positive*) if $v \geq 0$ (resp. $v > 0$). In the

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

same way, for two $Q \times Q$-matrices M, N with real coefficients, one writes $M \leq N$ when $M_{p,q} \leq N_{p,q}$ for all $p, q \in Q$ and $M < N$ when $M_{p,q} < N_{p,q}$ for all $p, q \in Q$. The $Q \times Q$-matrix M is said to be *nonnegative* (resp. *positive*) if $M \geq 0$ (resp. $M > 0$). We shall often use the elementary fact that if $M > 0$ and $v \geq 0$ with $v \neq 0$, then $Mv > 0$.

A nonnegative matrix M is said to be *irreducible* if for all indices p, q, there is an integer k such that $M^k_{p,q} > 0$, where M^k denotes the kth power of M. It is easy to verify that M is irreducible if and only if $(I + M)^n > 0$ where n is the dimension of M. It is also easy to prove that M is reducible (i.e. M is not irreducible) if there is a reordering of the indices such that M is block triangular, that is of the form

$$M = \begin{bmatrix} U & V \\ 0 & W \end{bmatrix} \tag{1.7.1}$$

with U, W of dimension > 0.

A nonnegative matrix M is called *primitive* if there is an integer k such that $M^k > 0$. A primitive matrix is irreducible but the converse is not true.

A nonnegative matrix M is called *aperiodic* if the greatest common divisor of the integers k such that $M^k_{i,i} > 0$ for some i is equal to 1 (including the case where the set of integers k is empty). It can be verified that a matrix is primitive if and only if it is aperiodic and irreducible.

The Perron–Frobenius theorem asserts that for any nonnegative matrix M, the following holds

1. The matrix M has a real eigenvalue ρ_M such that $|\lambda| \leq \rho_M$ for any eigenvalue λ of M.
2. If $M \leq N$ with $M \neq N$, then $\rho_M < \rho_N$.
3. There corresponds to ρ_M a nonnegative eigenvector v and ρ_M is the only eigenvalue with a nonnegative eigenvector.
4. If M is irreducible, the eigenvalue ρ_M is simple and there corresponds to ρ_M a positive eigenvector v.
5. If M is primitive, all other eigenvalues have modulus strictly less than ρ_M. Moreover, $(1/\rho^n_M)M^n$ converges to a matrix of the form vw, where v (w) is a right (left) eigenvector corresponding to ρ, that is $Mv = \rho v$ ($wM = \rho w$) and $wv = 1$.

We shall give a sketch of a proof of this classical theorem. Let us first show that one may suppose that M is irreducible. Indeed, if M is reducible, we may consider a triangular decomposition as in Equation (1.7.1). Applying by induction the theorem to U and W, we obtain the result with ρ_M equal to the maximal value of the moduli of eigenvalues of U and W. The corresponding eigenvector is completed with zeroes (and thus condition 4 fails to hold).

We suppose from now on that M is irreducible. For a nonnegative Q-vector v, let

$$r_M(v) = \min\{(Mv)_i/v_i \mid 1 \leq i \leq n, v_i \neq 0\}$$

Thus $r_M(v)$ is the largest real number r such that $Mv \geq rv$. The function r_M is known as the *Wielandt* function. One has $r_M(\lambda v) = r_M(v)$ for all real number $\lambda \geq 0$. Moreover, r_M is continuous on the set of nonnegative vectors.

The set X of nonnegative vectors v such that $\|v\| = 1$ is compact. Since a continuous function on a compact set reaches its maximum on this set, there is an $x \in X$ such that $r_M(x) = \rho_M$ where $\rho_M = \max\{r_M(w) \mid w \in X\}$. Since $r_M(v) = r_M(\lambda v)$ for $\lambda \geq 0$, we have $\rho_M = \max\{r_M(w) \mid w \geq 0\}$.

We show that $Mx = \rho_M x$. By the definition of the function r_M, we have $Mx \geq \rho_M x$.

Set $y = Mx - \rho_M x$. Then $y \geq 0$. Assume $Mx \neq \rho_M x$. Then $y \neq 0$. Since $(I + M)^n > 0$, this implies that the vector $(I + M)^n y$ is positive. But

$$\begin{aligned}(I + M)^n y &= (I + M)^n(Mx - \rho_M x) = M(I + M)^n x - \rho_M(I + M)^n x \\ &= Mz - \rho_M z,\end{aligned}$$

with $z = (I + M)^n x$. This shows that $Mz > \rho_M z$, which implies that $r_M(z) > \rho_M$, a contradiction with the definition of r_M. This shows that ρ_M is an eigenvalue with a nonnegative eigenvector.

Let us show that $\rho_M \geq |\lambda|$ for each real or complex eigenvalue λ of M. Indeed, let v be an eigenvector corresponding to λ. Then $Mv = \lambda v$. Let $|v|$ be the nonnegative vector with coordinates $|v_i|$. Then $M|v| \geq |z||v|$ by the triangular inequality. By the definition of the Wielandt function, this implies $r_M(|v|) \geq |\lambda|$ and consequently $\rho_M \geq |\lambda|$. This completes the proof of assertion 1.

We have already seen that there corresponds to ρ_M a nonnegative eigenvector x. Let us now verify that $x > 0$. But this is easy since $(I + M)^n x = (1 + \rho_M)^n x$, which implies that $(1 + \rho_M)^n x > 0$ and thus $x > 0$.

In order to prove assertion 2, let us consider N such that $M \leq N$. Then obviously $\rho_M \leq \rho_N$. Let us show that $\rho_M = \rho_N$ implies $M = N$. Let $v > 0$ be such that $Mv = \rho_M v$. Then $Nv \geq \rho_M v$ and we conclude as before that $Nv = \rho_M v$. From $Mv = Nv$ with $v > 0$, we conclude that $M = N$ as asserted.

We now complete the proof of assertion 3. Let $Mv = \lambda v$ with $v \geq 0$. Since, as before, $(I + M)^n v = (1 + \lambda)^n v$, we actually have $v > 0$. Let D be the diagonal matrix with coefficients $v_1, v_2, \ldots, v_n$ and let $N = D^{-1}MD$. Since $n_{i,j} = m_{i,j}v_j/v_i$, we have $\sum_j b_{i,j} = \lambda$ for $1 \leq i \leq n$. Let w be a nonnegative eigenvector of N for the eigenvalue ρ_M. We normalize w in such a way that $w_i \leq 1$ for all i and $w_t = 1$ for one index t. Then,

$\rho_M = \sum_j n_{t,j} w_j \leq \sum_j n_{t,j} = \lambda$. Thus $\lambda = \rho_M$ as asserted. This completes the proof of assertion 3.

We further have to prove that ρ_M is simple. Let $p(\lambda) = \det(\lambda I - M)$ be the characteristic polynomial of M. We have $p'(\lambda) = \sum_i \det(\lambda I - M_i)$ where M_i is the matrix obtained from M by replacing the ith row and column by 0. Indeed,

$$\det(\lambda I - M) = \det(\lambda e_1 - v_1, \ldots, \lambda e_n - v_n)$$

where e_i is the ith unit vector and v_i is the ith column of M. Since the determinant is a multilinear function, this gives the desired formula for $p'(\lambda)$. One has $M_i \leq M$ and $M_i \neq M$ for each i, because an irreducible matrix cannot have a null row. By assertion 2, $\rho_{M_i} < \rho_M$ and thus $\det(\rho_M I - M_i) > 0$, whence $p'(\rho_M) > 0$. This shows that the root ρ_M is simple.

Let us finally prove assertion 5. Let λ be an eigenvalue of M such that $|\lambda| = \rho_M$. Let v be an eigenvector for the eigenvalue λ. Then, from $Mv = \rho_M v$, we obtain $M|v| \geq \rho_M |v|$ whence $M|v| = \rho_M |v|$ by the same argument as before. Let k be such that $M^k > 0$. Then, from $|M^k v| = M^k |v|$, we deduce that v is collinear to a nonnegative real vector. This shows that λ is real and thus that $\lambda = \rho_M$.

Since M has a simple eigenvalue ρ_M strictly greater than every other eigenvalue, the sequence $(1/\rho_M^n)M^n$ converges to a matrix of rank one, which is thus of the indicated form.

This completes the proof of the Perron–Frobenius theorem. For an indication of another proof, see Problem 1.7.1.

The practical computation of the maximal eigenvalue of a primitive matrix M can be done using the following algorithm. It is based on the fact that by assertion 5 the sequence defined by $x^{(n+1)} = (1/r(x^{(n)}))Mx^{(n)}$ converges to an eigenvector corresponding to the maximal eigenvalue, and thus $r(x^{(n)})$ converges to an eigenvector. The starting value $x^{(0)}$ can be an arbitrary positive vector.

DominantEigenvalue(M, x)

```
1 y ← x
2 do   (y, x) ← (Mx, y)
3      r ← min_{1≤i≤n} y_i/x_i
4      y ← (1/r)y
5 while y ≉ x
6 return r
```

where $y \approx x$ means that y is numerically close to x.

The vector computed by this algorithm is called an *approximate eigenvector*. The definition is the following. Let M be a nonnegative matrix. Let

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

r be such that $r \leq \rho_M$. Then a vector v such that $Mv \geq rv$ is called an approximate eigenvector relative to r.

Example 1.7.3. Let

$$M = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}.$$

The matrix M is nonnegative and irreducible. The eigenvalues of M are $\varphi = (1 + \sqrt{5})/2$ and $\hat{\varphi} = (1 - \sqrt{5})/2$. The vector $x = \begin{bmatrix} \varphi \\ 1 \end{bmatrix}$ is an eigenvector relative to φ. The vector $v = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ is an approximate eigenvector relative to $r = 1$ and Mv is an approximate eigenvector relative to $r = 3/2$.

1.8. Probability distributions on words

In this section, we consider the result of randomly selecting the letters composing a word. We begin with the formal definition of a probability law ruling this selection.

1.8.1. Information sources

Given an alphabet $\mathcal{A}$, a *probability distribution* on the set of words on $\mathcal{A}$ is a function $\pi : \mathcal{A}^* \to [0, 1]$ such that $\pi(\varepsilon) = 1$ and for each word $x \in \mathcal{A}^*$,

$$\sum_{a \in \mathcal{A}} \pi(xa) = \pi(x).$$

The definition implies that $\sum_{x \in \mathcal{A}^n} \pi(x) = 1$ for all $n \geq 0$. Thus a probability distribution on words does not make the set of all words a probability space but it does for each set $\mathcal{A}^n$.

Probability distributions on words are sometimes defined with a different vocabulary. One considers a sequence of random variables $(X_1, X_2, \ldots, X_n, \ldots)$ with values in the set $\mathcal{A}$. Such a sequence is often called a discrete time information *source* or also a stochastic *process*. For $x = a_1 \cdots a_n$ with $a_i \in \mathcal{A}$, set

$$\pi(x) = \mathbf{P}(X_1 = a_1, \ldots, X_n = a_n)$$

Then π is a probability distribution in the previous sense. Conversely, if π is a probability distribution, this formula defines the nth order joint distribution of the sequence $(X_1, X_2, \ldots, X_n, \ldots)$. We will say that $\mathbf{P}$ and π *correspond* to each other.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

Two particular cases are worth mentioning: Bernoulli distributions and Markov chains.

First, a *Bernoulli distribution* corresponds to successively independent choices of the symbols in a word, with a fixed distribution on letters. Thus it is given by a probability distribution on the set $\mathcal{A}$ extended by simple multiplication. For $a_1, a_2, \ldots, a_n \in \mathcal{A}$, one has $\pi(a_1 a_2 \cdots a_n) = \pi(a_1)\pi(a_2)\cdots\pi(a_n)$. In the terminology of information sources, a Bernoulli distribution corresponds to a sequence of independent, identically distributed (i.i.d.) random variables.

For example, if the alphabet has two letters a and b with probabilities $\pi(a) = p$ and $\pi(b) = q = 1 - p$, then $\pi(w) = p^{|w|_a} q^{|w|_b}$. The random variable X whose value is the number of letters b in a word w of length n has the distribution

$$\mathbf{P}(X = m) = \binom{n}{m} p^{n-m} q^m$$

This distribution is called the *binomial distribution*. Its expectation and variance are

$$\mathbf{E}(X) = np, \qquad \mathrm{Var}(X) = npq\,.$$

Second, a *Markov chain* corresponds to the case where the probability of choosing a symbol depends on the previous choice, but not on earlier choices. Thus, a Markov chain is given by an initial distribution π on $\mathcal{A}$ and by an $\mathcal{A} \times \mathcal{A}$ stochastic matrix P of conditional probabilities $P(a, b)$, that is such that for all $a \in \mathcal{A}$, $\sum_{b\in\mathcal{A}} P(a, b) = 1$. Then

$$\pi(a_1 a_2 \cdots a_n) = \pi(a_1) P(a_1, a_2) \cdots P(a_{n-1}, a_n).$$

In terms of stochastic processes, $P(a, b)$ is the conditional probability given for all $n \geq 2$ by $P(a, b) = \mathbf{P}(X_n = b \mid X_{n-1} = a)$. The powers of the matrix P can be used to compute the probability $\mathbf{P}(X_n = a)$. Indeed, one has $\mathbf{P}(X_n = a) = (\pi P^n)(a)$.

Example 1.8.1. Consider the Markov chain over $\mathcal{A} = \{a, b\}$ given by the matrix

$$\begin{bmatrix} 1/2 & 1/2 \\ 1 & 0 \end{bmatrix}$$

and the initial distribution $\pi(a) = \pi(b) = 1/2$. For example, one has $\pi(aab) = \pi(aaba) = 1/8$. This distribution assigns probability 0 to any word containing two consecutive letters b because $P(b, b) = 0$.

A distribution π on $\mathcal{A}^*$ is said to be *stationary* if for all $x \in \mathcal{A}^*$, one has $\pi(x) = \sum_{a\in\mathcal{A}} \pi(ax)$. In terms of stochastic processes, this means that the

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

joint distribution does not depend on the choice of time origin, that is,

$$\mathbf{P}(X_i = a_i, m \leq i \leq n) = \mathbf{P}(X_{i+1} = a_i, m \leq i \leq n).$$

A Bernoulli distribution is a stationary distribution. A Markov chain is stationary if and only if $\pi P = \pi$, that is if π is an eigenvector of the matrix P for the eigenvalue 1. The distribution π on $\mathcal{A}$ is itself called stationary.

A Markov chain is *irreducible* if, for all $a, b \in \mathcal{A}$, there exists an integer $n \geq 0$ such that $P^n(a, b) > 0$. This is exactly the definition of an irreducible matrix.

Similarly, a Markov chain is *aperiodic* if the matrix P is aperiodic.

The fundamental theorem of Markov chains says that for any irreducible Markov chain, there is a unique stationary distribution π, and whatever be the initial distribution, $\mathbf{P}(X_n = a)$ tends to $\pi(a)$. The proof uses the Perron–Frobenius theorem.

Example 1.8.2. Consider the Markov chain over $\mathcal{A} = \{a, b\}$ given by the same matrix

$$P = \begin{bmatrix} 1/2 & 1/2 \\ 1 & 0 \end{bmatrix}$$

and the initial distribution $\pi(a) = 2/3$, and $\pi(b) = 1/3$. This Markov chain is irreducible, and π is its unique stationary distribution.

A Markov chain is actually a particular case of a more general concept which is a probability distribution on words given by a finite automaton. Let $\mathfrak{A} = (Q, \mathcal{A})$ be a finite deterministic automaton. Let π be a probability distribution on Q. For each state $q \in Q$, consider a probability distribution on the set of edges starting in q. This is again denoted by π. Thus

$$\sum_{q \in Q} \pi(q) = 1, \qquad \sum_{a \in \mathcal{A}} \pi(q, a) = 1 \text{ for all } q \in Q.$$

This defines a probability distribution on the set of paths in $\mathfrak{A}$: given a path $\gamma : q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} \cdots$, we set $\pi(\gamma) = \pi(q_0)\pi(q_0, a_0)\pi(q_1, a_1) \cdots$. This in turn defines a probability distribution on the set of words as follows: for a word w, $\pi(w)$ is the sum of $\pi(\gamma)$ over all paths (γ) with label w. The probability on words obtained in this way is a transfer of a Markov chain on the edges of the automaton.

We now give two examples of probability distributions on words. The first one is a distribution given by a finite automaton, the second one is more general.

Example 1.8.3. Consider the automaton given in Figure 1.48. Let $\pi(1) = 1$, $\pi(2) = \pi(3) = 0$, and let $\pi(1, a, 2) = \pi(1, b, 2) = 1/2$, $\pi(2, a, 2) =$

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

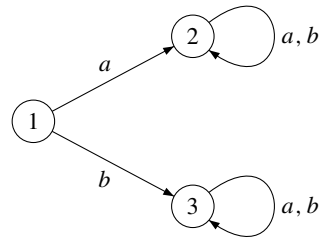


Figure 1.48. A finite automaton.

$\pi(2, b, 2) = 1/2$, and $\pi(3, a, 3) = 0$, $\pi(2, b, 2) = 1$. The probability π induced on words by this distribution on the automaton is such that $\pi(b^n) = 1/2$ for any word $n \geq 1$. This distribution keeps an unbounded memory of the past, and is therefore not a Markov distribution.

Example 1.8.4. Let $t = abbabaabbaababba \cdots$ be the Thue–Morse word that is the fixed point of the morphism μ which maps a to ab and b to ba. Let $\mathcal{S}$ be the set of factors of t. Define a function δ on words in $\mathcal{S}$ of length at least 4 as follows. For $w \in \mathcal{S}$ with $|w| \geq 4$, set $\delta(w) = v$, where v is the unique word of $\mathcal{S}$ such that

$$\mu(v) = \begin{cases} w \quad \text{or} \quad xwx & \text{if } |w| \text{ is even,} \\ wx \quad \text{or} \quad xw & \text{otherwise,} \end{cases}$$

for some $x \in \{a, b\}$.

For $w \in \mathcal{S}$, define $\pi(w)$ recursively by $\pi(w) = \pi(\delta(w))/2$ if $|w| \geq 4$, and by the value given in Figure 1.49 otherwise. It is easy to verify that π is an invariant probability distribution on $\mathcal{S}$. Indeed, one has $\pi(\varepsilon) = 1$ and, for each $w \in \mathcal{S}$,

$$\pi(wa) + \pi(wb) = \pi(w), \quad \pi(aw) + \pi(bw) = \pi(w).$$

The notion of a probability distribution on words leads naturally to the definition of a probability measure on the set $\mathcal{A}^\omega$ of infinite words. This produces a real probability distribution, instead of the distribution on each set $\mathcal{A}^n$.

Let $\mathfrak{C}$ be the set of thin cylinders, that is $\mathfrak{C} = \{w\mathcal{A}^\omega \mid w \in \mathcal{A}^*\}$, and let Σ be the σ-algebra generated by $\mathfrak{C}$. Recall that the σ-algebra generated by $\mathfrak{C}$ is the smallest family of sets containing $\mathfrak{C}$ and closed under complements and countable unions. A function μ from a σ-algebra Σ to the real numbers is said to be σ-*additive* if

$$\mu\Big(\bigcup_n E_n\Big) = \sum_n \mu(E_n)$$

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

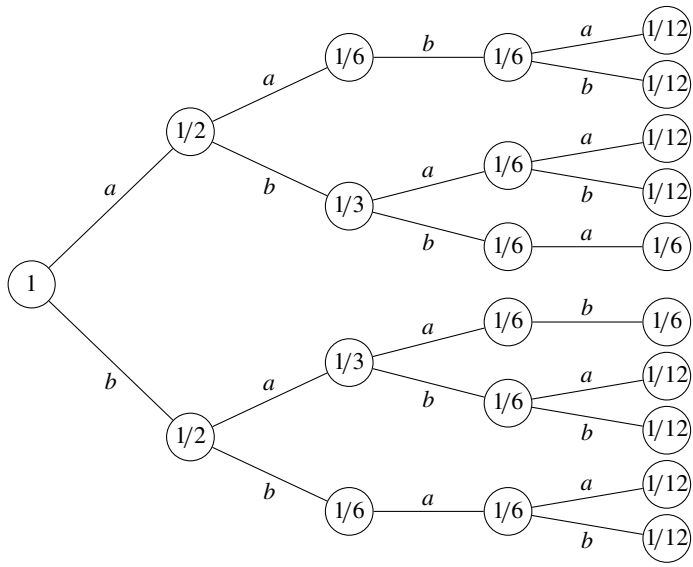


Figure 1.49. A probability distribution on the factors of the Thue–Morse infinite word.

for any family E_n of pairwise disjoint sets from Σ. A *probability measure* μ on $(\mathcal{A}^\omega, \Sigma)$ is a real valued function on Σ such that $\mu(\mathcal{A}^\omega) = 1$ and which is σ-additive.

By a classical theorem due to Kolmogorov, for each probability distribution π there exists a unique probability measure μ on $(\mathcal{A}^\omega, \Sigma)$ such that $\mu(x\mathcal{A}^\omega) = \pi(x)$.

Example 1.8.5. Let us consider again the distribution π on $\mathcal{A}^*$ with $\mathcal{A} = \{a, b\}$ of Example 1.8.3. The corresponding probability measure μ on $\mathcal{A}^\omega$ is such that $\mu(b^\omega) = 1/2$. Indeed, because $\mathcal{A}^\omega = \cup_{i\geq 0} b^i a\mathcal{A}^\omega \cup b^\omega$, one has by the property of σ-additivity

$$\mu(b^\omega) = \mu(\mathcal{A}^\omega) - \sum_{i\geq 0} \mu(b^i a\mathcal{A}^\omega) = 1 - \sum_{i\geq 0} \pi(b^i a) = 1 - \pi(a) = 1/2.$$

1.8.2. Entropy

Let U be a finite set, and let X be a random variable with values in U. Set $p(u) = \mathbf{P}(X = u)$. We define the *entropy* of X as

$$H(X) = -\sum_{u\in U} p(u) \log p(u).$$

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

We use the convention that $0 \log 0 = 0$. We also use the convention that the logarithm is taken in base 2. In this way, when the set U has two elements 0 and 1, with $p(0) = p(1) = 1/2$, then $H(X) = 1$. More generally, if U has n elements, then

$$H(X) \leq \log n \tag{1.8.1}$$

and the equality $H(X) = \log n$ holds if and only if $p(u) = 1/n$ for $u \in U$.

To prove this statement, we first establish the following assertion: Let p_i, q_i, for $(1 \leq i \leq n)$ be two finite probability distributions with $p_i, q_i > 0$. Then

$$\sum p_i \log p_i \geq \sum p_i \log q_i \tag{1.8.2}$$

with equality if and only if $p_i = q_i$ for $i = 1, \ldots, n$.

Indeed, observe first that $\log_e(x) \leq x - 1$ for $0 < x$ with equality if and only if $x = 1$. Thus for $1 \leq i \leq n$

$$\log_e(q_i/p_i) \leq q_i/p_i - 1$$

and consequently

$$\sum p_i \log_e(q_i/p_i) \leq \sum q_i - 1 = 0$$

This shows Inequality (1.8.2) for the logarithm in base e. Multiplying by an appropriate constant gives the general inequality. Equality holds if and only if $p_i = q_i$ for all i.

If we choose $q_i = 1/n$ for all i, Inequality (1.8.2) becomes Inequality (1.8.1).

If (X, Y) is a two-dimensional random variable with values in $U \times V$, we set $p(u, v) = \mathbf{P}(X = u, Y = v)$. Thus

$$H(X, Y) = - \sum_{(u,v) \in U \times V} p(u, v) \log p(u, v)$$

Finally, set $p(u|v) = \mathbf{P}(X = u | Y = v)$. Then we first define

$$H(X|v) = - \sum_{u \in U} p(u|v) \log p(u|v)$$

and for two random variables X, Y, we set

$$H(X|Y) = \sum_{v \in V} H(X|v) p(v).$$

It is easy to check that

$$H(X|Y) = - \sum_{(u,v) \in U \times V} p(u, v) \log p(u|v).$$

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

It can be checked that

$$H(X, Y) = H(Y) + H(X|Y). \tag{1.8.3}$$

Indeed

$$\begin{aligned} H(X, Y) &= -\sum_{u,v} p(u, v) \log p(u, v) \\ &= -\sum_{u,v} p(u, v) \log(p(u|v)p(v)) \\ &= -\sum_{u,v} p(u, v) \log p(u|v) - \sum_{u,v} p(u, v) \log p(v) \\ &= H(X|Y) + H(Y). \end{aligned}$$

It can also be verified that

$$H(X, Y) \leq H(X) + H(Y) \tag{1.8.4}$$

and that the equality holds if and only if X and Y are independent. Indeed,

$$\begin{aligned} H(X) + H(Y) &= -\sum_{u} p(u) \log p(u) - \sum_{v} p(v) \log p(v) \\ &= -\sum_{u,v} p(u, v) \log p(u) - \sum_{u,v} p(u, v) \log p(v) \\ &= -\sum_{u,v} p(u, v) \log(p(u)p(v)) \\ &\geq -\sum_{u,v} p(u, v) \log p(u, v) \end{aligned}$$

where the last inequality follows from Inequality (1.8.2).

More generally, if $(X_1, \ldots, X_n)$ is an information source, then $H_n = H(X_1, \ldots, X_n)$ is defined as the entropy of the random variable $(X_1, \ldots, X_n)$. In terms of a probability distribution π, we have

$$H_n = -\sum_{x \in \mathcal{A}^n} \pi(x) \log \pi(x).$$

Thus H_n is the entropy of the finite probability space $U = \mathcal{A}^n$.

Assume now that the source $(X_1, \ldots, X_n, \ldots)$ is stationary. The entropy of the source is defined as

$$H = \lim_{n \to \infty} \frac{1}{n} H_n.$$

According to the context, we write indistinctly H or $H(X)$. We show that this limit exists. First, observe that, by Inequality (1.8.4), for all $m, n \geq 1$

$$H(X_1, \ldots, X_{m+n}) \leq H(X_1, \ldots, X_m) + H(X_{m+1}, \ldots, X_{m+n}).$$

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

Since the source is stationary, $H(X_{m+1}, \ldots, X_{m+n}) = H(X_1, \ldots, X_n)$. This implies

$$H_{m+n} \leq H_m + H_n.$$

Thus the sequence (H_n) is a subadditive sequence of positive numbers. This implies that H_n/n has a limit.

We now give expressions for the entropy of particular sources. The entropy of a Bernoulli distribution π is

$$H = -\sum_{a \in \mathcal{A}} \pi(a) \log(\pi(a)).$$

Indeed, $H_n = nH_1$ since the random variables $X_1, \ldots, X_n$ are independent and identical. As a particular case, if $\pi(a) = 1/q$ for all $a \in \mathcal{A}$, then $H = \log q$.

The entropy of an irreducible Markov chain with matrix P and stationary distribution π is

$$H = \sum_{a \in \mathcal{A}} \pi(a) H^a$$

where $H^a = -\sum_{b \in \mathcal{A}} P(a, b) \log P(a, b)$. Indeed, by Formula (1.8.3), one has

$$H(X_1, \ldots, X_n) = H(X_1) + \sum_{k=1}^{n-1} H(X_{k+1} \mid X_k).$$

By definition,

$$H(X_{n+1} \mid X_n) = \sum_{a \in \mathcal{A}} H(X_{n+1} \mid X_n = a) \mathbf{P}(X_n = a)$$

and $H(X_{n+1} \mid X_n = a) = H^a$. Since $\mathbf{P}(X_n = a)$ tends to $\pi(a)$, $H(X_{n+1} \mid X_n)$ tends to $\sum_{a \in \mathcal{A}} H^a \pi(a)$. This implies

$$\lim \frac{1}{n} H(X_1, \ldots, X_n) = \sum_{a \in \mathcal{A}} H^a \pi(a).$$

Example 1.8.6. Consider again the Markov chain given by

$$P = \begin{bmatrix} 1/2 & 1/2 \\ 1 & 0 \end{bmatrix}$$

and the initial distribution $\pi(a) = 2/3$, and $\pi(b) = 1/3$. Then $H^a = 1$, $H^b = 0$ and $H = 2/3$.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

1.8.3. Topological entropy

For a set $\mathcal{S}$ of words, one defines the *topological entropy* of $\mathcal{S}$ as the limit

$$h(\mathcal{S}) = \limsup \frac{1}{n} \log s_n$$

where s_n is the number of words of length n in $\mathcal{S}$. This entropy is called the topological entropy to distinguish it from the entropy previously defined.

Let π be a stationary distribution. Let $\mathcal{S}$ be the *support* of π, that is, the set of words $x \in \mathcal{A}^*$ such that $\pi(x) > 0$. Then

$$H(\pi) \leq h(\mathcal{S}).$$

Indeed, in view of Inequality (1.8.1), one has for each $n \geq 1$,

$$H_n \leq \log s_n$$

where s_n is the number of words of length n in $\mathcal{S}$. The inequality follows by taking the limit. Thus the topological entropy of $\mathcal{S}$ is an upper bound to the value of possible entropies related to a stationary probability distribution supported by $\mathcal{S}$.

In the case of a regular set $\mathcal{S}$, the entropy $h(\mathcal{S})$ can be easily computed using the Perron–Frobenius theorem. Indeed, let $\mathfrak{A}$ be a deterministic automaton recognizing $\mathcal{S}$, and let M be the adjacency matrix of the underlying graph. By the Perron–Frobenius theorem, there is a real positive eigenvalue λ which is the maximum of the moduli of all eigenvalues. One has the formula

$$h(\mathcal{S}) = \log \lambda.$$

This formula expresses the fact that the number s_n of words of length n in $\mathcal{S}$ grows as λ^n.

Example 1.8.7. Consider again the Golden mean automaton of Example 1.3.5 which we redraw in Figure 1.50 for convenience. It recognizes the set $\mathcal{S}$ of words without two consecutive letters a. We have

$$M = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

and $\lambda = (1 + \sqrt{5})/2$. Thus $h(\mathcal{S}) = \log(1 + \sqrt{5})/2$.

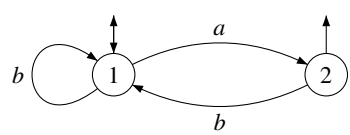


Figure 1.50. The Golden mean automaton.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

1.8.4. Distribution of maximal entropy

As we have seen in the previous section, the topological entropy of a set $\mathcal{S}$ is an upper bound to the value of possible entropies related to a stationary probability distribution supported by $\mathcal{S}$. A probability distribution π supported by $\mathcal{S}$ such that $H(\pi) = h(\mathcal{S})$ is called a *distribution of maximal entropy*. Intuitively, a distribution of maximal entropy on a set of words $\mathcal{S}$ is such that all words of $\mathcal{S}$ of a given length have approximately the same probability.

We are going to show that for each rational set $\mathcal{S}$, there exists a distribution π supported by $\mathcal{S}$ of maximal entropy, that is such that $H(\pi) = h(\mathcal{S})$.

We consider a deterministic automaton $\mathfrak{A}$ recognizing $\mathcal{S}$. Let G be the underlying graph of $\mathfrak{A}$ with labels removed. We assume that G is strongly connected. Let Q be the set of vertices of G and let M be its adjacency matrix. The fact that G is strongly connected is equivalent to the irreducibility of M.

By the Perron–Frobenius theorem, an irreducible matrix M has a real positive simple eigenvalue λ larger than or equal to the modulus of any other eigenvalue.

The number of paths of length n in G is asymptotically equivalent to λ^n. We prove that there is a labelling of G by positive real numbers which results in a Markov chain on Q of entropy $\log \lambda$. This produces a probability distribution exactly supported by S, which has maximal entropy $\log \lambda$.

Again by the Perron–Frobenius theorem, there exist a right eigenvector v and a left eigenvector w for the eigenvalue λ such that all v_i and w_i are strictly positive. We normalize v and w such that $v \cdot w = 1$. Let

$$P_{ij} = (v_j/\lambda v_i)m_{i,j}$$

and let $\pi_i = v_i w_i$. The matrix P is stochastic since

$$\sum_j P_{i,j} = \sum_j (v_j/\lambda v_i)m_{i,j} = \sum_j v_j m_{i,j}/\lambda v_i = 1.$$

The Markov chain with transition matrix P and initial distribution π is stationary. Indeed,

$$\sum_i \pi_i P_{i,j} = \sum_i v_i w_i (v_j/\lambda v_i)m_{i,j} = (v_j/\lambda)\sum_i w_i m_{i,j} = (v_j/\lambda)\lambda w_j = \pi_j.$$

The entropy of the Markov chain is $\log \lambda$. Indeed, the probability of any path γ of length n from i to j is

$$p(\gamma) = \frac{w_i v_j}{\lambda^n}.$$

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

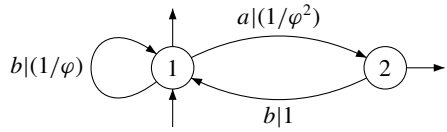


Figure 1.51. The Golden mean automaton with transition probabilities.

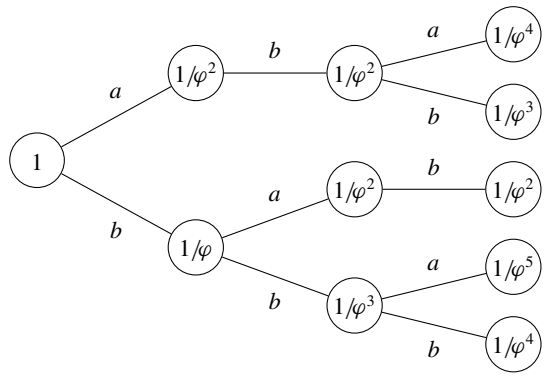


Figure 1.52. The tree of the Golden mean.

This proves the existence of a distribution with maximal entropy on $\mathcal{S}$ when the graph of the automaton is strongly connected. In this case, the uniqueness can also be proved. The existence in the general case can be shown to reduce to this one.

Example 1.8.8. Let us consider again the Golden mean automaton of Example 1.3.5 which recognizes the set $\mathcal{S}$ of words without aa. We have

$$M = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}, \quad w = \begin{bmatrix} \varphi & 1 \end{bmatrix}, \quad (1+\varphi^2)v = \begin{bmatrix} \varphi \\ 1 \end{bmatrix},$$

$$P = \begin{bmatrix} 1/\varphi & 1/\varphi^2 \\ 1 & 0 \end{bmatrix}, \quad \pi = \begin{bmatrix} \varphi^2/(1+\varphi^2) & 1/(1+\varphi^2) \end{bmatrix}.$$

The values of the transition probabilities are represented on the automaton of Figure 1.51. The probability distribution on words induced by this Markov chain is pictured in Figure 1.52.

As a consequence of the above construction, the distribution of maximal entropy associated with a rational set $\mathcal{S}$ is given by a finite automaton. It is even more remarkable that it can be given by the *same* automaton as the set $\mathcal{S}$ itself. This appears clearly in the above example where the automaton of Figure 1.51 is the same as the Golden mean automaton of Figure 1.11.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

1.8.5. Ergodic sources and compressions

Consider a source $X = (X_1, X_2, \ldots, X_n, \ldots)$ on the alphabet $\mathcal{A}$ associated to a probability distribution π. Given a word $w = a_1 \cdots a_n$ on $\mathcal{A}$, denote by $f_N(w)$ the frequency of occurrences of the word w in the first N terms of the sequence X.

We say that the source X is *ergodic* if for any word w, the sequence $f_N(w)$ tends almost surely to $\pi(w)$. An ergodic source is stationary. The converse is not true, as shown by the following example.

Example 1.8.9. Let us consider again the distribution of Example 1.8.3. This distribution is stationary. We have $f_N(b) = 1$ when the source outputs only letters b, although the probability of b is $1/2$. Thus, this source is not ergodic.

Example 1.8.10. Consider the distribution of Example 1.8.4. This source is ergodic. Indeed, the definition of π implies that the frequency $f_N(w)$ of any factor w in the Thue–Morse word tends to $\pi(w)$.

It can be proved that any Bernoulli source is ergodic. This implies in particular the statement known as the strong law of large numbers: if the sequence $X = (X_1, X_2, \ldots, X_n, \ldots)$ is independent and identically distributed then, setting $S_n = X_1 + \cdots + X_n$, the sequence $(1/n)S_n$ converges almost surely to the common value $\mathbf{E}(X_i)$.

More generally, any irreducible Markov chain equipped with its stationary distribution as initial distribution is an ergodic source.

Ergodic sources have the important property that typical messages of the same length have approximately the same probability, which is 2^{-nH} where H is the entropy of the source. Let us give a more precise formulation of this property, known as the *asymptotic equirepartition property*. Let $(X_1, X_2, \ldots)$ be an ergodic source with entropy H. Then for any $\epsilon > 0$ there is an N such that for all $n \geq N$, the set of words of length n is the union of two sets $\mathcal{R}$ and $\mathcal{T}$ satisfying

(i) $\pi(\mathcal{R}) < \epsilon$

(ii) for each $w \in \mathcal{T}$,

$$2^{-n(H+\epsilon)} < \pi(w) < 2^{-n(H-\epsilon)}$$

where π denotes the probability distribution on $\mathcal{A}^n$ defined by $\pi(a_1 a_2 \cdots a_n) = \mathbf{P}(X_1 = a_1, \ldots, X_n = a_n)$. Thus, the set of messages of length n is partitioned into a set $\mathcal{R}$ of negligible probability and a set $\mathcal{T}$ of "typical" messages all having approximately probability 2^{-nH}.

Since $\pi(w) \geq 2^{-n(H+\epsilon)}$ for $w \in \mathcal{T}$, the number of typical messages satisfies $\mathrm{Card}(\mathcal{T}) \leq 2^{n(H+\epsilon)}$. This observation allows us to see that the entropy gives a lower bound for the compression of a text. Indeed, if

the messages of length n are coded unambiguously by binary messages of average length ℓ, then $\ell/n \geq H - \epsilon$ since otherwise two different messages would have the same coding. On the other hand, any coding assigning different binary words of length $n(H+\epsilon)$ to the typical messages and arbitrary values to the other messages will give a coding of compression rate approximately equal to H.

It is interesting in practice to have compression methods which are universal in the sense that they do not depend on a particular source. Some of these methods however achieve asymptotically the theoretical lower bound given by the entropy for all ergodic sources. We sketch here the presentation of one of these methods among many, the *Ziv–Lempel encoding algorithm*. This algorithm fits well in our selection of topics because it is combinatorial in nature.

We consider for a word w the factorization

$$w = x_1 x_2 \cdots x_m u$$

where

1. for each $i = 1, \ldots, m$, the word x_i is chosen to be the shortest possible not the set $\{x_0, x_1, x_2, \ldots, x_{i-1}\}$, with the convention $x_0 = \varepsilon$,
2. the word u is a prefix of some x_i.

This factorization is called the *Ziv–Lempel factorization* of w. It appears again in Chapter 8. For example, the Fibonacci word has the factorization

$$(a)(b)(aa)(ba)(baa)(baab)(ab)(aab)(aba)\cdots.$$

The coding of the word w is the sequence $(n_1, a_1), (n_2, a_2), \ldots, (n_m, a_m)$ where $n_1 = 0$ and $x_1 = a_1$, and for each $i = 2, \ldots, n$, we have $x_i = x_{n_i} a_i$, with $n_i < i$ and a_i a letter. Writing each integer n_i in binary gives a coding of length approximately $m \log m$ bits. It can be shown that for any ergodic source, the quantity $m \log m/n$ tends almost surely to the entropy of the source. Thus this coding is an optimal universal coding.

Practically, the coding of a word w uses a set D called the *dictionary* to maintain the set of words $\{x_1, \ldots, x_i\}$. We use a trie (see Section 1.3.1) to represent the set D. We also suppose that the word ends with a final symbol to avoid coding the last factor u.

```
ZLencoding(w)
1 ▷ returns the Ziv–Lempel encoding c of w
2 T ← NewTrie()
3 (c, i) ← (ε, 0)
4 while i < |w| do
5       (ℓ, p) ← LongestPrefixInTrie(w, i)
6       a ← w[i + ℓ]
```

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

```
 7      q ← NewVertex()
 8      Next(p, a) ← q          ▷ updates the trie T
 9      c ← c · (p, a)          ▷ appends (p, a) to c
10      i ← i + ℓ + 1
11  return c
```

The result is a linear time algorithm. The decoding is also simple. The important point is that there is no need to transmit the dictionary. Indeed, one builds it in the same way as it was built in the encoding phase. It is convenient this time to represent the dictionary as an array of strings.

```
ZLdecoding(c)
 1  (w, i) ← (ε, 0)
 2  D[i] ← ε
 3  while c ≠ ε do
 4      (p, a) ← Current()      ▷ returns the current pair in c
 5      Advance()
 6      y ← D[p]
 7      i ← i + 1
 8      D[i] ← ya               ▷ adds ya to the dictionary
 9      w ← wya
10  return w
```

The functions Current() and Advance() manage the sequence c, considering each pair as a token. The practical details of the implementation are delicate. In particular, it is advised not to let the size of the dictionary grow too much. One strategy is to limit the size of the input, encoding it in blocks. Another one resets the dictionary once it has exceeded some prescribed size. In either case, the decoding algorithm must of course also follow the same strategy.

1.8.6. Unique ergodicity

We have seen that in some cases, given a formal language $\mathcal{S}$, there exists a unique invariant measure with entropy equal to the topological entropy of the set $\mathcal{S}$. In particular, it is true in the case of a regular set $\mathcal{S}$ recognized by an automaton with a strongly connected graph. In this case, the measure is also ergodic since it is the invariant measure corresponding to an irreducible Markov chain. There are even cases in which there is a unique invariant measure supported by $\mathcal{S}$. This is the so-called property of *unique ergodicity*. We will see below that this situation arises for the factors of fixed points of primitive morphisms.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

Example 1.8.4 is one illustration of this case. We obtained the result by an elementary computation. In the general case, one considers a morphism $f : \mathcal{A}^* \to \mathcal{A}^*$ that admits a fixed point $u \in \mathcal{A}^\omega$. Let M be the $\mathcal{A} \times \mathcal{A}$–matrix defined by

$$M_{a,b} = |f(a)|_b$$

where $|x|_a$ is the number of occurrences of the symbol a in the word x. We suppose the morphism f to be primitive, which by definition means that the matrix M itself is primitive. It is easy to verify that for any n, the entry $M^n_{a,b}$ is the number of occurrences of b in the word $f^n(a)$.

Since the matrix M associated to the morphism f is primitive it is also irreducible. By the Perron–Frobenius theorem, there is a unique real positive eigenvalue λ and a real positive eigenvector v such that $vM = \lambda v$. We normalize v by $\sum_{a\in\mathcal{A}} v_a = 1$.

Using the fact that M is primitive, again by the Perron–Frobenius theorem, $(1/\lambda^n)M^n_{a,b}$ tends to a matrix with rows proportional to v_b when n tends to ∞. This shows that the frequency of a symbol b in u is equal to v_b.

The value of the distribution of maximal entropy on the letters is given by $\pi(a) = v_a$. For words of length ℓ larger than 1, a similar computation can be carried out, provided that one passes to the alphabet of overlapping words of length ℓ, as shown in the following example.

Example 1.8.11. Let us consider again the set $\mathcal{S}$ of factors of the Thue–Morse infinite word t (Example 1.8.4). The matrix of the morphism $\mu : a \to ab, b \to ba$ is

$$M = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}.$$

The left eigenvector is $v = [1/2,\ 1/2]$ and the maximal eigenvalue is 2. Accordingly, the probability of the symbols is $\pi(a) = \pi(b) = 1/2$. To compute by this method the probability of the words of length 2, we replace the alphabet $\mathcal{A}$ by the alphabet $\mathcal{A}_2 = \{x, y, z, t\}$ with $x = aa$, $y = ab$, $z = ba$ and $t = bb$. We replace μ by the morphism μ_2 obtained by coding successively the overlapping blocks of length 2 appearing in $\mu(\mathcal{A}^2)$.

It is enough to truncate at length 2 in order to get a morphism that has as its unique fixed point the infinite word t_2 obtained by coding overlapping blocks of length 2 in t. Thus

$$\mu_2 : \begin{array}{l} x \mapsto yz \\ y \mapsto yt \\ z \mapsto zx \\ t \mapsto zy \end{array}$$

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

has the fixed point

$$t_2 = ytzyzxytzxyz\cdots.$$

The matrix associated with μ_2 is

$$M^{(2)} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}.$$

The left eigenvector is $v_2 = [1/6, 1/3, 1/3, 1/6]$, consistently with the values of π given in Figure 1.49.

1.8.7. Practical estimate of the entropy

The entropy of a source given by an experiment and not by an abstract model (for example like a Markov chain) can be usefully estimated. This occurs in practice in the context of natural languages or for sources producing signals recorded by some physical measure.

The case of natural languages is of practical interest for the purpose of text compression. An estimate of the entropy H of a natural language like English implies for example that an optimal compression algorithm can encode using H bits per character on average. The definition of a quantity which can be called "entropy of English" deserves some commentary. First we have to clarify the nature of the sequences considered. A reasonable simplification is to assume that the alphabet is composed of the 26 ordinary letters (and thus without the upper/lower case distinction) plus possibly a blank character to separate words. The second convention is of a different nature. If one wants to consider a natural language as an information source, an assumption has to be made about the nature of the source. The good approximation obtained by finite automata for the description of natural languages makes it reasonable to assume that a natural language such as English can be considered as an irreducible Markov chain and thus as an ergodic source. Thus it makes sense to estimate the probabilities by the frequencies observed on a text or a corpus of texts and to use these approximations to estimate the entropy H by $H \approx H_n/n$ where

$$H_n = -\sum_k p_k \log p_k$$

and where the p_k are the probabilities of the n-grams. One actually has $H \leq H_n/n$. It is worth remarking that the approximation thus obtained is much better than that obtained by using $H \approx h_n/n$ with

$$h_n = \log s_n$$

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

Table 1.2. Entropies of n-grams on an alphabet of 26 or 27 letters.

number of symbols	26	27
H_1	4.14	4.03
$H_2/2$	3.56	3.32
$H_3/3$	3.30	3.10

where s_n is the number of possible n-grams in correct English sentences. For small n the approximation is bad because some n-grams are far more frequent than others, and for large n the computation is not feasible because the number of correct sentences is too large.

One has $H \leq \log_2(26) \approx 4.7$ when considering only 26 symbols and $H \leq \log_2(27) \approx 4.76$ on 27 symbols. Further values are given in Table 1.2 leading to an upper bound $H \leq 3$. An algorithm to compute the frequencies of n-grams is easy to implement. It uses a buffer s which is initialized to the first n symbols of the text and which is updated by shifting the symbols one place to the left and adding the current symbol of the text at the last place. This is done by the function CURRENT(). The algorithm maintains a set S of n-grams together with a map FREQ() containing the frequencies of each n-gram. A practical implementation should use a representation of sets like a hashtable, allowing the set to be stored in a space proportional to the size of S (and not to the number of all possible n-grams, which grows too fast).

```
ENTROPY(n)
 1 ▷ returns the n-th order entropy H_n
 2 S ← ∅                          ▷ S is the set of n-grams in the text
 3 do  s ← CURRENT()              ▷ s is the current n-gram of the text
 4     if s ∉ S then
 5         S ← S ∪ s
 6         FREQ(s) ← 1
 7     else FREQ(s) ← FREQ(s) + 1
 8 while there are more symbols
 9 for s ∈ S do
10     PROB(s) ← FREQ(s)/Card S
11 return (1/n) Σ_{s∈S} PROB(s) log PROB(s)
```

Another approach leads to a better estimate of H. It is based on an experiment which uses a human being as an oracle. The idea is to scan a text through a window of $n-1$ consecutive characters and to ask a subject

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

Table 1.3. Experimental bounds for the entropy of English.

n	1	2	3	4	5	6	7
upper bound	4.0	3.4	3.0	2.6	2.1	1.9	1.3
lower bound	3.2	2.5	2.1	1.8	1.2	1.1	0.6

to guess the symbol following the window contents, repeating the question until the answer is correct. The average number of probes is an estimate of the conditional entropy $H(X_n|X_1, \ldots, X_{n-1})$. The values obtained are shown in Table 1.3.

1.9. Statistics on words

In this section, we consider the problem of computing the probability of appearance of some properties on words defined using the concepts introduced at the beginning of the chapter. In particular, we shall study the average number of factors or subwords of a given type in a regular set.

1.9.1. Occurrences of factors

For any integer valued random variable X with probability distribution $p_n = \mathbf{P}(X = n)$, one introduces the generating series $f(z) = \sum_{n\geq 0} p_n z^n$. If we denote $q_n = \sum_{m\geq n} p_m$, then the generating series $g(z) = \sum_{n\geq 0} q_n z^n$ is given by the formula

$$g(z) = \frac{1 - f(z)}{1 - z}.$$

This implies in particular that the expectation $\mathbf{E}(X) = \sum_{n\geq 0} np_n$ of X also has the expression $\mathbf{E}(X) = g(1)$. These general observations about random variables have an important interpretation when the random variable X is the length of a prefix in a given prefix code.

Let π be a probability distribution on $\mathcal{A}^*$. For a prefix code $\mathcal{C} \subset \mathcal{A}^*$, the value $\pi(\mathcal{C}) = \sum_{x\in\mathcal{C}} \pi(x)$ can be interpreted as the probability that a long enough word has a prefix in $\mathcal{C}$. Accordingly, we have $\pi(\mathcal{C}) \leq 1$.

Let $\mathcal{C}$ be a prefix code such that $\pi(\mathcal{C}) = 1$. The average length of the words of $\mathcal{C}$ is

$$\lambda(\mathcal{C}) = \sum_{x\in\mathcal{C}} |x|\pi(x).$$

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

One has the useful identity

$$\lambda(\mathcal{C}) = \pi(\mathcal{P})$$

where $\mathcal{P} = \mathcal{A}^* - \mathcal{C}\mathcal{A}^*$ is the set of words which do not have a prefix in $\mathcal{C}$. Indeed, let $p_n = \pi(\mathcal{C} \cap \mathcal{A}^n)$ and $q_n = \sum_{m \geq n} p_m$. Then, $\lambda(\mathcal{C}) = \sum_{n \geq 1} np_n = \sum_{n \geq 1} q_n$. Since $\pi(\mathcal{P} \cap \mathcal{A}^n) = q_n$, this proves the claim.

The generating series $C(z) = \sum_{n \geq 0} p_n z^n$ is related to $P(z) = \sum_{n \geq 0} q_n z^n$ by

$$C(z) - 1 = P(z)(1 - z).$$

When π is a Bernoulli distribution, one may use unambiguous expressions on sets to compute probability of events definable in this way. Indeed, the unambiguous operations translate to operations on probability generating series. If $\mathcal{W}$ is a set of words, we set

$$W(z) = \sum_{n \geq 0} \pi(\mathcal{W} \cap \mathcal{A}^n) z^n.$$

Then, if $\mathcal{U} + \mathcal{V}$, $\mathcal{U}\mathcal{V}$, and $\mathcal{U}^*$ are unambiguous expressions, we have

$$(U + V)(z) = U(z) + V(z), \quad (UV)(z) = U(z)V(z),$$
$$(U^*)(z) = \frac{1}{1 - U(z)}.$$

We now give two examples of this method.

Consider first the problem of finding the expected waiting time $T(w)$ before seeing a word w. We are going to show that it is given by the formula

$$T(w) = \frac{\pi(\mathcal{Q})}{\pi(w)} \tag{1.9.1}$$

where $\mathcal{Q} = \{q \in \mathcal{A}^* \mid wq \in \mathcal{A}^* w \text{ and } |q| < |w|\}$. Thus $\mathcal{Q}$ is the set of (possibly empty) words q such that $w = sq$ with s a nonempty suffix of w.

Let $\mathcal{C}$ be the prefix code formed of words that end with w for the first time. Let $\mathcal{V}$ be the set of prefixes of $\mathcal{C}$, which is also the set of words which do not contain w as a factor. We can write

$$\mathcal{V}w = \mathcal{C}\mathcal{Q}. \tag{1.9.2}$$

Moreover both sides of this equality are unambiguous. Thus, since $\pi(\mathcal{C}) = 1, \pi(\mathcal{V})\pi(w) = \pi(\mathcal{Q})$, whence Formula (1.9.1). Formula (1.9.2) can also be used to obtain an explicit expression for the generating series $C(z)$. Indeed, using (1.9.2), one obtains $V(z)\pi(w)z^m = C(z)Q(z)$, where m is the length

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

of w. Replacing $V(z)$ by $(1 - C(z))/(1 - z)$, one obtains

$$C(z) = \frac{\pi(w)z^m}{\pi(w)z^m + Q(z)(1 - z)} \tag{1.9.3}$$

The polynomial $Q(z)$ is called the *autocorrelation polynomial* of w. Its explicit expression is

$$Q(z) = 1 + \sum_{p \in P(w)} \pi(w_{n-p} \cdots w_{n-1})z^p$$

where $P(w)$ is the set of periods of the word $w = w_0 \cdots w_{n-1}$, and w_i denotes the ith letter of w. A slightly more general definition is given in Chapter 6.

Example 1.9.1. In the particular case of $w = a^m$ and $A = \{a, b\}$ with $\pi(a) = p$, $\pi(b) = q = 1 - p$, the autocorrelation polynomial of w is

$$Q(z) = \frac{1 - p^m z^m}{1 - pz}.$$

Consequently, $\pi(\mathcal{Q}) = (1 - p^m)/q$ and Formula (1.9.1) and (1.9.3) become

$$T(a^m) = \frac{1 - p^m}{qp^m}, \quad C(z) = \frac{(1 - pz)p^m z^m}{1 - z + qp^m z^{m+1}},$$

so that for $p = q = 1/2$,

$$T(w) = 2^{m+1} - 2, \quad C(z) = \frac{(1 - z/2)z^m/2^m}{1 - z + z^{m+1}/2^{m+1}}$$

Formula (1.9.1) can be considered as a paradox. Indeed, it asserts that with $\pi(a) = \pi(b) = 1/2$, the waiting time for the word $w = aa$ is 6 while it is 4 for $w = ab$.

Formula (1.9.1) is related to the automaton recognizing the words ending with w and consequently to Algorithm SEARCHFACTOR. We illustrate this on an example. Let $w = abaab$. The minimal automaton recognizing the words on $\{a, b\}$ ending with w for the first time is represented in Figure 1.53. The transitions of the automaton can actually be computed using the array b introduced in algorithm BORDER.

	0	1	2	3	4	5
b :	−1	0	0	1	1	2

For example, the transition from state 3 by letter b is to state 2 because $b[3] = 1$ and $w[1] = b$. The set Q can also be read on the array b. Actually, we have $Q = \{\varepsilon, aab\}$ since the border of w has length 2 ($b[5] = 2$) and $b[2] = 0$.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

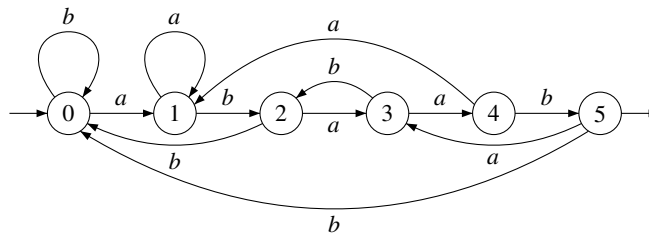


Figure 1.53. The minimal automaton recognizing the words ending with *abaab*.

As a second example, we now consider the problem of finding the probability f_n that the number of a equals the number of b for the first time in a word of length n on $\{a, b\}$ starting with a, with $\pi(a) = p, \pi(b) = q = 1 - p$. This is the classical problem of return to 0 in a random walk on the line.

The set of words starting with a and having as many as as bs for the first time is the Dyck set $\mathcal{D}$ already studied in Section 1.6. We have already seen that $\mathcal{D} = a\mathcal{D}^*b$. Thus, the generating series $D(z) = \sum_{n\geq 0} f_{2n}z^{2n}$ satisfies

$$D^2 - D + pqz^2 = 0.$$

$$D(z) = \frac{1 - \sqrt{1 - 4pqz^2}}{2}.$$

This formula shows in particular that for $p = q$, $\pi(\mathcal{D}) = 1/2$ since $\pi(\mathcal{D}) = D(1)$. But for $p \neq q$, $\pi(\mathcal{D}) < 1/2$. An elementary application of the binomial formula gives the coefficient f_n of $D(z) = \sum_{n\geq 0} f_n z^n$

$$f_{2n} = \frac{1}{n}\binom{2n-2}{n-1}p^n q^n .$$

1.9.2. Extremal problems

We consider here the problem of computing the average value of several maxima concerning words. We assume here that the source is Bernoulli, that is that the successive letters are drawn independently with a constant probability distribution π.

We begin with the case of the *longest run of successive occurrences of some letter* a with $\pi(a) = p$. The probability of seeing a run of k consecutive letters a beginning at some given position in a word of length n is p^k. So the average number of runs of length k is approximately np^k.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

Let K_n be the average value of the maximal length of a run of letters a in the words of length n.

Intuitively, since the longest run is likely to be unique, we have $np^{K_n} = 1$. This equation has the solution $K_n = \log_{1/p} n$. One can elaborate the above intuitive reasoning to prove that

$$\lim_{n\to\infty} \frac{K_n}{\log_{1/p} n} = 1\,. \tag{1.9.4}$$

This formula shows that, on average, the maximal length of a run of letters a is logarithmic in the length of the word.

A simple argument shows that the same result holds when runs are extended to be words over some fixed subset $\mathcal{B}$ of the alphabet $\mathcal{A}$. In this case, p is replaced by the sum of the probabilities of the letters in $\mathcal{B}$.

Another application of the above result is the computation of the average length of the longest common factor starting at the same position in two words of the same length. Such a factor x induces in two words w and w' the factorizations $w = uxv$ and $w' = u'xv'$ with $|u| = |u'|$. A factor is just a run of symbols (a, a) in the word (w, w') written over the alphabet of pairs of letters. The value of p for Equation (1.9.4) is

$$p = \sum_{a\in\mathcal{A}} \pi(a)^2\,.$$

The *average length of the longest repeated factor* in a word is also logarithmic in the length of the word. It is easily seen that over a q letter alphabet, the length k of the longest repeated factor is at least $\lfloor \log_q n \rfloor$ and thus the average length of the longest repeated factor is at least $\log_q n$. It can be proved that it is also $O(\log n)$.

The longest common factor of two words can be computed in linear time. An algorithm (LENGTHS-OF-FACTORS) is given in Chapter 2. The *average length of the longest common factor* of two words of the same length is also logarithmic in the length. More precisely, let C_n denote the average length of the longest common factor of two words of the same length n. Then

$$\lim_{n\to\infty} \frac{C_n}{\log_{1/p} n} = 2\,. \tag{1.9.5}$$

The intuitive argument used to derive Formula (1.9.4) can be adapted to this case to explain the value of the limit. Indeed, the the average number of common factors of length k in two words of length n is approximately $n^2 p^k$. Solving the equation $n^2 p^k = 1$ gives $k = \log_{1/p} n^2 = 2\log_{1/p} n$.

Table 1.4. Some upper and lower bounds for c_k.

k	lower bound	upper bound
2	0.76	0.86
3	0.61	0.77
10	0.39	0.54
15	0.32	0.46

The case of subwords contrasts with the case of factors. We have already given in Section 1.2.4 an algorithm (LCSLENGTHARRAY(x, y)) which computes the length of the longest common subwords of two words. The essential result concerning subwords is that the average length $c(k, n)$ of the longest common subwords of two words of length n on k symbols is $O(n)$. More precisely, there is a constant c_k such that

$$\lim_{n\to\infty} \frac{c(k,n)}{n} = c_k.$$

This result is easy to prove, even if the proof does not give a formula for c_k. Indeed, we have $c(k, n+m) \geq c(k, n) + c(k, m)$ since this inequality holds for the length of the longest common subwords of any pair of words. This implies that the sequence $c(k, n)/n$ converges (we have already met this argument in Section 1.8.2). There is no known formula for c_k but only estimates given in Table 1.4.

Problems

Section 1.1

1.1.1 Show that the number of words of length n on q letters with a given subword of length k is

$$\sum_{i=0}^{n-k} \binom{n-i-1}{k-1} q^i (q-1)^{n-i-k}.$$

In particular, this number does not depend on the particular word chosen as a subword. (Hint: consider the automaton recognizing the set of words having a given word as subword.)

1.1.2 Let $c : (A \cup \varepsilon) \times (A \cup \varepsilon) \to \mathbb{R} \cup \infty$ be a function assigning a *cost* to each pair of elements equal to a symbol or to the empty word. Assume that

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

(i) the restriction of c to $A \times A$ is a distance,
(ii) $c(\varepsilon, a) = c(a, \varepsilon) > 0$ for all $a \in A$.

Each transformation on a word is assigned a cost using the cost c as follows. A substitution of a symbol a by a symbol b adds a cost $c(a, b)$. An insertion of a symbol a counts for $c(\varepsilon, a)$ and a deletion for $c(a, \varepsilon)$. Let $d(u, v)$ be the distance defined as the minimal cost of a sequence of transformations that changes u into v.

Show that d is a distance on A^*. Show that d coincides with

1. the Hamming distance if $c(a, b) = 1$ for $a \neq b$ and $c(a, \varepsilon) = c(\varepsilon, a) = \infty$,
2. the subword distance if $c(a, b) = \infty$ for $a, b \in A$ and $a \neq b$, and $c(a, \varepsilon) = c(\varepsilon, a) = 1$ for all $a \in A$.

Section 1.2

1.2.1 The sharp border array of a word x of length m is the array sb of size $m + 1$ such that $sb[m] = b[m]$ and for $1 \leq j \leq m - 1$, $sb[j]$ is the largest integer i such that $x[0\,..\,i-1] = x[j-i\,..\,j-1]$ and $x[j] \neq x[i]$. By convention, $sb[j] = -1$ if no such integer i exists. For example, if $x = abaababa$, the array sb is

	0	1	2	3	4	5	6	7	8
b :	−1	0	−1	1	0	−1	3	−1	3

.

Show that the following variant of Algorithm BORDER computes the array sb in linear time.

```
BORDERSHARP(x)
 1 ▷ x has length m, sb has size m + 1
 2 i ← 0
 3 sb[0] ← −1
 4 for j ← 1 to m − 1 do
 5       ▷ Here x[0..i − 1] = border(x[0..j − 1])
 6       if x[j] = x[i] then
 7             sb[j] ← sb[i]
 8       else sb[j] ← i
 9             do   i ← sb[i]
10             while i ≥ 0 and x[j] ≠ x[i]
11             i ← b[i]
12       i ← i + 1
13 sb[m] ← i
14 return b
```

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

Show that, in Algorithm SEARCHFACTOR, one may use the table sb of sharp borders instead of the table b of borders, resulting in a faster algorithm.

Section 1.3

1.3.1 This exercise shows how to answer the following questions: what is the minimal Hamming distance between a word w and the words of a regular set X and how is a word of X computed which realizes the minimum?

These questions are solved by the following algorithm known as the *Viterbi algorithm*.

Let $\mathcal{A} = (Q, i, T)$ be a finite automaton over the alphabet A and, for each $p \in Q$, let X_p be the set recognized by the automaton (Q, i, p).

Let $w = a_0 \cdots a_{n-1}$ be a word of length n. For a symbol $a \in A$ and $0 \leq i < n$ we denote $c(a, i) = 0$ if $a = a_i$ and $c(a, i) = 1$ otherwise. We compute the function $d : Q \times \mathbb{N} \to \mathbb{N}$ defined by: $d(p, i)$ is the minimal Hamming distance of the words in $X_p \cap A^i$ to the word $a_0 \cdots a_{i-1}$.

```
VITERBI(w)
1 for i ← 0 to n − 1 do
2       for each edge (p, a, q) do
3             if d(p, i − 1) + c(a, i) < d(q, i) then
4                   d(q, i) ← d(p, i − 1) + c(a, i)
5 return min_{t∈T} d(t, n − 1)
```

Show how to modify this algorithm to return a word in X that is closest to w.

1.3.2 Prove that the minimal automaton recognizing the set $\mathcal{S}(w)$ of suffixes of a word of length n has at most $2n$ states. Hint: show that for any p, q, the sets $p^{-1}\mathcal{S}(w)$ and $q^{-1}\mathcal{S}(w)$ are either disjoint or comparable.

Section 1.5

1.5.1 Let $\mathfrak{A} = (Q, I, T)$ be a transducer over $\mathcal{A}$, $\mathcal{B}$ with n states. Let M be the maximal length of output labels in the edges of $\mathfrak{A}$. Suppose that $\mathfrak{A}$ is equivalent to a sequential transducer $\mathfrak{B}$, obtained by the determinization algorithm. Let $(u, q) \in B^* \times Q$ be a pair appearing in a state of $\mathfrak{B}$. Show that $|u| \leq 2n^2M$.

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

Section 1.7

1.7.1 A *rational function* is a function of the form $f(z) = \sum_{n\geq 0} a_n z^n$ such that $f(z)q(z) = p(z)$ for two polynomials p, q with $q(0) = 1$. It is said to be nonnegative if $a_n \geq 0$ for all $n \geq 0$. We shall use the fact that if $f(z)$ is a nonnegative rational function such that $f \neq 0$ and $f(0) = 0$, then the radius of convergence σ of $f^*(z) = 1/(1 - f(z))$ is a simple pole of f^* such that $\sigma \leq |\pi|$ for any other pole π (see the Notes of this chapter for a reference). Moreover σ is the unique real number such that $f(\sigma) = 1$.

Let M be an $n \times n$ irreducible matrix and let

$$M = \begin{bmatrix} u & v \\ w & N \end{bmatrix}$$

with N of dimension $n - 1$. Let $f(z) = uz + v(I - zN)^{-1}wz^2$. Show that

$$1/(1 - f(z)) = (I - Mz)^{-1}_{1,1}.$$

Use the result quoted above on nonnegative rational functions to prove that

1. the spectral radius ρ_M of M is $1/\sigma$ where σ is such that $f(\sigma) = 1$,
2. each row of the matrix $[(\sigma - z)(I - Mz)^{-1}]_{z=\sigma}$ is a positive eigenvector of M corresponding to $1/\sigma$.

(Hint: use the relation $I + (I - Mz)^{-1}Mz = (I - Mz)^{-1}$).

Section 1.8

1.8.1 Consider a uniform primitive morphism $f : \mathcal{A}^* \to \mathcal{A}^*$ with a fixpoint $u \in \mathcal{A}^\omega$. We indicate here a method to compute the frequency of the factors of length ℓ in u by a faster method than the one used in Section 1.8.6. Let F_ℓ be the set of factors of length ℓ of u. Let $M^{(\ell)}$ be the $F_\ell \times F_\ell$–matrix defined by $M^{(\ell)}_{x,y} = |f(x)|_y$ as in Example 1.8.11. Let p be an integer such that $|f^p(a)| > \ell - 2$ for all $a \in \mathcal{A}$. Let U be the $F_2 \times F_\ell$–matrix defined as follows. For $a, b \in \mathcal{A}$ such that $ab \in F_2$ and $y \in F_\ell$, $U_{ab,y}$ is the number of occurrences of y in $f^p(ab)$ that begin in the prefix $f^p(a)$. Show that

$$UM^{(\ell)} = M^{(2)}U,$$

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

that $M^{(2)}$ and $M^{(\ell)}$ have the same dominant eigenvalue ρ, and that if v_2 is an eigenvector of $M^{(2)}$ corresponding to ρ, then $v_\ell = v_2 U$ is an eigenvector of $M^{(\ell)}$ corresponding to ρ.

1.8.2 Let $\mu : a \to ab, b \to ba$ be the morphism with fixpoint the Thue–Morse word. Show that for $\ell = 5$, $p = 3$, the matrix U of the previous problem (with the 12 factors of length 5 of the Thue–Morse word listed in alphabetic order) is

$$U = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

and that the vector $v_2 U$ with $v_2 = \begin{bmatrix} 1 & 2 & 2 & 1 \end{bmatrix}$ is the vector with all components equal to 4. Deduce that the 12 factors of length 5 of the Thue–Morse word have the same frequency (see Example 1.8.4).

1.8.3 Consider the following transformation T on words: a word w is replaced by the word $T(w)$ of the same length obtained as follows: list the cyclic shifts of w in alphabetic order as the rows $w_1, w_2, \ldots, w_n$ of an array. Then $T(w)$ is the last column of the array. For example, let $w = abracadabra$. The list of conjugates of w sorted in alphabetical order is represented below.

	1	2	3	4	5	6	7	8	9	10	11
1	*a*	*a*	*b*	*r*	*a*	*c*	*a*	*d*	*a*	*b*	*r*
2	*a*	*b*	*r*	*a*	*a*	*b*	*r*	*a*	*c*	*a*	*d*
3	*a*	*b*	*r*	*a*	*c*	*a*	*d*	*a*	*b*	*r*	*a*
4	*a*	*c*	*a*	*d*	*a*	*b*	*r*	*a*	*a*	*b*	*r*
5	*a*	*d*	*a*	*b*	*r*	*a*	*a*	*b*	*r*	*a*	*c*
6	*b*	*r*	*a*	*a*	*b*	*r*	*a*	*c*	*a*	*d*	*a*
7	*b*	*r*	*a*	*c*	*a*	*d*	*a*	*b*	*r*	*a*	*a*
8	*c*	*a*	*d*	*a*	*b*	*r*	*a*	*a*	*b*	*r*	*a*
9	*d*	*a*	*b*	*r*	*a*	*a*	*b*	*r*	*a*	*c*	*a*
10	*r*	*a*	*a*	*b*	*r*	*a*	*c*	*a*	*d*	*a*	*b*
11	*r*	*a*	*c*	*a*	*d*	*a*	*b*	*r*	*a*	*a*	*b*

The word $T(w)$ is the last column of the array. Thus in our example $T(w) = rdarcaaaabb$. Show that $w \mapsto T(w)$ is a bijection up to conjugacy.

1.8.4 Let $u_{\mathcal{S}}(z)$ be the generating series of the number of words of length n in $\mathcal{S}$, that is

$$u_{\mathcal{S}}(z) = \sum_{w \in \mathcal{S}} z^{|w|} .$$

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

Show that

$$u_{\mathcal{S}}(z) = S(qz),$$

where $S(z)$ is the generating series for the uniform Bernoulli distribution on q symbols.

1.8.5 Show that the generating series of the set $\mathcal{F}$ of words over $\{a, b\}$ without factor $w = a^n$ is

$$u_{\mathcal{F}}(z) = \frac{1 - z^n}{1 - 2z + z^{n+1}}.$$

Notes

Several textbooks treat the subject of algorithms on words in much more detail than we did here. In the first place, several general textbooks on algorithms like Aho, Hopcroft, and Ullman (1975) or Sedgewick (1983) include automata and pattern matching algorithms among many other topics. In the second place, several books like Crochemore and Rytter (1994), Gusfield (1997) or Baeza-Yates and Ribero-Neto (1999) are entirely dedicated to word algorithms.

Words. The distance introduced in Problem 1.1.2 is known as the *edit distance* or also the *alignment distance*. It has been introduced first by Levenshtein (1965) and it is used in many ways in bioinformatics (see Sankoff and Kruskal 1983).

The algorithm VITERBI in Problem 1.3.1 is used in the context of convolutional error-correcting codes (see McEliece 2002). It appears again in Chapter 4.

Elementary algorithms. The algorithm BORDER computing the border of a word in linear time and the linear time algorithm SEARCHFACTOR that checks whether a word is a factor of another (Algorithm SEARCHFACTOR) are originally due to Knuth, Morris, and Pratt (1977). This algorithm is the first one of a large family of algorithms constituting the field of *pattern matching* algorithms. See Crochemore, Hancart, and Lecroq (2001) for a general presentation. The algorithm BORDERSHARP of Problem 1.2.1 is from Knuth *et al.* (1977).

The quadratic algorithm to compute a longest common subword (Algorithm LCS) is usually credited to Hirschberg (1977), although many authors discovered it independently (see Sankoff and Kruskal 1983). It is not known whether a linear algorithm exists. An algorithm working in time $O(p \log n)$ on two words of length n with p pairs of matching positions is due to Hunt and Szymanski (1977).

https://doi.org/10.1017/CBO9781107341005.002 Published online by Cambridge University Press

The linear algorithm CIRCULARMIN that computes the least conjugate of a word is due to Booth (1980). Several refinements were proposed, see Shiloach (1981), Duval (1983), Apostolico and Crochemore (1991). The algorithm giving the factorization in Lyndon words (Algorithm LYNDONFACTORIZATION) is due to Fredricksen and Maiorana (1978), see also Duval (1983).

Tries and automata. Tries are treated in many textbooks on algorithms (e.g., Aho, Hopcroft, and Ullman 1983). Our treatment of the implementation of automata and pattern matching is also similar to that of most textbooks (see for example Aho *et al.* 1975).

The exact complexity of the minimization problem for deterministic finite automata is not yet known. Moore's algorithm appears in a historical paper (Moore 1956). Hopcroft's minimization algorithm appears first in Hopcroft (1971). The linear minimization algorithm for DAWGs is from Revuz (1992). It can be considered as an extension of the tree isomorphism algorithm in Aho *et al.* (1975).

Gilbreath's card trick (Example 1.3.9) is described as follows in Chapter 9 of Gardner (1966): *Consider a deck of* $2n$ *cards ordered in such a way that red and black cards alternate. Cut the deck into two parts and give it a riffle shuffle. Cut it once more, this time not completely arbitrarily but at a place where two cards of the same colour meet. Square up the deck.*

Then for every $i = 1, \ldots, n$ *the pair consisting of the* $(2i-1)$*-th and the* $2i$*-th card is of the form* (red, black) *or* (black, red). The property of binary sequences underlying the card trick is slightly less general than Formula (1.3.1).

The source of Exercise (1.3.2) is Blumer, Blumer, Haussler, Ehrenfeucht, Chen, and Seiferas (1985). The automaton can be used in several contexts, including as a transducer called the *suffix transducer* (see Chapter 2).

Pattern Matching. The equivalence of regular expressions and finite automata is a classical result known as *Kleene's theorem*. We present here only one direction of this result, namely the construction of finite automata from regular expressions. This transformation is used in many situations. Actually, regular expressions are often used as a specification of some pattern and the equivalent finite automaton can be considered as an implementation of this specification. The converse transformation gives rise to algorithms that are less frequently used. One case of use is for the computation of generating series (see Section 1.9).

There is basically only one method to transform a regular expression into a finite automaton which operates by induction on the structure of the

regular expression. However, several variants exist. The one presented here is due to Thompson (1968). It uses ε-transitions and produces a normalized automaton that has a unique initial state with no edge entering it and a unique terminal state with no edge going out of it. Another variant produces an automaton without ε-transitions (see Eilenberg 1974 for example). The resulting automaton has in general fewer states than the one obtained by Thomson's algorithm. Yet another variant produces directly a deterministic automaton (see Berry and Sethi 1986 or Aho, Sethi, and Ullman 1986).

Transducers. The notion of a rational relation and of a transducer dates back to the origins of automata theory although there are few books treating extensively this aspect of the theory. Eilenberg's book (Eilenberg 1974) represents a significant date in the clarification of the concepts and notation. Later books treating transducers include Berstel (1979) and Sakarovich (2004). A word on the terminology concerning what we call here sequential transducers. The term "sequential machine" (also called "Mealy machine") is in general used only in the case of sequential letter-to-letter transducers. The version using (possibly empty) word outputs is often called a "generalized sequential machine" (or gsm). A further generalization, used by Schützenberger (1977), introduces a class of tranducers called *subsequential* which allow the additional use of a terminal suffix. We simply call here *sequential* these subsequential transducers.

Any sequential function is a rational function but the converse is of course not true. Several characterizations of rational functions, in particular special classes of transducers realize rational fuctions. Among these are so-called *bimachines* used in Chapter 3. The determinization algorithm has been first studied in Schützenberger (1977) and Choffrut (1979). In particular, the characterization of sequential transducers by the twinning property is in Choffrut (1977, 1979). See also Reutenauer (1990).

The source of Problem 1.5.1 is Béal and Carton (2002). It can be checked in polynomial time whether a transducer is equivalent to a sequential one (see Weber and Klemm 1995 or Béal and Carton 2002). The normalization algorithm was first considered by Choffrut (1979) and subsequently by Mohri (1994) and by Béal and Carton (2001).

A quite different algorithm relying on shortest paths algorithms has been proposed by Breslauer (1998). For recent developments, see the survey on minimization algorithms of transducers in Choffrut (2003).

Parsing. The section on parsing follows essentially Aho *et al.* (1986). The Dyck language is named after the group theorist Walther von Dyck (Dyck 1882). Context-free grammars are an important model for modelling hierarchically structured data. They appear in various equivalent forms, such

as recursive transition networks (RTNs) used in natural language processing (see Chapter 3). Parsers are ubiquitous in data processing systems, including natural language processing. The abstract model of a parser is a pushdown automaton which is a particular case of the general model of a Turing machine.

It is a remarkable fact that a large class of grammars can be parsed in one pass, from left to right, and in linear time. This was first established by Knuth (1965) who introduced in particular the *LR* analysis described here.

Word enumeration. A detailed proof of the Perron–Frobenius theorem can be found in Gantmacher (1959). The proof given here is due to Wielandt, whence the name of "Wielandt function", also called Collatz–Wielandt function (see Allouche and Shallit (2003)).

Problem 1.7.1 presents the connection between the theorem of Perron–Frobenius and related statements concerning the poles of nonnegative rational functions. For a proof of the statement appearing at the beginning of the problem, see Eilenberg (1974) or Berstel and Reutenauer (1984). This approach gives a proof of the Perron–Frobenius theorem that differs from the one given in Section 1.7.2. See McCluer (2000) for a survey on several possible proofs of this theorem.

Probability. Our presentation of probability distributions on words is inspired by Welsh (1988), Szpankowski (2001), and Shields (1969). A proof of the fundamental theorem of Markov chains can be found in most textbooks on probability theory (see, e.g., Feller (1968), chapter XV). The theorem of Kolmogorov on probability measures on infinite words can be found in Feller (1971), chapter IV. The notion of entropy is due to Shannon (1948). Many textbooks contain a presentation of the main properties of entropy (see e.g. Ash 1990). The computation of the distribution of maximal entropy (Section 1.8.4) is originally due to Shannon. Our presentation follows Lind and Marcus (1996), chapter 13.

The computation of the frequencies of factors in fixpoints of substitutions is reproduced from Queffélec (1987). The method described in Problem 1.8.1 is also from Queffélec (1987).

The asymptotic equirepartition property of ergodic sources is known as the Shannon–McMillan theorem. See Shields (1969) for a proof. The Ziv–Lempel coding originally appears in Ziv and Lempel (1977). A complete presentation of this popular coding can be found in Bell, Cleary, and Witten (1990) with several variants.

The entropy of English has been studied by Shannon. In particular, Tables 1.2 and 1.3 are from Shannon (1951). They are reproduced in several manuals on text compression (see e.g. Welsh 1988 or Bell *et al.* 1990).

Statistics on words. Events defined by prefix codes (Section 1.9.1) are presented in Feller (1968) under the name of recurrent events. Formula (1.9.1) appears already in the paper Schützenberger (1964). The name of autocorrelation polynomials appears in Guibas and Odlyzko (1981b). Formula (1.9.4) is due to Erdös Rényi (1970). For a proof, see Waterman (1995), chapter 11. Formula (1.9.5) is due to Arratia, Morris, and Waterman (1988) (see Waterman (1995), chapter 11). Table 1.4 is from Sankoff and Kruskal (1983). It has been proved recently that $C_k\sqrt{k} \to 2$ when $k \to +\infty$ (Kiwi *et al.* 2004).

The transformation described in Problem 1.8.3 is known as the Burrows–Wheeler transformation (see Manzini (2001)). It is the basis of a text compression method. Indeed, the idea is that adjacent rows of the table of cyclic shifts will often begin by a long common prefix and $T(w)$ will therefore have long runs of identical symbols. For example, in a text in English, most rows beginning with 'nd' will end with 'a'.