CHAPTER 2

Structures for Indexes

2.0. Introduction

The chapter presents data structures used to memorize the suffixes of a text and some of their applications. These structures are designed to give a fast access to all factors of the text, and this is the reason why they have a fairly large number of applications in text processing.

Two types of objects are considered in this chapter, digital trees and automata, together with their compact versions. Trees put together common prefixes of the words in the set. Automata gather in addition their common suffixes. The structures are presented in order of decreasing size.

The representation of all the suffixes of a word by an ordinary digital tree called a suffix trie (Section 2.1) has the advantage of being simple but can lead to a memory size that is quadratic in the length of the considered word. The compact tree of suffixes (Section 2.2) is guaranteed to hold in linear memory space.

The minimization (related to automata) of the suffix trie gives the minimal automaton accepting the suffixes and is described in Section 2.4. Compaction and minimization yield the compact suffix automaton of Section 2.5.

Most algorithms that build the structures presented in this chapter work in time $O(n \times \log \operatorname{Card} \mathcal{A})$, for a text of length n, assuming that there is an ordering on the alphabet $\mathcal{A}$. Their execution time is thus linear when the alphabet is finite and fixed. Locating a word of length m in the text then takes $O(m \times \log \operatorname{Card} \mathcal{A})$ time.

The main application of presented techniques is to provide the basis for implementing indexes, which is described in Section 2.6. But the direct access to factors of a word authorizes a great number of other applications.

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

We briefly mention how to detect repetitions or forbidden words in a text (Section 2.7). Structures can also be used to search for fixed patterns in texts because they can be regarded as pattern matching machines (see Section 2.8). This method is extended in a particularly effective way, for searching conjugates (or rotations) of a pattern, in Section 2.8.3.

2.1. Suffix trie

The tree of suffixes of a word, called its *suffix trie*, is a deterministic automaton that accepts the suffixes of the word and in which there is a unique path from the initial state to any state. It can be viewed as a digital tree which represents the set of suffixes of the word. Standard methods can be used to implement these automata, but its tree structure authorizes a simplified representation.

Considering a tree implies that the terminal states of the tree are in one-to-one correspondence with the words of the accepted language. The tree is thus finite only if its language is also finite. Consequently, the explicit representation of the tree has an algorithmic interest only for finite languages.

Sometimes one forces the tries to have, for terminal states, only the external nodes of the tree. With this constraint, a language $\mathcal{L}$ is representable by a trie only if no proper prefix of a word of $\mathcal{L}$ is in $\mathcal{L}$. It results from this remark that if y is a nonempty word, only $\mathrm{Suff}(y) \setminus \{\varepsilon\}$ is representable by a trie having this property, and this takes place only when the last letter of y appears only once in y. For this reason one frequently adds for this purpose a marker at the end of the word. We prefer to attach an output to the nodes of the tree, which is in conformity with the concept used, that of automaton. Only the nodes whose output is defined are regarded as terminals. In addition, there are only very slight differences between the implementations of the two features.

The suffix trie of a word y is denoted by $\mathfrak{T}(y)$. Its nodes are the factors of y, ε is the initial state, and the suffixes of y are the terminal states. The transition function δ of $\mathfrak{T}(y)$ is defined by $\delta(u, a) = ua$ if ua is a factor of y and $a \in \mathcal{A}$. The output of a terminal state, which is then a suffix, is the position of this suffix in y. An example of a suffix trie is displayed in Figure 2.1.

A classical construction of $\mathfrak{T}(y)$ is carried out by adding successive suffixes of y in the tree under construction, from the longest suffix, y itself, until the shortest, the empty word.

The current operation inserts $y[i \,..\, n-1]$, the suffix at position i, in the structure which already contains all the longer suffixes. It is illustrated by Figure 2.2. We call the *head* of a suffix its longest prefix common to a suffix

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

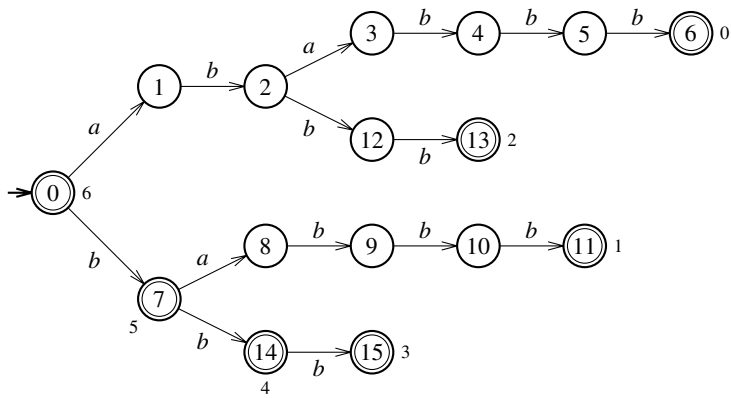


Figure 2.1. Trie $\mathfrak{T}(ababbb)$ of suffixes of *ababbb*. Terminal states are marked by double circles. The output associated with a terminal state is the position of the corresponding suffix on the word *ababbb*. The empty suffix, by convention, is associated with the length of the word.

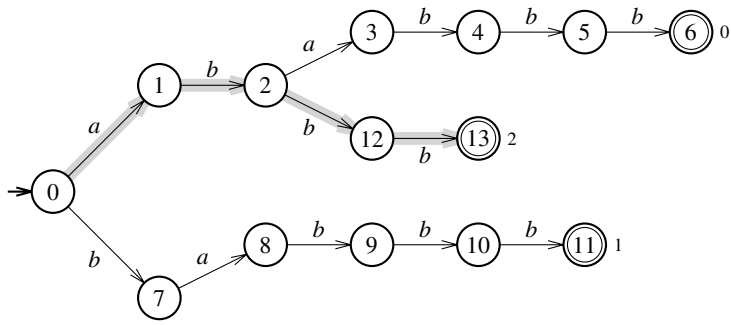


Figure 2.2. The trie $\mathfrak{T}(ababbb)$ (see Figure 2.1) during its construction, just after the insertion of suffix *abbb*. The fork, state 2, corresponds to the head, *ab*, of the suffix. It is the longest prefix of *abbb* that appears before the concerned position. The tail of the suffix is *bb*, the label of the path grafted at this stage from the fork and leading to states 12 and 13.

occurring at a smaller position. It is also the longest prefix of $y[i \,..\, n-1]$ that is the label of some path starting at the initial state of the automaton in construction. The target state of the path is called a *fork* (two divergent paths start from this state). If $y[i \,..\, k-1]$ is the head of the suffix at position i ($y[i \,..\, n-1]$) the word $y[k \,..\, n-1]$ is called the *tail* of the suffix.

More precisely, one calls fork any state of the automaton which is of (out-degree) degree at least 2, or which is both of degree 1 and terminal.

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

Algorithm SUFFIXTRIE builds the suffix trie of y.Its code is given below. It is supposed that the automaton is represented by lists of successors (adjacency lists). The list associated with state p is denoted by $adj[p]$ and contains pairs of the form (a, q) where a is a letter and q a state. The function TARGET implements transitions of the automaton, so TARGET(p, a) is q when $(a, q) \in adj[p]$ (or more generally when $(au, q) \in adj[p]$ for some word u, as considered in the next sections). States of the automaton have the attribute *output* whose value is a position. When creating a state, the procedure NEWSTATE allocates an empty adjacency list and set as undefined the value of the attribute *output*. Only the output of terminal states is set by the algorithm. The procedure NEWAUTOMATON creates a new automaton, say M, with only one state, its initial state $initial(M)$.

In the algorithm, the insertion of the current suffix $y[i\,..\,n-1]$ in the automaton M under construction, starts with the computation of its head, $y[i\,..\,k-1]$, and of the associated fork, $p = \delta(initial(M), y[i\,..\,k-1])$, from which is grafted the tail of the suffix (denoting by δ the transition function of M). The value of the function SLOWFIND applied to the pair $(initial(M), i)$ is precisely the sought pair (p, k). The creation of the path of label $y[k\,..\,n-1]$ from p together with the definition of the output of its target is carried out at lines 5–9.

The last step of the execution, insertion of the empty suffix, just defines the output of the initial state, which value is $n = |y|$ by convention (line 10).

```
SUFFIXTRIE(y, n)
 1  M ← NEWAUTOMATON()
 2  for i ← 0 to n − 1 do
 3        (fork, k) ← SLOWFIND(initial(M), i)
 4        p ← fork
 5        for j ← k to n − 1 do
 6              q ← NEWSTATE()
 7              adj[p] ← adj[p] ∪ {(y[j], q)}
 8              p ← q
 9        output[p] ← i
10  output[initial(M)] ← n
11  return M
```

```
SLOWFIND(p, i)
1  for k ← i to n − 1 do
2        if TARGET(p, y[k]) is undefined then
3              return (p, k)
4        p ← TARGET(p, y[k])
5  return (p, n)
```

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

Proposition 2.1.1. *Algorithm* SUFFIXTRIE *builds the suffix trie of a word of length n in time $\Theta(n^2)$.*

Proof. The correctness is easy to check on the code of the algorithm.

For the evaluation of execution time, let us consider stage i. Let us suppose that $y[i\,..\,n-1]$ has head $y[i\,..\,k-1]$ and has tail $y[k\,..\,n-1]$. The call to SLOWFIND (line 3) performs $k-i$ operations and the **for** loop at lines 5–8 does $n-k$ ones, which gives a total of $n-i$ operations. Thus the **for** loop indexed by i at lines 2–9 executes $n+(n-1)+\cdots+1$ operations, which gives a total execution time of $\Theta(n^2)$. ∎

2.1.1. Suffix links

It is possible to accelerate the preceding construction by improving the search for forks. The technique described here is used in the following section where it leads to an actual gain in the execution time that is measurable with the asymptotic evaluation.

Let av be a suffix of y with a nonempty head az ($a \in \mathcal{A}$). The prefix z of v thus appears in y before the considered occurrence. This implies that z is a prefix of the head of suffix v. The search for this head and the corresponding fork can thus be done by starting in state z instead of starting systematically with the initial state as done in the preceding algorithm. However, this supposes that, the state az being known, one has a fast access to state z. For that, one introduces a function defined on the states of the automaton and called the *suffix link function*. It is denoted by s_y and defined, for each state az ($a \in \mathcal{A}$, $z \in \mathcal{A}^*$), by $s_y(az) = z$. State z is called the *suffix link* of state az. Figure 2.3 displays in dashed arrows the suffix link function of the trie of Figure 2.1.

The algorithm SLOWFIND-BIS uses the suffix link function for the computation of the suffix trie of y. The function is implemented by a table named $s\ell$. Suffix links are actually computed there only for the forks and their ancestors, except for the initial state. The rest is just an adaptation of algorithm SLOWFIND that includes the definition of the suffix link table $s\ell$. The new algorithm is called SLOWFIND-BIS.

```
SLOWFIND-BIS(p, k)
1  while k < n and TARGET(p, y[k]) is defined do
2        q ← TARGET(p, y[k])
3        (e, f) ← (p, q)
4        while e ≠ initial(M) and sℓ[f] is undefined do
5              sℓ[f] ← TARGET(sℓ[e], y[k])
6              (e, f) ← (sℓ[e], sℓ[f])
```

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

```
 7        if sℓ[f] is undefined then
 8              sℓ[f] ← initial(M)
 9        (p, k) ← (q, k + 1)
10  return (p, k)
```

Algorithm SUFFIXTRIE-BIS is an adaptation of SUFFIXTRIE. It uses the function SLOWFIND-BIS instead of SLOWFIND.

```
SUFFIXTRIE-BIS(y, n)
 1  M ← NEWAUTOMATON()
 2  sℓ[initial(M)] ← initial(M)
 3  (fork, k) ← (initial(M), 0)
 4  for i ← 0 to n − 1 do
 5        k ← max{k, i}
 6        (fork, k) ← SLOWFIND-BIS(sℓ[fork], k)
 7        p ← fork
 8        for j ← k to n − 1 do
 9              q ← NEWSTATE()
10              adj[p] ← adj[p] ∪ {(y[j], q)}
11              p ← q
12        output[p] ← i
13  output[initial(M)] ← n
14  return M
```

Proposition 2.1.2. *Algorithm* SUFFIXTRIE-BIS *builds the suffix trie of y in time $\Theta(\text{Card } Q)$, where Q is the set of states of $\mathfrak{T}(y)$.*

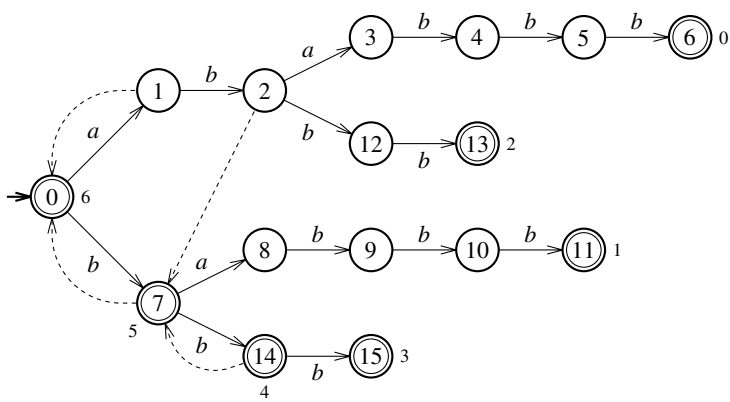


Figure 2.3. The trie $\mathfrak{T}(ababbb)$ with suffix links of forks and of their ancestors indicated by dashed arrows.

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

Proof. The operations of the main loop, apart from line 6 and the **for** loop at lines 8–11, are carried out in constant time, which gives a time $O(n)$ for their total execution.

Each operation of the internal loop of the algorithm SLOWFIND-BIS, which is called at line 6, leads to the creation of a suffix link. The total number of links being bounded by Card Q, the cumulated time of all the executions of line 6, is $O(\text{Card } Q)$.

The execution time of the loop 8–11 is proportional to the number of states that it creates. The cumulated time of all the executions of lines 8–11 is thus still $O(\text{Card } Q)$.

Consequently, the total time of the construction is $\Theta(\text{Card } Q)$, which is the announced result. ∎

The size of $\mathfrak{T}(y)$ can be quadratic. This is the case for example for a word whose letters are pairwise distinct. For this category of words algorithm SUFFIXTRIE-BIS is in fact not faster than SUFFIXTRIE.

For certain words, it is enough to prune the hanging branches (below the forks) of $\mathfrak{T}(y)$ to obtain a structure of linear size. This kind of pruning gives the tree called the position tree of y (see Figure 2.4), which represents the shortest factors occurring only once in y or the suffixes that identify other positions. However, considering the position tree does not completely solve the question of memory space for the structure that can still have a quadratic size. It can be checked for example that the word $a^k b^k a^k b^k$ ($k \in \mathbb{N}$) of length $4k$ has a pruned suffix trie that contains more k^2 nodes.

The structure of the compact tree of the following section is a solution to obtaining a structure of linear size. The automata of Sections 2.4 and 2.5 provide another type of solution.

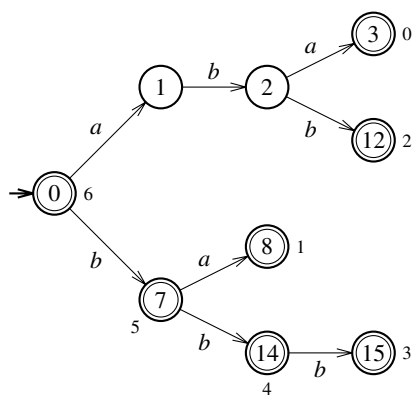


Figure 2.4. Position tree of *ababbb*. It accepts the shortest factors which identify positions on the word, and some suffixes.

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

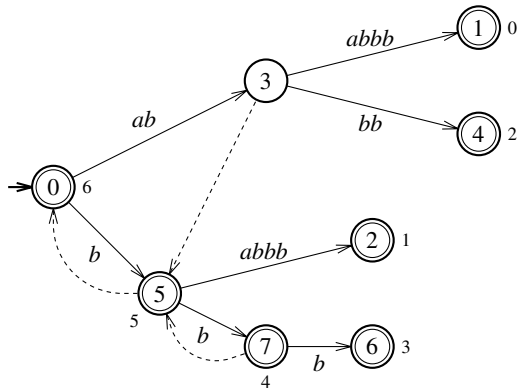


Figure 2.5. The (compact) suffix tree $\mathcal{S}(ababbb)$ with its suffix links.

2.2. Suffix tree

The compact suffix trie of word y, simply called its *suffix tree* and denoted by $\mathcal{S}(y)$, is obtained by removing the nodes of degree 1 which are not terminal in its suffix trie. This operation is called the compaction of the trie. The compact tree preserves only the forks and the terminal nodes of the suffix trie. Labels of edges then become words of positive variable length. Observe that if two edges start from a same node and are labelled by the words u and v then the first letters of these words are distinct, that is $u[0] \neq v[0]$. This comes from the fact that the suffix trie is a deterministic automaton.

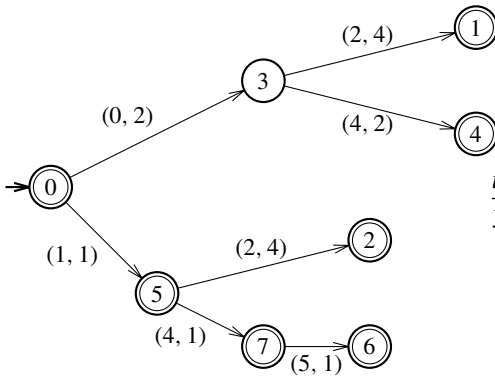
Figure 2.5 shows the compact suffix tree obtained by compaction of the suffix trie of Figure 2.1.

Proposition 2.2.1. *The compact suffix tree of a word of length $n > 0$ has between $n + 1$ and $2n$ nodes. The number of forks of the tree is between 1 and n.*

Proof. The tree contains $n + 1$ distinct terminal nodes corresponding to the $n + 1$ suffixes they represent. This gives the lower bound.

Each fork that is not terminal has at least two children. For a fixed number of external nodes, the maximum number of these forks is obtained when each one has exactly two children. In this case, one obtains at most n forks (terminal or not). Since for $n > 0$ the initial state is both a fork and a terminal node, one obtains the bound $(n + 1) + n - 1 = 2n$ on the total number of nodes. ∎

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press



i	0	1	2	3	4	5
$y[i]$	a	b	a	b	b	b

Figure 2.6. Representation of labels in the (compact) suffix tree $\mathfrak{S}(ababbb)$. (To be compared with the tree in Figure 2.5.) Label $(2, 4)$ of edge $(3, 1)$ represents the factor of length 4 at position 2 in y, that is, the word *abbb*.

The fact that the compact suffix tree has a linear number of nodes does not imply the linearity of its representation, because this also depends on the total size of labels of the edges. The example of a word of length n that has n distinct letters shows that this size can well be quadratic. Nevertheless, labels of edges being all factors of y, each one can be represented by a pair position-length (or also starting position-end position), provided that the word y resides in memory with the tree to allow an access to the labels. A word u that is the label of an edge (p, q) is represented by the pair $(i, |u|)$ where i is the position of some occurrence of u in y. We write $label(p, q) = (i, |u|)$ and assume that the implementation of the tree provides a direct access to this label. This representation of labels is illustrated in Figure 2.6 for the tree of Figure 2.5.

Proposition 2.2.2. *Representing labels of edges by pairs of integers, the total size of the compact suffix tree of a word is linear in its length, that is, the size of* $\mathfrak{S}(y)$ *is* $\Theta(|y|)$.

Proof. The number of nodes of $\mathfrak{S}(y)$ is $\Theta(|y|)$ according to Proposition 2.2.1. The number of edges of $\mathfrak{S}(y)$ is one unit less than the number of nodes. The assumption on the representation of labels of edges implies that each edge occupies a constant space. This gives the result. ∎

The suffix link function introduced in the preceding section finds its complete usefulness in the construction of compact suffix tries. It allows a fast construction when, moreover, the algorithm SLOWFIND of

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

the preceding section is replaced by the algorithm FASTFIND hereafter that has a similar function. The possibility of retaining only the forks of the tree, in addition to terminal states, rests on the following lemma.

Proposition 2.2.3. *In a suffix trie, the suffix link of a nonempty fork is a fork.*

Proof. For a nonempty fork, there are two cases to consider according to whether the fork, say au ($a \in \mathcal{A}$, $u \in \mathcal{A}^*$) has degree at least 2, or has degree 1 and is terminal.

Let us suppose first that the degree of au is at least 2. For two distinct letters b and c, the words aub and auc are factors of y. The same property then holds for $u = s_y(au)$ which is thus of degree at least 2 and therefore is a fork.

If the fork au has degree 1 and is terminal, then aub is a factor of y for some letter b and simultaneously au is a suffix of y. Thus, ub is a factor of y and u is a suffix of y, which shows that $u = s_y(au)$ is also a fork. ∎

The following property is used as a basis for the computation of suffix links in the algorithm SUFFIXTREE that builds the suffix tree. We denote by δ the transition function of $\mathfrak{S}(y)$.

Lemma 2.2.4. *Let (p, q) be an edge of $\mathfrak{S}(y)$ and $y[j \,..\, k-1]$, $j < k$, be its label. If q is a fork of the tree, then*

$$s_y(q) = \begin{cases} \delta(p, y[j+1 \,..\, k-1]) & \text{if } p \text{ is the initial state,} \\ \delta(s_y(p), y[j \,..\, k-1]) & \text{otherwise.} \end{cases}$$

Proof. As q is a fork, $s_y(q)$ is defined according to Proposition 2.2.3. If p is the initial state of the tree, that is, if $p = \varepsilon$, one has $s_y(q) = \delta(\varepsilon, y[j+1 \,..\, k-1])$ by definition of s_y.

In the other case, there is a single path from the initial state ending at p because $\mathfrak{S}(y)$ is a tree. Let av be the nonempty label of this path with $a \in \mathcal{A}$ and $v \in \mathcal{A}^*$ (that is, $p = av$). One has $\delta(\varepsilon, v) = s_y(p)$ and $\delta(\varepsilon, v \cdot y[j \,..\, k-1]) = s_y(q)$. It follows that $s_y(q) = \delta(s_y(p), y[j \,..\, k-1])$ (the automaton is deterministic), as announced. ∎

2.2.1. Construction

The strategy we select to build the suffix tree of y successively inserts the suffixes of y in the structure, from the longest to the shortest, as in the preceding section. As for the algorithm SUFFIXTRIE-BIS, the insertion of the tail of the running suffix is done after a slow find starting from the

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

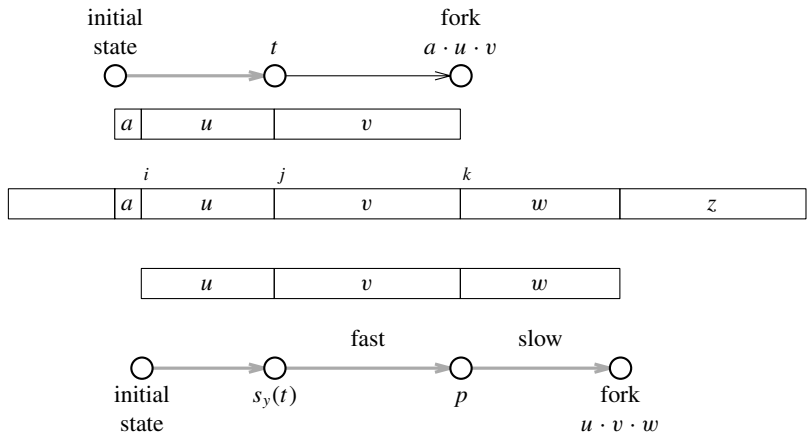


Figure 2.7. Schema for the insertion of the suffix $y[i \,..\, n-1] = u \cdot v \cdot w \cdot z$ of y in the (compact) suffix tree during its construction, when the suffix link is not defined on the fork $a \cdot u \cdot v$. Let t be the parent of this fork and v be the label of the associated edge. One first computes $p = \delta(s_y(t), v)$ using FastFind, then the fork of the suffix using SlowFind as in Section 2.1.

suffix link of the current fork. When this link does not exist it is created (lines 6–11 of SuffixTree) by using the equality of the preceding statement. Calculation is performed by the algorithm FastFind that satisfies

$$\text{FastFind}(r, j, k) = \delta(r, y[j \,..\, k-1])$$

for the r state of the tree and j, k positions on y for which

$$r \cdot y[j \,..\, k-1] \text{ is a factor of } y.$$

The diagram for the insertion of one suffix inside the tree in construction is presented in Figure 2.7.

```
SuffixTree(y, n)
 1  M ← NewAutomaton()
 2  sℓ[initial(M)] ← initial(M)
 3  (fork, k) ← (initial(M), 0)
 4  for i ← 0 to n − 1 do
 5        k ← max{i, k}
 6        if sℓ[fork] is undefined then
 7              t ← parent of fork
 8              (j, ℓ) ← label(t, fork)
 9              if t = initial(M) then
```

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

```
10                  ℓ ← ℓ − 1
11              sℓ[fork] ← FastFind(sℓ[t], k − ℓ, k)
12          (fork, k) ← SlowFindC(sℓ[fork], k)
13          if k < n then
14              q ← NewState()
15              adj[fork] ← adj[fork] ∪ {((k, n − k), q)}
16          else q ← fork
17          output[q] ← i
18  output[initial(M)] ← n
19  return M
```

Algorithm SlowFindC is merely adapted from algorithm SlowFind to take into account the fact that labels of edges are words. However, when the sought target falls in the middle of an edge it is now necessary to cut this edge. Let us notice that $\text{Target}(p, a)$, if it exists, is the state q for which a is the first letter of the label of the edge (p, q). Labels can be words of length strictly more than 1; thus, it is not true in general that $\text{Target}(p, a) = \delta(p, a)$.

```
SlowFindC(p, k)
 1  while k < n and Target(p, y[k]) is defined do
 2      q ← Target(p, y[k])
 3      (j, ℓ) ← label(p, q)
 4      i ← j
 5      do  i ← i + 1
 6          k ← k + 1
 7      while i < j + ℓ and k < n and y[i] = y[k]
 8      if i < j + ℓ then
 9          adj[p] ← adj[p] \ {((j, ℓ), q)}
10          r ← NewState()
11          adj[p] ← adj[p] ∪ {((j, i − j), r)}
12          adj[r] ← adj[r] ∪ {((i, ℓ − i + j), q)}
13          return (r, k)
14      p ← q
15  return (p, k)
```

The improvement on the execution time of the construction of a suffix tree by the algorithm SuffixTree rests, in addition to the compaction of the data structure, on an additional algorithmic element: the implementation of FastFind. Resorting to the particular algorithm described by the code below is essential to obtaining the execution time stated in Theorem 2.2.7.

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

The algorithm FastFind is used to compute a fork. It is applied to state r and word $y[j\,..\,k-1]$ only when

$$r \cdot y[j\,..\,k-1] \text{ is a factor of } y.$$

In this case, from state r there is a path whose label is prefixed by $y[j\,..\,k-1]$. Moreover, as the automaton is deterministic, the shortest of these paths is unique. The algorithm uses this property to determine edges of the path by only checking the first letter of their label. The code below, or at least its main part, implements the recurrence relation given in the proof of Lemma 2.2.5.

The algorithm FastFind is used more precisely for computing the value $\delta(r, y[j\,..\,k-1])$ (or that of $\delta(r, v)$ with the notations of the lemma). When the end of the traversed path is not the sought state, a state p is created and inserted between the last two states met.

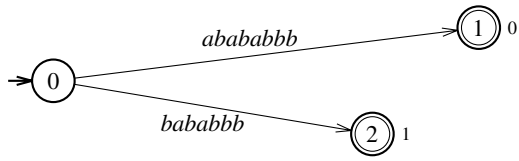
```
FastFind(r, j, k)
 1 ▷ Computation of δ(r, y[j..k − 1])
 2 if j ≥ k then
 3       return r
 4 else  q ← Target(r, y[j])
 5       (j′, ℓ) ← label(r, q)
 6       if j + ℓ ≤ k then
 7             return FastFind(q, j + ℓ, k)
 8       else  adj[r] ← adj[r] \ {((j′, ℓ), q)}
 9             p ← NewState()
10             adj[r] ← adj[r] ∪ {((j′, k − j), p)}
11             adj[p] ← adj[p] ∪ {((j′ + k − j, ℓ − k + j), q)}
12             return p
```

The work of algorithms SlowFindC and FastFind is illustrated by Figures 2.8 and 2.9.

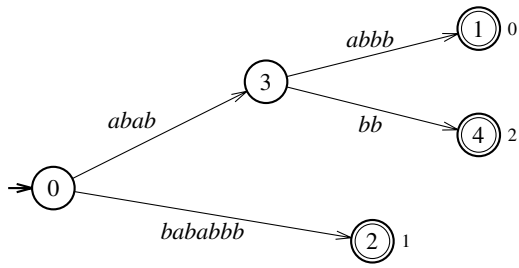
2.2.2. Complexity

The lemma which follows is used for the evaluation of the execution time of FastFind(r, j, k). It is an element of the proof of Theorem 2.2.7. It indicates that the computing time is proportional (up to a multiplicative coefficient that comes from the computing time of transitions) to the number of nodes of the traversed path, and not to the length of the label of the path, a result which one would obtain immediately by applying algorithm SlowFind (Section 2.1).

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press



(a) After insertion of *ababababb* and *babababb*.



(b) Suffix *ababbb* is added.

Figure 2.8. During the construction of $\mathfrak{S}(ababababb)$, insertion of suffixes *ababbb* and *babbb*. **(a)** Automaton obtained after the insertion of suffixes *ababababb* and *babababb*. The current fork is the initial state 0. **(b)** Suffix *ababbb* is added using letter-by-letter comparisons (slow find) and starting from state 0. This results in the creation of fork 3. The suffix link of 3 is not yet defined.

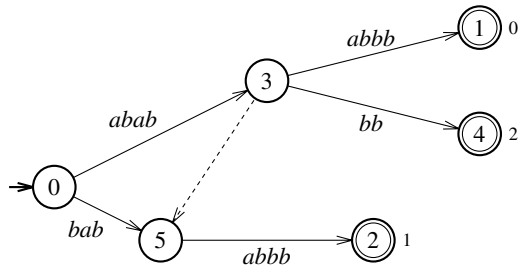
For a state r of $\mathfrak{S}(y)$ and a word v for which $r \cdot v$ is a factor of y, we denote by $end(r, v)$ the final vertex of the shortest path having origin r and whose label has v as a prefix. Observe that $end(r, v) = \delta(r, v)$ only if v is the label of the path.

Lemma 2.2.5. *Let r be a node of $\mathfrak{S}(y)$ and let v be a word such that $r \cdot v$ is a factor of y. Let $\langle r, r_1, \ldots, r_\ell \rangle$ be the path having origin r and end $r_\ell = end(r, v)$ in $\mathfrak{S}(y)$. The computation of $end(r, v)$ can be carried out in time $O(\ell \times \log \operatorname{Card} \mathcal{A})$ in the comparison model.*

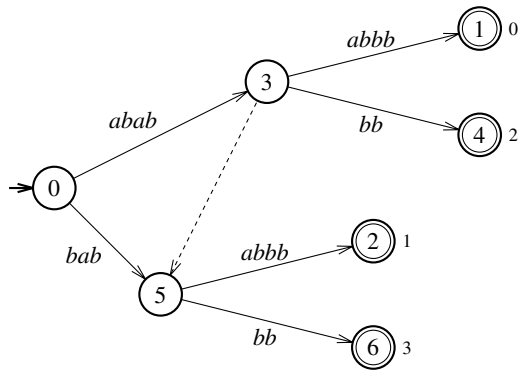
Proof. It is noticed that the path $\langle r, r_1, \ldots, r_\ell \rangle$ exists by the condition "$r \cdot v$ is a factor of y" and is unique because the tree is a deterministic automaton. If $v = \varepsilon$ one has $end(r, v) = r$. If not, let $r_1 = \textsc{Target}(r, v[0])$ and let v' be the label of edge (r, r_1). Note that

$$end(r, v) = \begin{cases} r_1 & \text{if } |v| \leq |v'| \text{ (that is, } v \text{ is a prefix of } v'), \\ end(r_1, v'^{-1}v) & \text{otherwise.} \end{cases}$$

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press



(c) Definition of the suffix link of state 3.



(d) Insertion of *babbb*.

Figure 2.9. During the construction of $\mathcal{S}(abababbb)$ (*continued*). **(c)** The first step of the insertion of suffix *babbb* starts with the definition of the suffix link of state 3, which is state 5. This is a fast find process from state 0 by word *bab*. **(d)** The second step of the insertion of *babbb* leads to the creation of state 6. State 5, which is the fork of suffix *babbb*, becomes the current fork to continue the construction.

This relation shows that each stage of the computation takes time $\alpha + \beta$, where α is a constant and β is the computing time of TARGET$(r, v[0])$. This gives time $O(\log \mathrm{Card}\, \mathcal{A})$ in the comparison model.

The computation of r_ℓ which includes traversing the path $\langle r, r_1, \ldots, r_\ell \rangle$ thus takes time $O(\ell \times \log \mathrm{Card}\, \mathcal{A})$ as announced. ∎

Corollary 2.2.6. *Let r be a node of $\mathcal{S}(y)$ and j, k be two positions on y, $j < k$, such that $r \cdot y[j\,..\,k-1]$ is a factor of y. Let ℓ be the number of states of the tree traversed during the computation of* FASTFIND(r, j, k). *Then, the execution time of* FASTFIND(r, j, k) *is $O(\ell \times \log \mathrm{Card}\, \mathcal{A})$ in the comparison model.*

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

Proof. Let $v = y[j \,..\, k-1]$ and let $\langle r, r_1, \ldots, r_\ell \rangle$ be the path ending at $end(r, v)$. The computation of $end(r, v)$ is done by FastFind that implements the recurrence relation of the proof of Lemma 2.2.5. It thus takes time $O(\ell \times \log \mathrm{Card}\, \mathcal{A})$. During the last recursive call, a state p may be created and related edges modified. This operation takes time $O(\log \mathrm{Card}\, \mathcal{A})$. This gives the total time $O(\ell \times \log \mathrm{Card}\, \mathcal{A})$ of the statement. ∎

Theorem 2.2.7. *The computation of* SuffixTree$(y) = \mathfrak{S}(y)$ *takes* $O(|y| \times \log \mathrm{Card}\, \mathcal{A})$ *time in the comparison model.*

Proof. The fact that SuffixTree$(y) = \mathfrak{S}(y)$ is based mainly on Lemma 2.2.4 by checking that the algorithm again uses the elementary technique of Section 2.1.

The evaluation of the running time rests on the following observations (see Figure 2.7):

- Each stage of the computation done by FastFind, except perhaps the last stage, leads to the traversal of a state and strictly increases the value of $k - \ell$ (j on the figure), which never decreases.
- Each stage of the computation done by SlowFind, except perhaps the last stage, strictly increases the value of k, which never decreases.
- Each other instruction of the **for** loop leads to the incrementing of variable i, which never decreases.

The number of stages done by FastFind is thus bounded by $|y|$, which gives $O(|y| \times \log \mathrm{Card}\, \mathcal{A})$ time for these stages according to Corollary 2.2.6. The same reasoning applies to the number of stages carried out by SlowFind, and also for the other stages, still giving time $O(|y| \times \log \mathrm{Card}\, \mathcal{A})$.

Therefore, one obtains a total execution time $O(|y| \times \log \mathrm{Card}\, \mathcal{A})$.

∎

2.3. Contexts of factors

We present in this section the formal basis for the construction of the minimal automaton which accepts the suffixes of a word, and is called the *suffix automaton* of the word. Some properties contribute to the proof of the construction of the automaton (Theorems 2.3.10 and 2.4.7 later).

The suffix automaton is denoted by $\mathfrak{A}(y)$. Its states are classes of the (right) syntactic equivalence associated with $\mathrm{Suff}(y)$, that is, are the sets of factors of y having the same right context within y. These states are in one-to-one correspondence with the (right) contexts of the factors of y in y itself. Let us recall that the (right) context of a word u is $\mathcal{R}_y(u) = u^{-1}\mathrm{Suff}(y)$. We

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

denote by $\equiv_y$ the syntactic congruence which is defined, for $u, v \in A^*$, by

$$u \equiv_y v$$

if and only if

$$\mathcal{R}_y(u) = \mathcal{R}_y(v).$$

One can also identify the states of $\mathfrak{A}(y)$ to sets of indices on y which are end positions of occurrences of equivalent factors.

The right contexts satisfy some properties stated below that are used later in the chapter. The first remark concerns the link between the relation "is a suffix of" and the inclusion of contexts. For any factor u of y, one denotes by

$$\textit{end-pos}(u) = \min\{|wu| \mid wu \text{ is a prefix of } y\} - 1,$$

the right position of the first occurrence of u in y. Note that $\textit{end-pos}(\varepsilon) = -1$.

Lemma 2.3.1. *Let $u, v \in \mathrm{Fact}(y)$ with $|u| \leq |v|$. Then,*

$$u \textit{ is a suffix of } v \textit{ implies } \mathcal{R}_y(v) \subseteq \mathcal{R}_y(u)$$

and

$$\mathcal{R}_y(u) = \mathcal{R}_y(v) \textit{ implies both } \textit{end-pos}(u) = \textit{end-pos}(v) \textit{ and } u \textit{ is a suffix of } v.$$

Proof. Let us suppose that u is a suffix of v. Let $z \in \mathcal{R}_y(v)$. By definition, vz is a suffix of y and, since u is a suffix of v, the word uz is also a suffix of y. Thus, $z \in \mathcal{R}_y(u)$, which proves the first implication.

Let us now suppose $\mathcal{R}_y(u) = \mathcal{R}_y(v)$. Let w, z be such that $y = w \cdot z$ with $|w| = \textit{end-pos}(u) + 1$. By definition of *end-pos*, u is a suffix of w. Therefore, z is the longest word in $\mathcal{R}_y(u)$. The assumption implies that z is also the longest word in $\mathcal{R}_y(v)$, which yields $|w| = \textit{end-pos}(v) + 1$. The words u and v are thus both suffixes of w, and as u is shorter than v one obtains that u is a suffix of v. This finishes the proof of the second implication and the whole proof. ∎

Another very useful property of the congruence is that it partitions the suffixes of a factor of y into intervals according to their length.

Lemma 2.3.2. *Let $u, v, w \in \mathrm{Fact}(y)$. If u is a suffix of v, v is a suffix of w and $u \equiv_y w$, then $u \equiv_y v \equiv_y w$.*

Proof. By Lemma 2.3.1, the assumption implies

$$\mathcal{R}_y(w) \subseteq \mathcal{R}_y(v) \subseteq \mathcal{R}_y(u).$$

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

Then, the equivalence $u \equiv_y w$ which means $\mathcal{R}_y(u) = \mathcal{R}_y(w)$ gives the conclusion. ∎

A consequence of the following property is that inclusion induces a tree structure on the right contexts. In this tree, the parent link is related to the proper inclusion of sets. This link, important for the fast construction of the automaton, corresponds to the suffix function defined then.

Corollary 2.3.3. *Let $u, v \in A^*$. Then, the contexts of u and v are comparable for inclusion or are disjoint, that is, at least one of the three following conditions is satisfied:*

1. $\mathcal{R}_y(u) \subseteq \mathcal{R}_y(v)$,
2. $\mathcal{R}_y(v) \subseteq \mathcal{R}_y(u)$,
3. $\mathcal{R}_y(u) \cap \mathcal{R}_y(v) = \emptyset$.

Proof. One proves the property by showing that the condition

$$\mathcal{R}_y(u) \cap \mathcal{R}_y(v) \neq \emptyset$$

implies

$$\mathcal{R}_y(u) \subseteq \mathcal{R}_y(v) \quad \text{or} \quad \mathcal{R}_y(v) \subseteq \mathcal{R}_y(u).$$

Let $z \in \mathcal{R}_y(u) \cap \mathcal{R}_y(v)$. Then, uz, vz are suffixes of y, and u, v are suffixes of yz^{-1}. Consequently, among u and v one is a suffix of the other. One obtains finally the conclusion by Lemma 2.3.1. ∎

2.3.1. Suffix function

On the set $\mathrm{Fact}(y)$ we consider the function s_y called the *suffix function* of y. It is defined, for all $v \in \mathrm{Fact}(y) \setminus \{\varepsilon\}$, by

$$s_y(v) = \text{longest suffix } u \text{ of } v \text{ such that } u \not\equiv_y v.$$

After Lemma 2.3.1, one deduces the equivalent definition:

$$s_y(v) = \text{ longest suffix } u \text{ of } v \text{ such that } \mathcal{R}_y(v) \subset \mathcal{R}_y(u).$$

Note that, by definition, $s_y(v)$ is a proper suffix of v (that is, $|s_y(v)| < |v|$). The following lemma shows that the suffix function s_y induces a failure function on states of $\mathfrak{A}(y)$.

Lemma 2.3.4. *Let $u, v \in \mathrm{Fact}(y) \setminus \{\varepsilon\}$. If $u \equiv_y v$, then $s_y(u) = s_y(v)$.*

Proof. By Lemma 2.3.1 one can suppose without loss of generality that u is a suffix of v. The word u cannot be a suffix of $s_y(v)$ because Lemma 2.3.2 would imply $s_y(v) \equiv_y v$, which contradicts the definition of $s_y(v)$. Consequently, $s_y(v)$ is a suffix of u. Since, by definition, $s_y(v)$ is the longest

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

suffix of v which is not equivalent to itself, it is also $s_y(u)$. Thus, $s_y(u) = s_y(v)$. ∎

Lemma 2.3.5. *Let $y \in \mathcal{A}^+$. The word $s_y(y)$ is the longest suffix of y that appears at least twice in y itself.*

Proof. The context $\mathcal{R}_y(y)$ is $\{\varepsilon\}$. As y and $s_y(y)$ are not equivalent, $\mathcal{R}_y(s_y(y))$ contains some nonempty word z. Then, $s_y(y)z$ and $s_y(y)$ are suffixes of y, which shows that $s_y(y)$ appears twice at least in y.

Any suffix w of y, longer than $s_y(y)$, is equivalent to y by definition of $s_y(y)$. It thus satisfies $\mathcal{R}_y(w) = \mathcal{R}_y(y) = \{\varepsilon\}$. Which shows that w appears only once in y and finishes the proof. ∎

The following lemma shows that the image of a factor of y by the suffix function is a word of maximum length in its equivalence class.

Lemma 2.3.6. *Let $u \in \mathrm{Fact}(y) \setminus \{\varepsilon\}$. Then, any word equivalent to $s_y(u)$ is a suffix of it.*

Proof. Let $w = s_y(u)$ and $v \equiv_y w$. We show that v is a suffix of w. The word w is a proper suffix of u. If the conclusion of the statement is false, according to Lemma 2.3.1 one obtains that w is a proper suffix of v. Then let $z \in \mathcal{R}_y(u)$. As w is a suffix of u equivalent to v, we have $z \in \mathcal{R}_y(w) = \mathcal{R}_y(v)$. Then, u and v are both suffixes of yz^{-1}, which implies that one is a suffix of the other. But this contradicts either the definition of $w = s_y(u)$ or the conclusion of Lemma 2.3.2, and proves that v is a suffix of $w = s_y(u)$. ∎

The preceding property is considered in Section 2.8 where the automaton is used as a pattern searching engine. One can check that the property of s_y is not satisfied in general on the minimal automaton which accepts the factors (and not only suffixes) of a word, or, more exactly, is not satisfied on the similar function defined from the right congruence defined from $\mathrm{Fact}(y)$ (instead of $\mathrm{Suff}(y)$).

2.3.2. Evolution of the congruence

The on-line construction of suffix automata relies on the relationship between $\equiv_{wa}$ and $\equiv_w$ which we examine here. By doing this, we consider that the generic word y is equal to wa for some letter a. The properties detailed below are also used to derive precise bounds on the size of the automaton in the following section.

The first relation states that $\equiv_{wa}$ is a refinement of $\equiv_w$.

Lemma 2.3.7. *Let $w \in \mathcal{A}^*$ and $a \in \mathcal{A}$. The congruence $\equiv_{wa}$ is a refinement of $\equiv_w$, that is for all words $u, v \in \mathcal{A}^*$, $u \equiv_{wa} v$ implies $u \equiv_w v$.*

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

Proof. Let us assume that $u \equiv_{wa} v$, that is, $\mathcal{R}_{wa}(u) = \mathcal{R}_{wa}(v)$, and show that $u \equiv_w v$, that is, $\mathcal{R}_w(u) = \mathcal{R}_w(v)$. We show $\mathcal{R}_w(u) \subseteq \mathcal{R}_w(v)$ only because the opposite inclusion results by symmetry.

If $\mathcal{R}_w(u) = \emptyset$ the inclusion is clear. If not, let $z \in \mathcal{R}_w(u)$. Then uz is a suffix of w, which implies that uza is a suffix of wa. The assumption gives that vza is a suffix of wa, and thus vz is a suffix of w, or $z \in \mathcal{R}_w(v)$, which finishes the proof. ∎

The congruence $\equiv_w$ partitions $\mathcal{A}^*$ into classes. Lemma 2.3.7 amounts to saying that these classes are unions of classes according to $\equiv_{wa}$ ($a \in \mathcal{A}$). It proves that only one or two classes with respect to $\equiv_w$ are divided into two subclasses to give the partition induced by $\equiv_{wa}$. One of these two classes consists of words not appearing in w. It contains the word wa itself which produces a new class and a new state of the suffix automaton (see Lemma 2.3.8). Theorem 2.3.10 and its corollaries give conditions for the division of another class and indicate how this is done.

Lemma 2.3.8. *Let $w \in \mathcal{A}^*$ and $a \in \mathcal{A}$. Let z be the longest suffix of wa that appears in w. If u is a suffix of wa strictly longer than z, then the equivalence $u \equiv_{wa} wa$ holds.*

Proof. It is a direct consequence of Lemma 2.3.5 because z occurs at least twice in wa. ∎

Before going to the main theorem we state an additional relation concerning right contexts.

Lemma 2.3.9. *Let $w \in \mathcal{A}^*$ and $a \in \mathcal{A}$. Then, for each word $u \in \mathcal{A}^*$,*

$$\mathcal{R}_{wa}(u) = \begin{cases} \{\varepsilon\} \cup \mathcal{R}_w(u)a & \text{if } u \text{ is a suffix of } wa, \\ \mathcal{R}_w(u)a & \text{otherwise.} \end{cases}$$

Proof. First notice that $\varepsilon \in \mathcal{R}_{wa}(u)$ is equivalent to: u is a suffix of wa. It is thus enough to show $\mathcal{R}_{wa}(u) \setminus \{\varepsilon\} = \mathcal{R}_w(u)a$.

Let z be a nonempty word of $\mathcal{R}_{wa}(u)$. We get that uz is a suffix of wa. The word uz can be written $uz'a$ with uz' a suffix of w. Consequently, $z' \in \mathcal{R}_w(u)$, and thus $z \in \mathcal{R}_w(u)a$.

Conversely, let z be a (nonempty) word in $\mathcal{R}_w(u)a$. It can be written $z'a$ for $z' \in \mathcal{R}_w(u)$. Thus, uz' is a suffix of w, which implies that $uz = uz'a$ is a suffix of wa, that is, $z \in \mathcal{R}_{wa}(u)$. This proves the converse statement and ends the proof. ∎

Theorem 2.3.10. *Let $w \in \mathcal{A}^*$ and $a \in \mathcal{A}$. Let z be the longest suffix of wa that appears in w. Let z' be the longest factor of w for which $z' \equiv_w z$. Then,*

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

for each $u, v \in \mathrm{Fact}(w)$,

$$u \equiv_w v \text{ and } u \not\equiv_w z \text{ imply } u \equiv_{wa} v.$$

Moreover, for each word u *such as* $u \equiv_w z$,

$$u \equiv_{wa} \begin{cases} z & \text{if } |u| \leq |z|, \\ z' & \text{otherwise.} \end{cases}$$

Proof. Let $u, v \in \mathrm{Fact}(w)$ be such that $u \equiv_w v$. By definition of the equivalence we get $\mathcal{R}_w(u) = \mathcal{R}_w(v)$. We suppose first that $u \not\equiv_w z$ and show that $\mathcal{R}_{wa}(u) = \mathcal{R}_{wa}(v)$, which is equivalent to $u \equiv_{wa} v$.

According to Lemma 2.3.9, we have just to show that u is a suffix of wa if and only if v is a suffix of wa. Indeed, it is enough to show that if u is a suffix of wa then v is a suffix of wa since the opposite implication results by symmetry.

So, let us suppose that u is a suffix of wa. We deduce from the fact that u is a factor of w and the definition of z that u is a suffix of z. We can thus consider the greatest integer $j \geq 0$ for which $|u| \leq |{s_w}^j(z)|$. Let us note that ${s_w}^j(z)$ is a suffix of wa (like z is), and that Lemma 2.3.2 ensures that $u \equiv_w {s_w}^j(z)$. From which we get $v \equiv_w {s_w}^j(z)$ by transitivity.

Since $u \not\equiv_w z$, we have $j > 0$. Lemma 2.3.6 implies that v is a suffix of ${s_w}^j(z)$, and thus also of wa as wished. This proves the first part of the statement.

Let us consider now a word u such as $u \equiv_w z$.

When $|u| \leq |z|$, to show $u \equiv_{wa} z$ by using the above argument, we have only to check that u is a suffix of wa because z is a suffix of wa. This, in fact, is a simple consequence of Lemma 2.3.1.

Let us suppose $|u| > |z|$. The existence of such a word u implies $z' \neq z$ and $|z'| > |z|$ (z is a proper suffix of z'). Consequently, by the definition of z, u and z' are not suffixes of wa. Using the above argument again, this proves $u \equiv_{wa} z'$ and finishes the proof. ∎

The two corollaries, stated below, of the preceding theorem refer to situations simple to manage during the construction of suffix automata.

Corollary 2.3.11. *Let* $w \in \mathcal{A}^*$ *and* $a \in \mathcal{A}$. *Let* z *be the longest suffix of* wa *that appears in* w. *Let* z' *be the longest word such as* $z' \equiv_w z$. *Let us suppose* $z' = z$. *Then, for each* $u, v \in \mathrm{Fact}(w)$,

$$u \equiv_w v \text{ implies } u \equiv_{wa} v.$$

Proof. Let $u, v \in \mathrm{Fact}(w)$ be such that $u \equiv_w v$. We prove the equivalence $u \equiv_{wa} v$. The conclusion comes directly from Theorem 2.3.10 if $u \not\equiv_w z$. Else, $u \equiv_w z$; by the assumption made on z and Lemma 2.3.1, we get $|u| \leq |z|$. Finally, Theorem 2.3.10 gives the same conclusion. ∎

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

Corollary 2.3.12. *Let $w \in \mathcal{A}^*$ and $a \in \mathcal{A}$. If the letter a does not appear in w, then, for each $u, v \in \mathrm{Fact}(w)$,*

$$u \equiv_w v \text{ implies } u \equiv_{wa} v.$$

Proof. Since a does not appear in w, the word z of Corollary 2.3.11 is the empty word. It is of course the longest of its class, which makes it possible to apply Corollary 2.3.11 and gives the same conclusion. ∎

2.4. Suffix automaton

The *suffix automaton* of a word y is the minimal automaton that accepts the set of suffixes of y. It is denoted by $\mathfrak{A}(y)$. The structure is intended to be used as an index on the word (see Section 2.6) but also constitutes a device to search for factors of y within another text (see Section 2.8). The most surprising property of the automaton is that its size is linear in the length of y although the number of factors of y can be quadratic. The construction of the automaton also takes a linear time on a fixed alphabet. Figure 2.10 shows an example of such an automaton to be compared with the trees in Figures 2.1 and 2.5.

As we do not force the automaton to be complete, the class of words which do not appear in y, whose right context is empty, is not a state of $\mathfrak{A}(y)$.

2.4.1. Size of suffix automata

The size of an automaton is expressed both by the number of its states and the number of its edges. We show that $\mathfrak{A}(y)$ has less than $2|y|$ states and less than $3|y|$ edges, for a total size $O(|y|)$. This result is based on Theorem 2.3.10 of the preceding section. Figure 2.11 shows an automaton that has the maximum number of states for a word length 7.

Proposition 2.4.1. *Let $y \in \mathcal{A}^*$ be a word of length n and let $st(y)$ be the number of states of $\mathfrak{A}(y)$. For $n = 0$, we have $st(y) = 1$; for $n = 1$, we have*

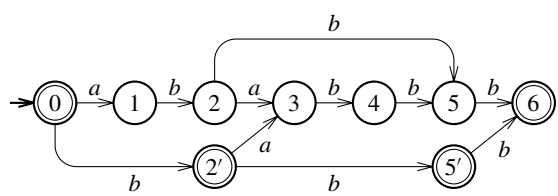


Figure 2.10. The (minimal) suffix automaton of *ababbb*.

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

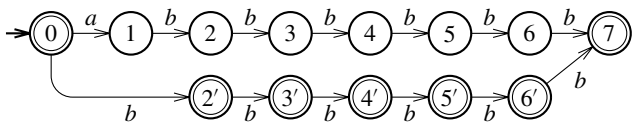


Figure 2.11. A suffix automaton with the maximum number of states.

$st(y) = 2$; for $n > 1$ finally, we have

$$n + 1 \leq st(y) \leq 2n - 1,$$

and the upper bound is reached if and only if y is of the form ab^{n-1}, for two distinct letters a, b.

Proof. The equalities concerning short words can be checked directly including $st(y) = 3$ when $|y| = 2$. Let us suppose $n > 2$. The minimal number of states of $\mathfrak{A}(y)$ is obviously $n + 1$ (otherwise the path labelled by y would contain a cycle yielding an infinite number of words recognized by the automaton); the minimal value is reached with $y = a^n$ ($a \in \mathcal{A}$).

Let us show the upper bound. By Theorem 2.3.10, each letter $y[i]$, $2 \leq i \leq n - 1$, increases by at most two the number of states of $\mathfrak{A}(y[0\,..\,i-1])$. As the number of states of $\mathfrak{A}(y[0]y[1])$ is 3, it follows that

$$\begin{aligned} st(y) &\leq 3 + 2(n-2) \\ &= 2n - 1, \end{aligned}$$

as announced.

The construction of a word of length n whose suffix automaton has $2n - 1$ states is still a simple application of Theorem 2.3.10 by noting that each letter $y[2], y[3], \ldots, y[n-1]$ must effectively lead to the creation of two states during the construction. Notice that after the choice of the first two letters, which must be different, there is no choice for the other letters. This produces the only possible form given in the statement. ∎

Lemma 2.4.2. *Let $y \in \mathcal{A}^+$ and let $ed(y)$ be the number of edges of $\mathfrak{A}(y)$. Then,*

$$ed(y) \leq st(y) + |y| - 2.$$

Proof. Let us call q_0 the initial state of $\mathfrak{A}(y)$, and consider the spanning tree of longest paths starting at q_0 in $\mathfrak{A}(y)$. The tree contains $st(y) - 1$ edges of $\mathfrak{A}(y)$ because it arrives exactly at one edge on each state except on the initial state.

With each other edge (p, a, q) we associate the suffix uav of y defined as follows: u is the label of the path starting at q_0 and ending at p; v is the label of the longest path from q arriving on a terminal state. Doing so, we get an

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

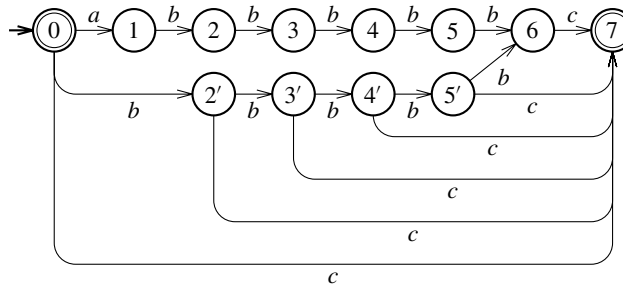


Figure 2.12. A suffix automaton with the maximum number of edges.

injection from the set of concerned edges to the set Suff(y). The suffixes y and ε are not concerned because they are labels of paths in the spanning tree. This shows that there is at most $\mathrm{Card}(\mathrm{Suff}(y) \setminus \{y, \varepsilon\}) = |y| - 1$ additional edges.

Summing up the numbers of edges of the two types, we get a maximum of $st(y) + |y| - 2$ edges in $\mathfrak{A}(y)$. ∎

Figure 2.12 shows an automaton that has the maximum number of edges for a word of length 7.

Proposition 2.4.3. *Let $y \in \mathcal{A}^*$ be a word of length n and let $ed(y)$ be the number of edges of $\mathfrak{A}(y)$. For $n = 0$, we have $ed(y) = 0$; for $n = 1$, we have $ed(y) = 1$; for $n = 2$, we have $ed(y) = 2$ or $ed(y) = 3$; finally, for $n > 2$, we have*

$$n \leq ed(y) \leq 3n - 4,$$

and the upper bound is reached if y is of the form $ab^{n-2}c$, where a, b, and c are three pairwise distinct letters.

Proof. We can directly check the results on short words. Let us consider $n > 2$. The lower bound is immediate and is reached by the word $y = a^n$ ($a \in \mathcal{A}$).

Let us then examine the upper bound. By Proposition 2.4.1 and Lemma 2.4.2 we obtain

$$\begin{aligned} ed(y) &\leq (2n - 1) + n - 2 \\ &= 3n - 3. \end{aligned}$$

The $2n - 1$ quantity is the maximum number of states obtained only if $y = ab^{n-1}$ ($a, b \in \mathcal{A}$, $a \neq b$). But for a word in this form the number of edges is only $2n - 1$. Thus, $ed(y) \leq 3n - 4$.

It can be checked that the automaton $\mathfrak{A}(ab^{n-2}c)$ (where $a, b, c \in \mathcal{A}$ with $\mathrm{Card}\{a, b, c\} = 3$) has $2n - 2$ states and $3n - 4$ edges. ∎

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

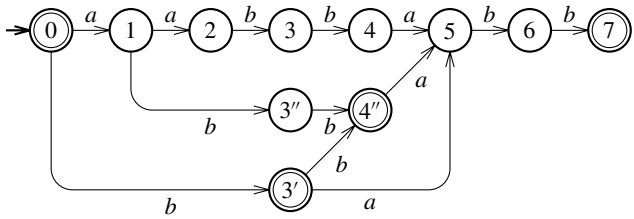


Figure 2.13. The suffix automaton $\mathfrak{A}(aabbabb)$. Suffix links on states are: $f[1] = 0$, $f[2] = 1$, $f[3] = 3''$, $f[3''] = 3'$, $f[3'] = 0$, $f[4] = 4''$, $f[4''] = 3'$, $f[5] = 1$, $f[6] = 3''$, $f[7] = 4''$. The suffix path of 7 is $\langle 7, 4'', 3', 0\rangle$, which includes all the terminal states of the automaton (see Corollary 2.4.6).

The following statement summarizes Propositions 2.4.1 and 2.4.3.

Theorem 2.4.4. *The total size of the suffix automaton of a word is linear in the length of the word.* ∎

2.4.2. Suffix links and suffix paths

Theorem 2.3.10 and its two consecutive corollaries provide the frame of the on-line construction of the suffix automaton $\mathfrak{A}(y)$. The algorithm controls the conditions which appear in these statements by means of a function defined on the states of the automaton, the suffix link function, and of a classification of the edges into solid and non-solid edges. We define these two concepts hereafter.

Let p be a state of $\mathfrak{A}(y)$, different from the initial state. State p is a class of factors of y that are equivalent with respect to equivalence $\equiv_y$. Let u be any word in the class ($u \neq \varepsilon$ because p is not the initial state). We define the suffix link of p, denoted by $f_y(p)$, as the congruence class of $s_y(u)$. The function f_y is called the *suffix link* function of the automaton. According to Lemma 2.3.4 the value of $s_y(u)$ is independent of the word u chosen in the class of p, which makes the definition coherent. The suffix link function is also called a failure function and used with this meaning in Section 2.8. An example is given in Figure 2.13.

For a state p of $\mathfrak{A}(y)$, we denote by $lg_y(p)$ the maximum length of words u in the congruence class of p. It is also the length of the longest path starting from the initial state and ending at p. The longest paths starting at the initial state form a spanning tree for $\mathfrak{A}(y)$ (a consequence of Lemma 2.3.1). Edges which belong to this tree are qualified as *solid*. In an equivalent way,

$$\text{edge } (p, a, q) \text{ is solid}$$

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

if and only if

$$lg_y(q) = lg_y(p) + 1.$$

This notion is used in the construction of the automaton.

Suffix links induce by iteration what we call *suffix paths* in $\mathfrak{A}(y)$ (see Figure 2.13). One can note that

$$q = f_y(p) \text{ implies } lg_y(q) < lg_y(p).$$

So, the sequence

$$\langle p, f_y(p), {f_y}^2(p), \ldots \rangle$$

is finite and ends at the initial state (which does not have a suffix link). It is called the suffix path of p in $\mathfrak{A}(y)$, and is denoted by $SP(p)$.

Let *last* be the state of $\mathfrak{A}(y)$ that is the class of word y itself. This state is characterized by the fact that it is not the origin of any edge. The suffix path of *last*,

$$\langle last, f_y(last), {f_y}^2(last), \ldots, {f_y}^{k-1}(last) = q_0 \rangle,$$

where q_0 is the initial state of the automaton, plays an important part in the on-line construction. It is used to effectively test conditions of Theorem 2.3.10 and its corollaries. We denote by δ the transition function of $\mathfrak{A}(y)$.

Proposition 2.4.5. *Let $u \in \text{Fact}(y) \setminus \{\varepsilon\}$ and let $p = \delta(q_0, u)$. Then, for each integer $j \geq 0$ for which ${s_y}^j(u)$ is defined,*

$$ {f_y}^j(p) = \delta(q_0, {s_y}^j(u)).$$

Proof. We prove the result by recurrence on j. If $j = 0$, ${f_y}^j(p) = p$ and ${s_y}^j(u) = u$, therefore the equality is satisfied by assumption.

Let $j > 0$ such as ${s_y}^j(u)$ is defined and suppose by recurrence assumption that ${f_y}^{j-1}(p) = \delta(i, {s_y}^{j-1}(u))$. By definition of f_y, $f_y({f_y}^{j-1}(p))$ is the congruence class of the word $s_y({s_y}^{j-1}(u))$. Consequently, ${f_y}^j(p) = \delta(q_0, {s_y}^j(u))$, which completes the recurrence and the proof. ∎

Corollary 2.4.6. *The terminal states of $\mathfrak{A}(y)$ are the states of the suffix path of last, SP(last).*

Proof. First, we prove that states of the path suffix are terminal. Let p be any state of $SP(last)$. One has $p = {f_y}^j(last)$ for some $j \geq 0$. Because $last = \delta(q_0, y)$, Proposition 2.4.5 implies $p = \delta(q_0, {s_y}^j(y))$; and as ${s_y}^j(y)$ is a suffix of y, p is a terminal state.

Conversely, let p be a terminal state of $\mathfrak{A}(y)$. Let then u be a suffix of y such that $p = \delta(q_0, u)$. Since u is a suffix of y, we can consider the greatest integer $j \geq 0$ for which $|u| \leq |{s_y}^j(y)|$. By Lemma 2.3.2 one

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

obtains $u \equiv_y s_y{}^j(y)$. Thus, $p = \delta(q_0, s_y{}^j(y))$ by definition of $\mathfrak{A}(y)$. Therefore, Proposition 2.4.5 applied to y implies $p = f_y{}^j(last)$, which proves that p appears in $SP(last)$. This ends the proof. ∎

2.4.3. On-line construction

It is possible to build the suffix automaton of y by applying to the suffix trie of Section 2.1 standard algorithms that minimize automata. But the suffix trie can be of quadratic size, which gives the time and space complexity of this approach. We present an on-line construction algorithm that avoids this problem and works in linear space with an execution time $O(|y| \times \log \mathrm{Card}\, \mathcal{A})$.

The algorithm treats the prefixes of y from the shorter, ε, to the longest, y itself. At each stage, just after having treated prefix w, the following information is available:

- The suffix automaton $\mathfrak{A}(w)$ with its transition function δ.
- The table f, defined on the states of $\mathfrak{A}(w)$, which implements the suffix function f_w.
- The table L, defined on the states of $\mathfrak{A}(w)$, which implements the function length, lg_w.
- The state *last*.

Terminal states of $\mathfrak{A}(w)$ are not explicitly marked, they are given implicitly by the suffix path of *last* (Corollary 2.4.6). The implementation of $\mathfrak{A}(w)$ with these additional elements is discussed just before the analysis of complexity of the computation.

Algorithm SUFFIXAUTOMATON that builds the suffix automaton of y relies on the procedure EXTENSION given further. This procedure treats the next letter of word y. It transforms the suffix automaton $\mathfrak{A}(w)$ already built into the suffix automaton $\mathfrak{A}(wa)$ (wa is a prefix of y, $a \in \mathcal{A}$). After all extensions, terminal states are eventually marked explicitly (lines 7 to 10).

```
SUFFIXAUTOMATON(y, n)
 1  M ← NEWAUTOMATON()
 2  L[initial(M)] ← 0
 3  last[M] ← initial(M)
 4  for each letter a of y, sequentially do
 5        ▷ Extension of M by letter a
 6        EXTENSION(a)
 7  p ← last[M]
 8  do    terminal(p) ← TRUE
 9        p ← f[p]
10  while p is defined
11  return M
```

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

Contrary to what happens for the construction of suffix trees, a state-splitting operation is necessary in some circumstances. It is realized by the following algorithm CLONE.

```
EXTENSION(a)
 1  new ← NEWSTATE()
 2  L[new] ← L[last[M]] + 1
 3  p ← last[M]
 4  do  adj[p] ← adj[p] ∪ {(a, new)}
 5      p ← f[p]
 6  while p is defined and TARGET(p, a) is undefined
 7  if p is undefined then
 8      f[new] ← initial(M)
 9  else q ← TARGET(p, a)
10      if (p, a, q) is a solid edge, that is, L[p] + 1 = L[q] then
11          f[new] ← q
12      else clone ← CLONE(p, a, q)
13          f[new] ← clone
14  last[M] ← new
```

```
CLONE(p, a, q)
 1  clone ← NEWSTATE()
 2  L[clone] ← L[p] + 1
 3  for each (b, q′) ∈ adj[q] do
 4      adj[clone] ← adj[clone] ∪ {(b, q′)}
 5  f[clone] ← f[q]
 6  f[q] ← clone
 7  do  adj[p] ← adj[p] \ {(a, q)}
 8      adj[p] ← adj[p] ∪ {(a, clone)}
 9      p ← f[p]
10  while p is defined and TARGET(p, a) = q
11  return clone
```

Figures 2.14, 2.15, 2.16, and 2.17 illustrate how the procedure EXTENSION works.

Theorem 2.4.7. *Algorithm* SUFFIXAUTOMATON *builds a suffix automaton, that is* SUFFIXAUTOMATON(y) *is the automaton* $\mathfrak{A}(y)$*, for* $y \in \mathcal{A}^*$.

Proof. We show by recurrence on $|y|$ that the automaton is computed correctly, as well as tables L and f and state *last*. It is shown then that terminal states are computed correctly.

If $|y| = 0$, the algorithm builds an automaton consisting of only one state which is both an initial and a terminal state. No transition is defined.

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

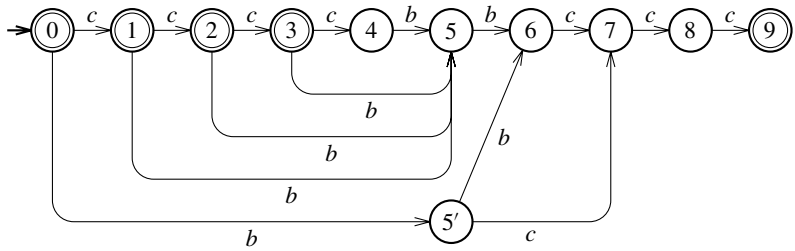


Figure 2.14. Automaton $\mathfrak{A}(ccccbbccc)$ on which is illustrated in Figures 2.15, 2.16, and 2.17 the procedure EXTENSION(a) according to three cases.

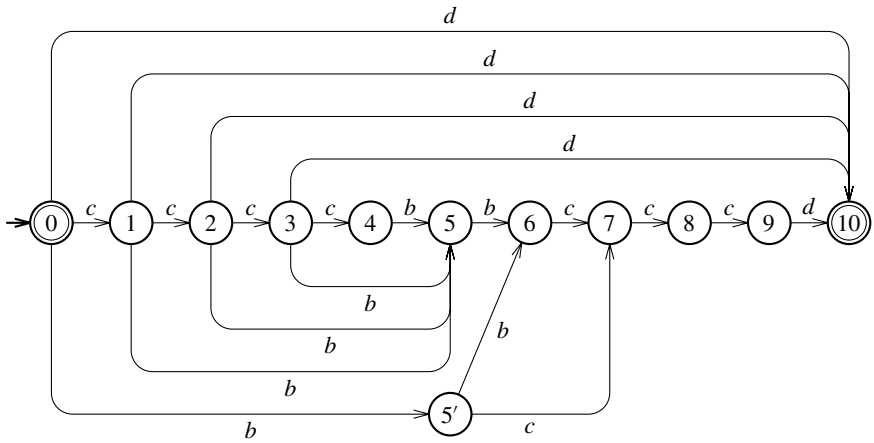


Figure 2.15. Suffix automaton $\mathfrak{A}(ccccbbcccd)$ obtained by extending $\mathfrak{A}(ccccbbccc)$ of Figure 2.14 by letter d. During the execution of the first loop of EXTENSION(d), state p traverses the suffix path $\langle 9, 3, 2, 1, 0\rangle$. At the same time, edges labelled by letter d are created, starting from these states and leading to 10, the last created state. The loop stops at the initial state. This situation corresponds to Corollary 2.3.12.

The automaton thus recognizes the language $\{\varepsilon\}$ which is Suff(y). Elements f and *last* as well as tables L and f are also correctly calculated.

We now consider that $|y| > 0$ and that $y = wa$, for $a \in \mathcal{A}$ and $w \in \mathcal{A}^*$. We suppose, by recurrence, that the current automaton M is $\mathfrak{A}(w)$ with its transition function δ_w, that $q_0 = initial(M)$, that $last = \delta_w(q_0, w)$, that table L satisfies $L[p] = lg_w(p)$ for any state p, and that table f satisfies $f[p] = f_w(p)$ for any state p different from the initial state.

We first show that the procedure EXTENSION carries out correctly the transformation of the automaton M, of the variable $last$, and of the tables L and f.

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

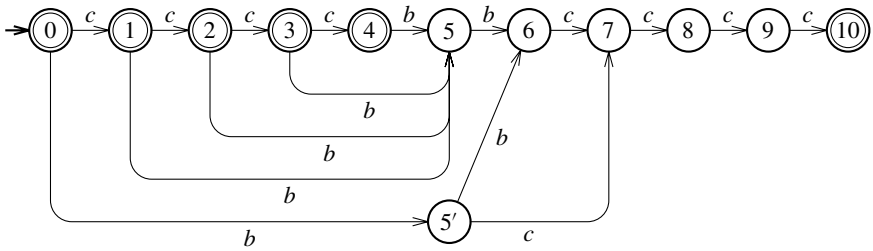


Figure 2.16. Suffix automaton $\mathfrak{A}(ccccbbcccc)$ obtained by extending $\mathfrak{A}(ccccbbccc)$ of Figure 2.14 by letter c. The first loop of the procedure EXTENSION(c) stops at state $3 = f[9]$ because an edge labelled by c starts from this state. Moreover, the edge $(3, c, 4)$ is solid. We obtain directly the suffix link of the new state created: $f[10] = \delta(3, c) = 4$. There is nothing else to do according to Corollary 2.3.11.

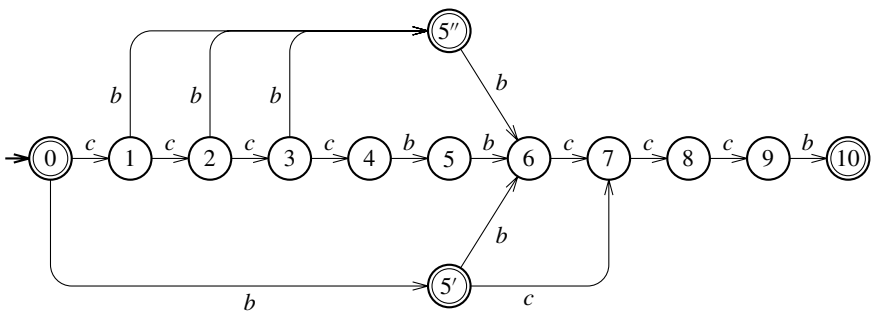


Figure 2.17. Suffix automaton $\mathfrak{A}(ccccbbcccb)$ obtained by extending $\mathfrak{A}(ccccbbccc)$ of Figure 2.14 by letter b. The first loop of the procedure EXTENSION(b) stops at state $3 = f[9]$ because an edge labelled by b starts from this state. In the automaton $\mathfrak{A}(ccccbbccc)$ edge $(3, b, 5)$ is not solid. The word $cccb$ is a suffix of $ccccbbcccb$ but $ccccb$ is not, although they both lead to state 5. This state is duplicated into the final state $5''$ that is the class of factors $cccb$, ccb and cb. Edges $(3, b, 5)$, $(2, b, 5)$ and $(1, b, 5)$ of $\mathfrak{A}(ccccbbccc)$ are redirected onto $5''$ according to Theorem 2.3.10.

The variable p of procedure EXTENSION runs through the states of the suffix path $SP(last)$ of $\mathfrak{A}(w)$. The first loop creates transitions labelled by a targeted at the new state new in agreement with Lemma 2.3.8. We also have the equality $L[new] = lg_y(new)$.

When the first loop stops, three disjoint cases arise:

1. p is not defined,
2. (p, a, q) is a solid edge,
3. (p, a, q) is a nonsolid edge.

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

Case 1. This situation occurs when the letter a does not occur in w; one has then $f_y(new) = q_0$. Thus, after the instruction at line 8 the equality $f[new] = f_y(new)$ holds. For the other states r, one has $f_w(r) = f_y(r)$ according to Corollary 2.3.12. Which gives the equalities $f[r] = f_y(r)$ at the end of the execution of the procedure EXTENSION.

Case 2. Let u be the longest word for which $\delta(q_0, u) = p$. By recurrence and Lemma 2.3.6, we have $|u| = lg_w(p) = L[p]$. The word ua is the longest suffix of y which is a factor of w. Thus, $f_y(new) = q$, which shows that $f[new] = f_y(new)$ after the instruction of line 11.

Since edge (p, a, q) is solid, by recurrence again, we have $|ua| = L[q] = lg_y(q)$, which shows that the words equivalent to ua according to $\equiv_w$ are not longer than ua. Corollary 2.3.11 applies with $z = ua$. And as in case 1, $f[r] = f_y(r)$ for all the states different from *new*.

Case 3. Let u be the longest word for which $\delta(q_0, u) = p$. The word ua is the longest suffix of y which is a factor of w. So, $f_y(new) = q$, and thus $f[new] = f_y(new)$. Since edge (p, a, q) is not solid, ua is not the longest word in its congruence class according to $\equiv_w$. Theorem 2.3.10 applies with $z = ua$, and z' the longest word for which $\delta(q_0, z') = q$. The class of ua according to $\equiv_w$ is divided into two subclasses with respect to $\equiv_{wa}$. They correspond to states q and *clone*.

Words v no longer than ua and such as $v \equiv_w ua$ are of the form $v'a$ with v' a suffix of u (a consequence of Lemma 2.3.1). Before the execution of the last loop, all these words v satisfy $q = \delta_w(q_0, v)$. Consequently, just after the execution of the loop, they satisfy $clone = \delta_y(q_0, v)$, as required by Theorem 2.3.10. Words v longer than ua and such as $v \equiv_w ua$ satisfy $q = \delta_y(q_0, v)$ after the execution of the loop, as required by Theorem 2.3.10, again. One can check that table f is updated correctly.

For each of the three cases, one can check that the value of *last* is correctly computed at the end of the execution of the procedure EXTENSION.

Finally, the recurrence shows that automaton M, state *last*, tables L and f are correct after the execution of procedure EXTENSION.

It remains to be checked that terminal states are correctly marked during the execution of the last loop of algorithm SUFFIXAUTOMATON. But this is a straight consequence of Corollary 2.4.6 because variable p runs through the suffix path of *last*. ∎

2.4.4. Complexity

To analyse the complexity of the algorithm SUFFIXAUTOMATON we first describe a possible implementation of the elements necessary for the construction.

We suppose that the automaton is represented by lists of successors. By doing this, operations of addition, update, and access concerning an edge

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

are performed in time $O(\log \mathrm{Card}\, \mathcal{A})$ with an efficient implementation of the lists. Function f_y is realized by table f which gives access to $f_y(p)$ in constant time.

To implement the solidity of edges table L is used. It represents the function lg_y, as the description of the procedure EXTENSION suggests (line 10). Another way of doing it uses a Boolean value per edge of the automaton. This induces a slight modification of the procedure which we describe as follows: each first edge created during the execution of the loops at lines 4–6 and lines 7–10 must be marked as solid; the other created edges are marked as nonsolid. This type of implementation does not require the use of table L, which can then be eliminated, reducing the memory space used. Nevertheless, table L finds its utility in applications like those of Section 2.8. We retain that the two types of implementation provide a constant-time access to the quality (solid or not solid) of an edge.

Theorem 2.4.8. *Algorithm* SUFFIXAUTOMATON *can be implemented so that the construction of* $\mathfrak{A}(y)$ *takes time* $O(|y| \times \log \mathrm{Card}\, \mathcal{A})$ *in a memory space* $O(|y|)$.

Proof. We choose an implementation by lists of successors for the transition function. States of $\mathfrak{A}(y)$ and tables f and L require a space $O(st(y))$, lists of edges a space $O(ed(y))$. Thus, the complete implementation takes a space $O(|y|)$, as a consequence of Propositions 2.4.1 and 2.4.3.

Another consequence of these propositions is that all the operations carried out either once per state or once per edge of the final automaton take a total time $O(|y| \times \log \mathrm{Card}\, \mathcal{A})$. The same result applies to the operations which are performed once per letter of y. It thus remains to be shown that the time spent for the executions of the two loops at lines 4–6 and lines 7–10 of the procedure EXTENSION are of the same order, namely $O(|y| \times \log \mathrm{Card}\, \mathcal{A})$.

We examine initially the case of the first loop. Let us consider the execution of the procedure EXTENSION during the transformation of $\mathfrak{A}(w)$ into $\mathfrak{A}(wa)$ (wa is a prefix of y, $a \in \mathcal{A}$). Let u be the longest word of state p during the test at line 6. The initial value of u is $s_w(w)$, and its final value satisfies $ua = s_{wa}(wa)$ (if p is defined). Let $k = |w| - |u|$ be the position of the suffix occurrence of u in w. Then, each test strictly increases the value of k during a call to the procedure. Moreover, the initial value of k at the beginning of the execution of the next call is not smaller than its final value reached at the end of the execution of the current call. So, k is never decreased and thus tests and instructions of this loop are done in $O(|y|)$.

A similar argument applies to the second loop at lines 7–10 of the procedure EXTENSION. Let v be the longest word of p during the test of the loop. The initial value of v is ${s_w}^j(w)$, for $j \geq 2$, and its final value

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

satisfies $va = s_{wa}{}^2(wa)$ (if p is defined). Then, the position of v as a suffix of w increases strictly at each test during successive calls to the procedure. Thus, again, tests and instructions of the loop are done in $O(|y|)$ time.

Consequently, the cumulated time of the executions of the two loops is $O(|y| \times \log \operatorname{Card} \mathcal{A})$, which finishes the proof. ∎

On a small alphabet, one can still choose an implementation of the automaton that is even more efficient than that by lists of successors, to the detriment of memory space, however. It is enough to use a transition matrix within $O(|y| \times \operatorname{Card} \mathcal{A})$ memory space and manage it like a sparse table. With this particular management, any operation on edges is done in constant time, which leads to the following result.

Theorem 2.4.9. *When the alphabet is fixed, algorithm* SUFFIXAUTOMATON *can be implemented so that the construction of* $\mathfrak{A}(y)$ *takes time* $O(|y|)$ *in a memory space* $O(|y| \times \operatorname{Card} \mathcal{A})$.

Proof. One can use, to implement the transition matrix, the technique for representing sparse tables which gives a direct access to each one of its entries while avoiding initializing the complete matrix. ∎

2.5. Compact suffix automaton

In this section, we describe briefly how to build a *compact suffix automaton* denoted by $\mathfrak{A}^c(y)$ for $y \in \mathcal{A}^*$. This automaton can be seen as the compact version of the suffix automaton of the preceding section, that is, it is obtained by removing the states that have only one outgoing transition and that are not terminal. It is the process used on the suffix trie of Section 2.1 to produce a structure of linear size.

The compact suffix automaton is also the minimized version, in the sense of automata theory, of the (compact) suffix tree of Section 2.2. It is obtained by identifying subtrees which recognize the same words.

Figure 2.18 shows the compact suffix automaton of *ababbb* that can be compared to the compact tree of Figure 2.5 and to the automaton of Figure 2.10.

Exactly as for the tree $\mathfrak{T}(y)$, in the automaton $\mathfrak{A}(y)$ we call *fork* any state that is of (outgoing) degree at least 2, or that is both of degree 1 and terminal. Forks of suffix automata satisfy the same property as forks of suffix trees, property which allows the compaction of the automaton. The proof of the next proposition is an immediate adaptation of that of Proposition 2.2.3.

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

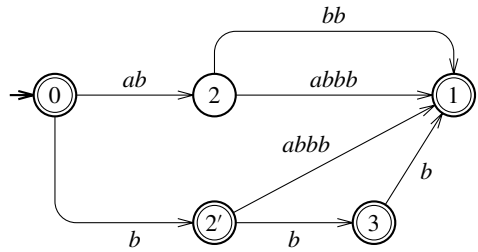
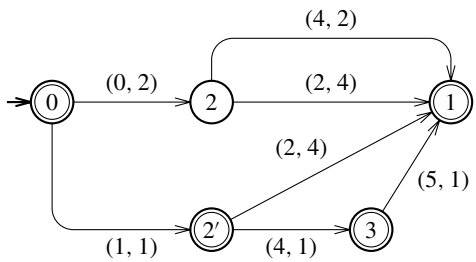


Figure 2.18. The compact suffix automaton $\mathfrak{A}^c(ababbb)$.



i	0	1	2	3	4	5
$y[i]$	a	b	a	b	b	b

Figure 2.19. Representation of labels in the compact suffix automaton $\mathfrak{A}^c(ababbb)$. (To be compared with the automaton in Figure 2.18.)

Proposition 2.5.1. *In the suffix automaton of a word, the suffix link of a fork (different from the initial state) is a fork.* ∎

When one removes nonfork states in $\mathfrak{A}(y)$, edges of the automaton must be labelled by (not empty) words and not only by letters. To get a structure of size linear in the length of y, labels of edges must not be stored explicitly. One represents them in constant space by means of a couple of integers. If the word u is a label of an edge (p, q), it is represented by the pair $(i, |u|)$ for which i is the position of an occurrence of u in y. We denote the label by $label(p, q) = (i, |u|)$ and suppose that the implementation of the automaton provides a direct access to it. This forces the storing of the word y together with the data structure. Figure 2.19 indicates how labels of the compact suffix automaton of *ababbb* are represented.

The size of compact suffix automata can be evaluated directly from sizes of compact suffix trees and of suffix automata.

Proposition 2.5.2. *Let $y \in \mathcal{A}^*$ be a word of length n and let $e_c(y)$ be the number of states of $\mathfrak{A}^c(y)$. For $n = 0$, we have $e_c(y) = 1$; for $n > 0$, we have*

$$2 \leq e_c(y) \leq n + 1,$$

and the upper bound is reached for $y = a^n$, $a \in \mathcal{A}$.

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

Proof. The result can be checked directly for the empty word.

Let us suppose $n > 0$. Let c be a letter, $c \notin \mathcal{A}$, and let us consider the tree $\mathfrak{S}(y \cdot c)$. This tree has exactly $n + 1$ external nodes on each one of which arrives an edge whose label ends by letter c. The tree has at most n internal nodes because they have at least two outgoing edges. When minimized to get a compact automaton, all external nodes are identified in only one state, which reduces the number of states to $n + 1$ at most. Removal of the letter c does not increase this value, which gives the upper bound. It can immediately be checked that $\mathfrak{A}^c(a^n)$ has $n + 1$ states exactly and that the obvious lower bound is reached when the alphabet of y has size n. ∎

Proposition 2.5.3. *Let $y \in \mathcal{A}^*$ be a word of length n and let $f_c(y)$ be the number of edges of $\mathfrak{A}^c(y)$. For $n = 0$, we have $f_c(y) = 0$; for $n = 1$, we have $f_c(y) = 1$; for $n > 1$, we have*

$$f_c(y) \leq 2(n - 1),$$

and the upper bound is reached for $y = a^{n-1}b$, where a, b are two distinct letters.

Proof. After checking the results for the short words, one notes that if x is of the form a^n, $n > 1$, one has $f_c(y) = n - 1$, a quantity that is smaller than $2(n - 1)$.

Let us suppose now that $\mathrm{Card}\, alph(y) \geq 2$. We continue the proof of the preceding lemma by still considering the word $y \cdot c$, $c \notin \mathcal{A}$. Its suffix tree has at most $2n$ nodes. Thus it has at most $2n - 1$ edges, which after compaction gives $2n - 2$ edges since the edges labelled by c disappear. This gives the announced upper bound. The automaton $\mathfrak{A}^c(a^{n-1}b)$ has n states and $2n - 2$ edges, as can be directly checked. ∎

The construction of $\mathfrak{A}^c(y)$ can be carried out starting from the tree $\mathfrak{S}(y)$ or from the automaton $\mathfrak{A}(y)$ (see Problems 2.5.1 and 2.5.2). However, to save memory space at construction time one rather takes advantage of a direct construction. It is the schema of this construction that is sketched here.

The construction borrows elements from the algorithms SUFFIXTREE and SUFFIXAUTOMATON. Thus, the edges of the automaton are marked as solid or not solid. The created edges targeted at new leaves of the tree become edges to state *last*. We also use the concepts of slow and fast traversal from the construction of suffix trees. It is on these two procedures that the changes are essential, and that are added duplications of states and redirections of edges as for the construction of suffix automata.

During the execution of a slow traversal, the attempt at crossing a non-solid edge leads to cloning its target, with a duplication similar to that done

during the execution of procedure EXTENSION at line 6. One can note that certain edges can be redirected by this process.

The second important point in the adaptation of the algorithms of the preceding sections relates to the fast traversal procedure. The main algorithm calls it for the definition of a suffix link as in the algorithm SUFFIXTREE. The difference comes when the target of a suffix link for a last-created fork (see lines 8–11 in procedure FASTFIND) is created. If a new state has to be created in the middle of a solid edge, the same process applies. But, if the edge is not solid, during a first step the edge is only redirected towards the concerned fork, and its label is updated accordingly. This leaves the suffix link undefined and leads to an iteration of the same process.

Phenomena that have just been described intervene in any sequential construction of this type of automaton. Taking them into account is necessary for a correct sequential computation of $\mathfrak{A}^c(y)$. They are present in the construction of $\mathfrak{A}^c(ababbb)$ (see Figure 2.18) for which three stages are detailed in Figure 2.20.

To conclude the section, we state the complexity of the direct construction of the compact suffix automaton. The formal description and the proof of the algorithm are left to the reader.

Proposition 2.5.4. *The computation of the compact suffix automaton* $\mathfrak{A}^c(y)$ *can be done in time* $O(|y| \times \log \operatorname{Card} \mathcal{A})$ *in a space* $O(|y|)$. ∎

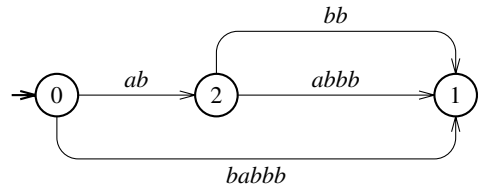
2.6. Indexes

Techniques introduced in the preceding sections find immediate applications in the design of indexes on textual data. The utility of considering the suffixes of a text for this kind of application comes from the obvious remark that any factor of a word is a prefix of some suffix of the text. Using suffix structures thus provides a kind of direct access to all the factors of a word or a language, and it is certainly the main interest of these techniques. This property gives rise to an implementation of an index on a text or a family of texts, with efficient algorithms for the basic operations (Section 2.6.2) such as questions of membership, location, and computation of lists of occurrences of patterns. Section 2.6.3 gives a solution in the form of a transducer.

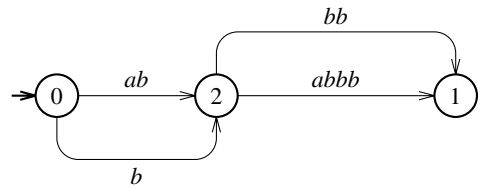
2.6.1. Implementation of indexes

The aim of an index is to provide an efficient mechanism for answering certain questions concerning the contents of a fixed text. This word is

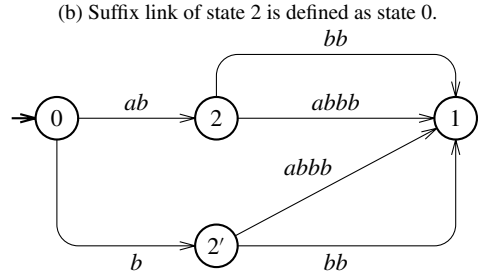
https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press



(a) After three insertions.



(b) Suffix link of state 2 is defined as state 0.



(c) Duplication of state 2.

Figure 2.20. Three steps of the construction of $\mathfrak{A}^{\mathrm{C}}(ababbb)$. **(a)** Automaton right after the insertion of the three longest suffixes of the word *ababbb*. The suffix link of state 2 is still undefined. **(b)** Computation by fast find of the suffix link of state 2, which results in transforming the edge $(0, babbb, 1)$ into $(0, b, 2)$. At the same time, the suffix *bbb* is inserted. **(c)** Insertion of the next suffix, *bb*, is done by slow find starting from state 0. Since edge $(0, b, 2)$ is not solid, its target, state 2, is duplicated as $2'$, which has the same transitions as 2. To finish the insertion of suffix *bb* it remains to cut the edge $(2', bb, 1)$ to insert state 3. Finally, the rest of the construction amounts to determining final states, and we get the automaton of Figure 2.18.

denoted by y ($y \in \mathcal{A}^*$) and its length is n ($n \in \mathbb{N}$). An *index* on y can be regarded as an abstract data type whose basic set is the set of factors of y, $\mathrm{Fact}(y)$, and that includes operations for accessing information related to factors of y. The concept is similar to the index of a book, which provides pointers to pages from a set of selected keywords. We rather consider what

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

can be called a *generalized index*, in which all the factors of the text are present. We describe indexes for only one word, but extending methods to a finite number of words is in general a simple matter.

We consider four principal operations on the index of a text. They are related to a word x, the query, to be searched for in y: membership, position, number of occurrences, and list of positions. This set of operations is often extended in real applications, in connection with the nature of data represented by y, to yield information retrieval systems. But the four operations we consider constitute the technical basis from which can be developed broader systems of queries.

For implementing indexes, we choose to treat the main method that leads to efficient and sometimes optimal algorithms. It is based on one of the data structures that represent suffixes of y and that are described in previous sections. The choice of the structure produces variations of the method. In this section we recall the elements of the data structures that must be available to execute the index operations. The operations themselves are treated in the next section.

The implementation of an index is built on automata of the preceding sections. Let us recall the data structures necessary to use the suffix tree, $\mathfrak{S}(y)$, of y. They are composed of:

- The word y itself stored in a table.
- An implementation of the automaton in the form of a transition matrix or list of edges per state, to represent the transition function δ, the access to the initial state, and a table of terminal states, for example.
- The table $s\ell$, defined on states, which represents the suffix link function of the tree.

Note that the word y itself must be maintained in memory, because the labelling of edges refers to it (see Section 2.2). The suffix link is useful for only certain applications, it can of course be eliminated when the implemented operations do not make use of it.

One can also consider the suffix automaton of y, $\mathfrak{A}(y)$, which produces in a natural way an index on factors of the text y. The structure includes:

- an implementation of the automaton as for the tree above,
- the table f that implements the failure function defined on states,
- the table L that indicates for each state the maximum length of the words reaching this state.

For this automaton it is not necessary to store the word y in memory. It appears in the automaton as the label of the longer path starting from the initial state. Tables f and L can be omitted if they are not useful for the set of selected operations.

Lastly, the compact version of the suffix automaton can be used in order to reduce even more the memory capacity needed to store the structure. Its

implementation uses in a standard way the same elements as for the suffix automaton (in a noncompact version) with, in addition, the word y in order to access to labels of edges, as for the suffix tree. One gets a noticeable space reduction in using this structure rather than the two preceding ones.

In the rest of the section we examine several types of solutions for realizing basic operations on indexes.

2.6.2. Basic operations

We consider in this section four operations related to factors of text y: membership (in Fact(y)), first position, number of occurrences, and list of the positions. The algorithms are presented after the global description of these four operations.

The first operation on an index is the membership of word x to the index, that is, the question of knowing if x is a factor of y. This question can be specified in two complementary ways according to whether one expects to find an occurrence of x in y. If x does not appear in y, it is often interesting in practice to find the longest prefix of x which is a factor of y. It is usually the type of response necessary to realize sequential searches in text editors.

Membership Given $x \in \mathcal{A}^*$, find the longest prefix of x that belongs to Fact(y).

In the contrary case ($x \in$ Fact(y)), methods produce without much modification the position of an occurrence of x, and even the position of the first or last occurrence of x in y.

Position Given x a factor of y, find the (left) position of its first (respectively last) occurrence in y.

Knowing that x is in the index, another relevant item of information is its number of occurrences in y. This information can drive later researches differently.

Number of occurrences Given x a factor of y, find how many times x appears in y.

Lastly, under the same assumption as before, complete information on the location of x in y is provided by the list of positions of its occurrences.

List of positions Given x a factor of y, produce the list of positions of the occurrences of x in y.

We describe solutions obtained by using the above data structures. It should be noticed that the structures sometimes require enrichment in order to guarantee an efficient execution of the algorithms.

Proposition 2.6.1. *Given one of the automata* $\mathfrak{S}(y)$, $\mathfrak{A}(y)$, *or* $\mathfrak{A}^c(y)$, *computing the longest prefix* u *of* x *that is a factor of* y *can be carried out in time* $O(|u| \times \log \operatorname{Card} \mathcal{A})$ *within memory space* $O(|y|)$.

Proof. By means of $\mathfrak{A}(y)$, in order to determine the word u, it is enough to follow a path labelled by a prefix of x starting from the initial state of the automaton. The traversal stops when a transition misses or when x is exhausted. This produces the longest prefix of x which is also a prefix of the label of a path starting at the initial state, that is, which appears in y since all the factors of y are labels of these paths. Overall, this is done after $|u|$ successful transitions and possibly one unsuccessful transition (when u is a proper prefix of x) at the end of the test. As each transition takes a time $O(\log \operatorname{Card} \mathcal{A})$ for an implementation in space $O(|y|)$ (by lists of successors), we obtain a total time $O(|u| \times \log \operatorname{Card} \mathcal{A})$.

The same process works with $\mathfrak{S}(y)$ and $\mathfrak{A}^c(y)$. Taking into account the representation of these structures, certain transitions are done by simple letter comparisons, but the maximum execution time is unchanged. ∎

POSITION

We now examine the operations for which it is supposed that x is a factor of y. The test of membership which can be carried out separately as in the preceding proposition, can also be integrated into the solutions of the other problems that interest us here. The use of transducers, which extend suffix automata for this type of question, is considered in the following section.

Finding the position $pos_y(x)$ of the first occurrence of x in y amounts to calculating its right position $end\text{-}pos_y(x)$ (see Section 2.3) because

$$pos_y(x) = end\text{-}pos_y(x) - |x| + 1.$$

Moreover, this is also equivalent to computing the maximum length of right contexts of x in y,

$$lc_y(x) = \max\{|z| \mid z \in \mathcal{R}_y(x)\},$$

because

$$pos_y(x) = |y| - lc_y(x) - |x|.$$

In a symmetrical way, in order to find the position $last\text{-}pos_y(x)$ of the last occurrence of x in y, it remains to calculate the minimal length $sc_y(x)$ of

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

its right contexts because

$$\textit{last-pos}_y(x) = |y| - sc_y(x) - |x|.$$

To be able to quickly answer requests related to the first or last positions of factors of y, structures of indexes are not sufficient alone, at least if one seeks to obtain optimal execution times. Consequently, one precomputes two tables indexed by the states of the selected automaton and that represent functions lc_y and sc_y. One thus obtains the following result.

Proposition 2.6.2. *Automata $\mathfrak{S}(y)$, $\mathfrak{A}(y)$, and $\mathfrak{A}^c(y)$ can be preprocessed in time $O(|y|)$ so that the first (or last) position on y of a factor x of y, as well as the number of occurrences of x, can be computed in time $O(|x| \times \log \operatorname{Card} \mathcal{A})$ within memory space $O(|y|)$.*

Proof. Let us call M the selected structure, δ its transition function, F its set of edges, and T its terminal states.

To begin let us consider the computation of $pos_y(x)$. The preprocessing of M relates to the computation of a table LC defined on states of M and aimed at representing the function lc_y. For a state p and a word $u \in \mathcal{A}^*$ with $p = \delta(\textit{initial}(M), u)$, we define

$$LC[p] = lc_y(u),$$

a quantity that is independent of the word u that labels a path from the initial state to p, according to Lemma 2.3.1. This value is also the maximum length of paths starting at p and ending at a terminal state in the automaton $\mathfrak{A}(y)$. For $\mathfrak{S}(y)$ and $\mathfrak{A}^c(y)$ this consideration still applies by defining the length of an edge as that of its label.

The table LC satisfies the recurrence relation:

$$LC[p] = \begin{cases} 0 & \text{if } \deg(p) = 0, \\ \max\{\ell + LC[q] \mid (p, v, q) \in F \text{ and } |v| = \ell\} & \text{otherwise.} \end{cases}$$

The relation shows that the computation of values $LC[p]$, for all the states of M, is done by a simple depth-first traversal of the graph of the structure. As its number of states and its number of edges are linear (see Sections 2.2, 2.4, and 2.5) and since the access to the label length of an edge is done in constant time according to the representation described in Section 2.2, the computation of the table takes a time $O(|y|)$ (independent of the alphabet).

Once the precomputation of table LC is performed, the computation of $pos_y(x)$ is done by searching for $p = \delta(\textit{initial}(M), x)$ and then by computing $|y| - LC[p] - |x|$. We then obtain the same asymptotic execution time as for the membership problem, namely $O(|x| \times \log \operatorname{Card} \mathcal{A})$. Let us

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

note that if

$$end(initial(M), x) = \delta(initial(M), xw)$$

with w nonempty, the value of $pos_y(x)$ is then $|y| - LC[p] - |xw|$, which does not modify the asymptotic evaluation of the execution time.

The computation of the position of the last occurrence of x in y is solved in a similar way by considering the table SC defined by

$$SC[p] = sc_y(u),$$

with the notations above. The relation

$$SC[p] = \begin{cases} 0 & \text{if } p \in T, \\ \min\{\ell + SC[q] \mid (p, v, q) \in F \text{ and } |v| = \ell\} & \text{otherwise,} \end{cases}$$

shows that the precomputation of SC takes a time $O(|y|)$, and that the computation of $last\text{-}pos_y(x)$ then takes $O(|x| \times \log \operatorname{Card} \mathcal{A})$ time.

Lastly, for accessing the number of occurrences of x one precomputes a table NB defined by

$$NB[p] = \operatorname{Card}\{z \in \mathcal{A}^* \mid \delta(p, z) \in T\},$$

which is precisely the sought quantity when $p = end(initial(M), x)$. The linear precomputation results from the relation

$$NB[p] = \begin{cases} 1 + \sum_{(p,v,q)\in F} NB[q] & \text{if } p \in T, \\ \sum_{(p,v,q)\in F} NB[q] & \text{otherwise.} \end{cases}$$

Then, the number of occurrences of x is obtained by computing the state $p = end(initial(M), x)$ and by accessing to $NB[p]$, which is done in the same time as for the above operations.

This ends the proof. ∎

An argument similar to the last element of the preceding proof allows an effective computation of the number of factors of y, that is, of the size of $\operatorname{Fact}(y)$. For that, one evaluates the quantity $CS[p]$, for all states p of the automaton, by using the relation

$$CS[p] = \begin{cases} 1 & \text{if } \deg(p) = 0, \\ 1 + \sum_{(p,v,q)\in F}(|v| - 1 + CS[q]) & \text{otherwise.} \end{cases}$$

If $p = \delta(initial(M), u)$ for some factor u of y, $CS[p]$ is the number of factors of y starting with u. This gives a linear-time computation of $\operatorname{Card} \operatorname{Fact}(y) = CS[q_0]$ (q_0 initial state of the automaton), that is in time $O(|y|)$ independent of the alphabet, given the automaton.

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

LIST OF POSITIONS

Proposition 2.6.3. *Given the tree $\mathfrak{S}(y)$ or the automaton $\mathfrak{A}^c(y)$, the list L of positions of the occurrences of a factor x of y can be computed in time $O(|x| \times \log \operatorname{Card} \mathcal{A} + k)$ within memory space $O(|y|)$, where k is the number of elements in L.*

Proof. The tree $\mathfrak{S}(y)$ is first considered. Let us point out from Section 2.1 that a state q of the tree is a factor of y, and that, if it is terminal, its output is the position of the suffix occurrence of q in y (in this case q is a suffix of y and $output[q] = |y| - |q|$). The positions of occurrences of x in y are the positions of suffixes prefixed by x. One thus obtains these positions by seeking terminal states of the subtree rooted at $p = end(initial(M), x)$ (see Section 2.2). Exploration of this subtree takes a time proportional to its size and indeed to its number of terminal nodes since each node that is not terminal has at least two children by definition of the tree. Finally, the number of terminal nodes is precisely the number k of elements of the list L.

In short, the computation of the list requires the computation of p and then the traversal of the subtree. The first phase is carried out in time $O(|x| \times \log \operatorname{Card} \mathcal{A})$, the second in time $O(k)$, which gives the announced result when $\mathfrak{S}(y)$ is used.

A similar reasoning applies to $\mathfrak{A}^c(y)$. Let $p = end(initial(M), x)$ and let w be such that $\delta(initial(M), xw) = p$. Starting from p, we explore the automaton by memorizing the length of the current path (the length of an edge is that of its label). A terminal state q that is reached by a path of length ℓ corresponds to a suffix of length ℓ which therefore occurs at position $|y| - \ell$. Then, $|y| - \ell - |xw|$ is the position of an occurrence of x in y. The complete traversal takes a time $O(k)$ as its equivalent traversal of the subtree of $\mathfrak{S}(y)$ just described. We thus obtain the same running time as with the compact suffix tree. ∎

Notice that the computation of the lists of positions is obtained without preprocessing the automata. By the way, using the (noncompact) suffix automaton of y requires a preprocessing which creates shortcuts to superimpose the structure of $\mathfrak{A}^c(y)$ on it, if one wishes to obtain the same running time.

2.6.3. Transducer of positions

Some of the questions of locating factors within the word y can be described in terms of transducers, that is automata in which edges have an output in addition to outputs on states. As an example, the function pos_y is realized

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

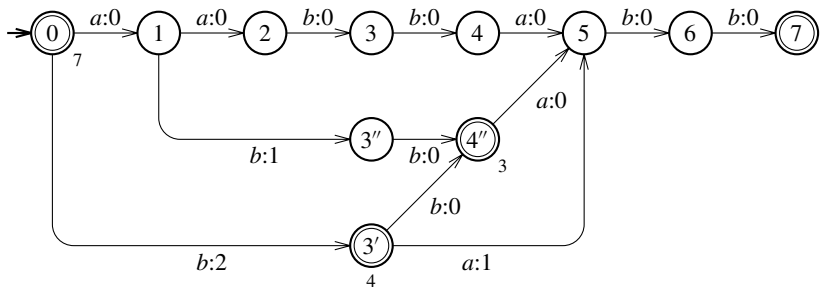


Figure 2.21. Transducer that realizes in a sequential way the function pos_y relative to $y = aabbabb$. Each edge is labelled by a pair (a, s) denoted by $a : s$, where a is the input of the edge and s its output. When scanning abb, the transducer produces the value 1 $(= 0 + 1 + 0)$, which is the position of the first occurrence of abb in y. The last state having output 3, one deduces that abb is a suffix at position 4 $(= 1 + 3)$ of y.

by the transducer of positions of y, denoted by $\mathcal{T}(y)$. Figure 2.21 gives an illustration of it.

The transducer $\mathcal{T}(y)$ is built on $\mathfrak{A}(y)$ by adding outputs to edges and by modifying the outputs associated with the terminal states. Edges of $\mathcal{T}(y)$ are of the form $(p, (a, s), q)$ where p, q are states and (a, s) the label of the edge. Letter $a \in \mathcal{A}$ is its input and integer $s \in \mathbb{N}$ is its output. The path

$$(p_0, (a_0, s_0), p_1), (p_1, (a_1, s_1), p_2), \ldots, (p_{k-1}, (a_{k-1}, s_{k-1}), p_k)$$

of the transducer has as input label the word $a_0a_1 \cdots a_{k-1}$, concatenation of input labels of edges of the path, and for output the sum $s_0 + s_1 + \cdots + s_{k-1}$.

The transformation of $\mathfrak{A}(y)$ into $\mathcal{T}(y)$ is done as follows. When (p, a, q) is an edge of $\mathfrak{A}(y)$ it becomes the edge $(p, (a, s), q)$ of $\mathcal{T}(y)$ with output

$$s = \textit{end-pos}_y(q) - \textit{end-pos}_y(p) - 1,$$

which is also

$$LC[p] - LC[q] - 1$$

with the notation LC used in the proof of Proposition 2.6.2. The output associated with a terminal state p is defined as $LC[p]$. The proof of Proposition 2.6.2 shows how to compute table LC from which one deduces a computation of outputs associated with edges and terminal states. The transformation is thus carried out in linear time.

Proposition 2.6.4. *Let u be the input label of a path starting at the initial state of the transducer $\mathcal{T}(y)$. Then, the output of the path is $pos_y(u)$.*

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

Moreover, if the end of the path is a terminal state having output t, u is a suffix of y and the position of this occurrence of u in y is $pos_y(u) + t$ $(= |y| - |u|)$.

Proof. We prove it by recurrence on the length of u. The first step of the recurrence, for $u = \varepsilon$, is immediate. Let us suppose that $u = va$ with $v \in \mathcal{A}^*$ and $a \in \mathcal{A}$. The output of the path having input label va is $r + s$, where r and s are respectively the outputs corresponding to inputs v and a. By the recurrence hypothesis, we have $r = pos_y(v)$. By definition of labels in $\mathcal{T}(y)$, we have

$$s = end\text{-}pos_y(u) - end\text{-}pos_y(v) - 1.$$

Therefore the output associated with u is

$$pos_y(v) + end\text{-}pos_y(u) - end\text{-}pos_y(v) - 1,$$

or also, since $end\text{-}pos_y(w) = pos_y(w) + |w| - 1$,

$$pos_y(u) + |u| - |v| - 1,$$

which is $pos_y(u)$ as expected. This finishes the proof of the first part of the statement.

If the end of the considered path is a terminal state, its output t is, by definition, $LC[u]$, which is $|y| - end\text{-}pos_y(u) - 1$ or $|y| - pos_y(u) - |u|$. Therefore $pos_y(u) + t = |y| - |u|$, which is the position in y of the suffix u as announced. ∎

The existence of the transducer of positions just described shows that the position of a factor in y can be computed sequentially, while reading the factor. The computation is even done in real time when transitions are performed in constant time.

2.7. Finding regularities

2.7.1. Repetitions

In this section we examine two questions concerning repetitions of factors within the text y. There are two dual problems that are solved efficiently by using a suffix tree or suffix automaton:

- Compute longest repeated factors of y.
- Find shortest factors having few occurrences in y.

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

These questions are parameterized by an integer k which bounds the number of occurrences.

Longest repetition Given an integer k, $k > 1$, find a longest word occurring at least k times in y.

Let $\mathfrak{A}(y)$ be the suffix automaton of y. If the table NB defined in the proof of Proposition 2.6.2 is available, the problem of the longest repetition remains of finding the states p of $\mathfrak{A}(y)$ which are the deepest in the automaton and for which $NB[p] \geq k$. The labels of longest paths from the initial state to ps are then solutions of the problem.

Indeed the solution comes without the use of table NB because values in the table do not need to be stored. We show how this is done for the instance of the problem with $k = 2$. One simply seeks a state (or all states), as deep as possible, that satisfies one of the two conditions:

- at least two edges leave p,
- an edge leaves p and p is a terminal state.

State p is then a fork and it is found by a mere traversal of the automaton. Proceeding in this way, no preliminary treatment of $\mathfrak{A}(y)$ is necessary and nevertheless the linear computing time is preserved. One can note that the execution time does not depend on the branching time in the automaton because no transition is executed, the search only traverses existing edges.

The two descriptions above are summarized in the following proposition.

Proposition 2.7.1. *Given one of the automata $\mathfrak{S}(y)$, $\mathfrak{A}(y)$, or $\mathfrak{A}^c(y)$, computing a longest repeated factor of y can be done in time and space $O(|y|)$.* ∎

The second problem deals with searching for a marker. A factor of y is so called when it marks a small number of positions on y.

Marker Given an integer k, $k > 1$, find a shortest word having less than k occurrences in y.

The use of a suffix automaton provides a solution to the problem of the same vein as that of the solution to the longest repetition problem. It amounts to finding, in the automaton, a state that is as close as possible to the initial state and that is the origin of less than k paths to a terminal state. Contrary to the above situation, however, a state associated with a marker is not necessarily a fork, but this has no effect on the solution. Again, a simple

traversal of the automaton solves the question, which gives the following result.

Proposition 2.7.2. *Given one of the automata* $\mathfrak{S}(y)$, $\mathfrak{A}(y)$, *or* $\mathfrak{A}^c(y)$, *the computation of a marker in* y *can be carried out in time and space* $O(|y|)$. ∎

2.7.2. Forbidden words

Searching for forbidden words is a reverse question to finding repetitions. It intervenes in the description of a certain type of text compression algorithms.

A word $u \in \mathcal{A}^*$ is called a *forbidden word* in the word $y \in \mathcal{A}^*$ if it is not a factor of y. And u is called a *minimal forbidden word* if in a supplement all its own proper factors are factors of y. In other words, the minimality relates to the ordering "is a factor of". This concept is in fact more relevant than the preceding one. We denote by $I(y)$ the set of minimal forbidden words in y.

One can notice that

$$u = u[0\,..\,k-1] \in I(y)$$

if and only if

u is not a factor of y but $u[0\,..\,k-2]$ and $u[1\,..\,k-1]$ are factors of y,

which results in the equality

$$I(y) = (\mathcal{A} \cdot \mathrm{Fact}(y)) \cap (\mathrm{Fact}(y) \cdot \mathcal{A}) \cap (\mathcal{A}^* \setminus \mathrm{Fact}(y)).$$

The equality shows in particular that the language $I(y)$ is finite. It is thus possible to represent $I(y)$ by a trie in which only the external nodes are terminal because of the minimality of words.

The algorithm ForbiddenWords, whose code is given below, builds the trie accepting $I(y)$ from the automaton $\mathfrak{A}(y)$. Figure 2.22 shows the example of the trie of forbidden words of *aabbabb*, obtained from the automaton of Figure 2.13. In the algorithm, the queue is used to traverse the automaton $\mathfrak{A}(y)$ in a width-first manner.

```
ForbiddenWords(𝔄(y))
 1  M ← NewAutomaton()
 2  L ← EmptyQueue()
 3  Enqueue(L, (initial(𝔄(y)), initial(M)))
 4  while not FileIsEmpty(L) do
 5        (p, p′) ← Dequeue(L)
```

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

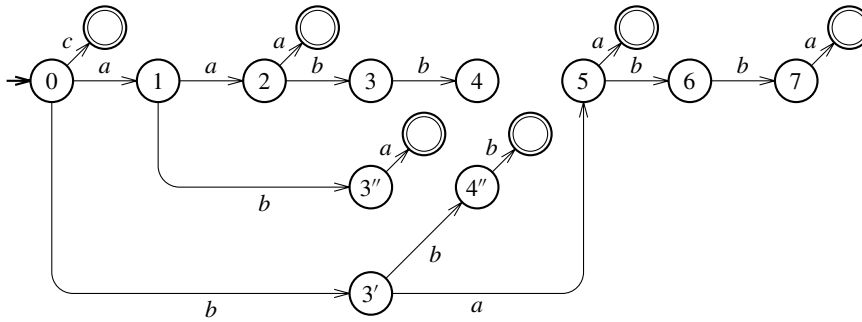


Figure 2.22. Trie of minimal forbidden words of the word *aabbabb* on the alphabet $\{a, b, c\}$, such as it is built by algorithm FORBIDDEN. Non-terminal states are those of automaton $\mathfrak{A}(aabbabb)$ of Figure 2.13. Note that states 3 and 4 as well as the edges reaching them can be removed. The forbidden word *babba*, recognized by the tree, is minimal because *babb* and *abba* are factors of *aabbabb*.

```
 6      for each a ∈ A do
 7          if TARGET(p, a) is undefined and
              (p = initial(A(y)) or TARGET(f[p], a) is defined) then
 8              q' ← NEWSTATE()
 9              terminal(q') ← TRUE
10              adj[p'] ← adj[p'] ∪ {(a, q')}
11          elseif TARGET(p, a) is defined and
              TARGET(p, a) not reached yet then
12              q' ← NEWSTATE()
13              adj[p'] ← adj[p'] ∪ {(a, q')}
14              ENQUEUE(L, (TARGET(p, a), q'))
15  return M
```

Proposition 2.7.3. *For $y \in \mathcal{A}^*$, the algorithm* FORBIDDENWORDS *produces, from the automaton $\mathfrak{A}(y)$, a tree that accepts the language $I(y)$. The execution can be done in time $O(|y| \times \log \operatorname{Card} \mathcal{A})$.*

Proof. It is noticed that edges created at line 13 duplicate the edges of the spanning tree of shortest paths of the graph of $\mathfrak{A}(y)$, because the automaton is traversed in width-first order (the queue L is aimed at that). Other edges are created at line 10 and are of the form (p', a, q') with $p', q' \in T'$, denoting by T' the set of terminal states of M. Let us denote by δ' the transition function associated with the edges of M created by the algorithm. By construction, the word u for which $\delta'(initial(M), u) = p'$ is the shortest word that reaches the state $p = \delta(initial(\mathfrak{A}(y)), u)$ in $\mathfrak{A}(y)$.

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

We start by showing that any word recognized by the tree that the algorithm produces is a minimal forbidden word. Let ua be such a word, necessarily nonempty ($u \in \mathcal{A}^*, a \in \mathcal{A}$). By assumption, the edge (p', a, q') was created at line 10 and $q' \in T'$. If $u = \varepsilon$, we have $p' = \mathit{initial}(M)$ and we notice that, by construction, $a \notin \mathit{alph}(y)$; therefore ua is effectively a minimal forbidden word. If $u \neq \varepsilon$, let us write it bv with $b \in \mathcal{A}$ and $v \in \mathcal{A}^*$. The state

$$s = \delta(\mathit{initial}(\mathfrak{A}(y)), v)$$

satisfies $s \neq p$ because both $|v| < |u|$ and, by construction, u is the shortest word that satisfies $p = \delta(\mathit{initial}(M), u)$. Therefore $f[p] = s$, by definition of suffix links. Then, again by construction, $\delta(s, a)$ is defined, which implies that va is a factor of y. The word $ua = bva$ is thus minimal forbidden since bv, va are factors of y but ua is not a factor of y.

It is then shown conversely that any forbidden word is recognized by the tree built by the algorithm. Let ua be such a word, necessarily nonempty ($u \in \mathrm{Fact}(y)$, $a \in \mathcal{A}$). If $u = \varepsilon$, the letter a does not appear in y, and thus $\delta(\mathit{initial}(\mathfrak{A}(y)), a)$ is not defined. The condition at line 7 is met and causes an edge to be created which leads to the recognition of the word ua by the automaton M. If $u \neq \varepsilon$, let us write it as bv with $b \in \mathcal{A}$ and $v \in \mathcal{A}^*$. Let

$$p = \delta(\mathit{initial}(\mathfrak{A}(y)), u).$$

As v is a proper suffix of u and va is a factor of y while ua is not a factor of y, if we consider the state

$$s = \delta(\mathit{initial}(\mathfrak{A}(y)), v),$$

we have necessarily $p \neq s$ and thus $s = f[p]$ by definition of suffix links. The condition at line 7 is thus still satisfied in this case, and this has the same effect as above. In conclusion, ua is recognized by the tree created by the algorithm, which finishes the proof. ∎

An unexpected consequence of the preceding construction is an upper bound on the number of minimal forbidden words in a word.

Proposition 2.7.4. *A word $y \in \mathcal{A}^*$ of length $|y| \geq 2$ has no more than* $\mathrm{Card}\,\mathcal{A} + (2|y| - 3) \times (\mathrm{Card}\,\mathit{alph}(y) - 1)$ *minimal forbidden words. It has* $\mathrm{Card}\,\mathcal{A}$ *of them if $|y| < 2$.*

Proof. According to the preceding proposition the number of minimal forbidden words in y is equal to the number of terminal states of the trie $I(y)$, which is also the number of edges entering these states.

There are exactly $\mathrm{Card}\,\mathcal{A} - \alpha$ such edges coming out from the initial state, by noting $\alpha = \mathrm{Card}\,\mathit{alph}(y)$. There are at most α outgoing edges from

the unique state of $\mathfrak{A}(y)$ having no outgoing transition. From other states there are at most $\alpha - 1$ outgoing edges. Since, for $|y| \geq 2$, $\mathfrak{A}(y)$ has at most $2|y| - 1$ states (Proposition 2.4.1), we obtain

$$\operatorname{Card} I(y) \leq (\operatorname{Card} \mathcal{A} - \alpha) + \alpha + (2|y| - 3) \times (\alpha - 1),$$

which gives

$$\operatorname{Card} I(y) \leq \operatorname{Card} \mathcal{A} + (2|y| - 3) \times (\alpha - 1),$$

as announced.

Finally, we have $I(\varepsilon) = \mathcal{A}$ and, for $a \in \mathcal{A}$, $I(a) = (\mathcal{A} \setminus \{a\}) \cup \{aa\}$. Therefore $\operatorname{Card} I(y) = \operatorname{Card} \mathcal{A}$ when $|y| < 2$. ∎

2.8. Pattern matching machine

Suffix automata can be used like machines to locate occurrences of patterns. We consider in this section the suffix automaton $\mathfrak{A}(x)$ to implement the search for x (length m) in a word y (length n). The other structures, compact tree $\mathfrak{S}(x)$ and compact automaton $\mathfrak{A}^c(x)$, can be used as well.

The searching algorithm rests on considering a transducer with a failure function. The transducer computes sequentially the lengths ℓ_i defined below. It is built upon the automaton $\mathfrak{A}(x)$, and the failure function, used to cope with nonexplicitly defined transitions of the searching automaton, is nothing else than the suffix link f defined on states of the automaton. The principle of the searching method is standard. The search is carried out sequentially along the word y. Adaptation and analysis of the algorithm with the tree $\mathfrak{S}(x)$ are immediate although the suffix link function of this structure is not a failure function according to the precise meaning of this concept (see Problem 2.2.4).

The advantage brought by the algorithm to other methods based on failure functions lies in a bounded amount of time to treat a letter of y, together with a more direct analysis of its time complexity. The price for this improvement is a more important need for memory capacity intended to store the automaton instead of a simple table, although the space remains linear.

2.8.1. Lengths of common factors

The search for x is based on computing lengths of factors of x appearing at any position on y. More precisely, the algorithm computes, at any position i on y, $0 \leq i < n$, the length

$$\ell_i = \max\{|u| \mid u \in \operatorname{Fact}(x) \cap \operatorname{Suff}(y[0\,..\,i])\}$$

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$y[i]$	a	a	a	b	b	b	a	b	b	a	a	b	b	a	b	b	b
ℓ_i	1	2	2	3	4	2	3	4	5	4	2	3	4	5	6	7	2
p_i	1	2	2	3	4	$4''$	5	6	7	5	2	3	4	5	6	7	$4''$

Figure 2.23. Using the automaton $\mathfrak{A}(aabbabb)$ (see Figure 2.13), algorithm LENGTHSOFFACTORS determines the factors common to *aabbabb* and y. Values ℓ_i and p_i are the respective values of variables ℓ and p of the algorithm related to position i. At position 8 for example, $\ell_8 = 5$ indicates that the longest factor of *aabbabb* ending there has length 5; it is *bbabb*; the current state is 7. An occurrence of the pattern is detected when $\ell_i = 7 = |aabbabb|$, as it is at position 15.

of the longest factor of x ending at this position. The detection of occurrences of x follows the obvious remark:

$$x \text{ occurs at position } i - |x| + 1 \text{ on } y$$

if and only if

$$\ell_i = |x|.$$

The algorithm which computes the lengths $\ell_0, \ell_1, \ldots, \ell_{n-1}$ is given below. It uses the table L, defined on states of the automaton (Section 2.4), to reset the length of the current factor, after a traversal through a suffix link (line 8). The correction of this instruction is a consequence of Lemma 2.3.6. A simulation of the computation is given in Figure 2.23.

```
LENGTHSOFFACTORS(𝔄(x), y)
 1  (ℓ, p) ← (0, initial(𝔄(x)))
 2  for i ← 0 to n − 1 do
 3       if TARGET(p, y[i]) is defined then
 4            (ℓ, p) ← (ℓ + 1, TARGET(p, y[i]))
 5       else do   p ← f[p]
 6            while p is defined and TARGET(p, y[i]) is undefined
 7            if p is defined then
 8                 (ℓ, p) ← (L[p] + 1, TARGET(p, y[i]))
 9            else (ℓ, p) ← (0, initial(𝔄(x)))
10       output ℓ
```

Theorem 2.8.1. *The algorithm* LENGTHSOFFACTORS *applied to the automaton* $\mathfrak{A}(x)$ *and the word* y $(x, y \in \mathcal{A}^*)$ *produces the lengths* $\ell_0, \ell_1, \ldots, \ell_{|y|-1}$. *It makes less than* $2|y|$ *transitions in* $\mathfrak{A}(x)$ *and runs in time* $O(|y| \times \log \operatorname{Card} \mathcal{A})$ *and space* $O(|x|)$.

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

Proof. The correctness of the algorithm is proved by recurrence on the length of prefixes of y. We show more exactly that the equalities

$$\ell = \ell_i$$

and

$$p = \delta(\mathit{initial}(\mathfrak{A}(x)), y[i - \ell + 1 \,..\, i])$$

are invariants of the **for** loop, by noting δ the transition function of $\mathfrak{A}(x)$.

Let $i \geq 0$. The already-treated prefix has length i and the current letter is $y[i]$. It is supposed that the condition is satisfied for $i - 1$. Thus, $u = y[i - \ell \,..\, i - 1]$ is the longest factor of x ending at position $i - 1$ and $p = \delta(\mathit{initial}(\mathfrak{A}(x)), u)$.

Let w be the suffix of length ℓ_i of $y[0 \,..\, i]$. Let us first suppose $w \neq \varepsilon$; therefore w rewrites $v \cdot y[i]$ with $v \in \mathcal{A}^*$. Note that v cannot be longer than u because this would contradict the definition of u, and thus v is a suffix of u.

If $v = u$, $\delta(p, y[i])$ is defined and provides the next value of p. Moreover, $\ell_i = \ell + 1$. These two points correspond to the update of (ℓ, p) carried out at line 4, which shows that the condition is satisfied for i in this situation.

When v is a proper suffix of u, we consider the greatest integer k, $k > 0$, for which v is a suffix of $s_x^k(u)$ where s_x is the suffix function relative to x (Section 2.3). Lemma 2.3.6 implies that $v = s_x^k(u)$ and that the length of this word is $L_x(q)$ where $q = \delta(\mathit{initial}(\mathfrak{A}(x)), v)$. The new value of p is thus $\delta(q, y[i])$, and that of ℓ is $L_x(q) + 1$. It is done so by the instruction at line 8, since f and L respectively implement the suffix function and the length function of the automaton, and according to Proposition 2.4.5 which establishes the relation with function s_x.

When $w = \varepsilon$, this means that letter $y[i] \notin \mathit{alph}(x)$. It is thus necessary to re-initialize the pair (ℓ, p), which is done at line 9.

Finally, it is noted that the proof is also valid for the treatment of the first letter of y, which finishes the proof of the invariant condition and proves the correctness of the algorithm.

For the complexity, one notices that each transition done, successfully or not, leads to i being incremented or to a strict increase in the value of $i - \ell$. As each one of these two expressions varies from 0 to $|y|$, we deduce that the number of transitions done by the algorithm is no more than $2|y|$. Moreover, as the execution time of all the transitions is representative of the total execution time, it is $O(|y| \times \log \operatorname{Card} \mathcal{A})$.

The memory space necessary to run the algorithm is used mainly to store the automaton $\mathfrak{A}(x)$ which has size $O(|x|)$ according to Theorem 2.4.4. This gives the last stated result, and finishes the proof. ∎

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

2.8.2. Optimization of suffix links

Since the algorithm LengthsOfFactors works in a sequential way, it is natural to consider its delay, that is, the maximum time spent on a letter of y. One realizes immediately that it is possible to modify the suffix function in order to reduce this time.

We define, for each state p of $\mathfrak{A}(x)$,

$$Next(p) = \{a \in \mathcal{A} \mid \delta(p, a) \text{ is defined}\}.$$

Then, the new suffix link $\hat{f}$ is defined, for the p state of $\mathfrak{A}(x)$, by the relation:

$$\hat{f}[p] = \begin{cases} f[p] & \text{if } Next(p) \subset Next(f[p]), \\ \hat{f}[f[p]] & \text{else, if this value is defined.} \end{cases}$$

Note that the relation can leave the value of $\hat{f}[p]$ undefined. The idea of this definition comes from the fact that the link is used as a failure function: there is no need to go to $f[p]$ if $Next(f[p]) \subseteq Next(p)$.

Note that in the automaton $\mathfrak{A}(x)$ one always has

$$Next(p) \subseteq Next(f[p]).$$

So, we can reformulate the definition of $\hat{f}$ as:

$$\hat{f}[p] = \begin{cases} f[p] & \text{if } \deg(p) \neq \deg(f[p]), \\ \hat{f}[f[p]] & \text{else, if this value is defined.} \end{cases}$$

The computation of $\hat{f}$ can thus be performed in linear time by considering outgoing degrees (deg) of states in the automaton.

The optimization of the suffix link leads to a reduction of the delay of algorithm LengthsOfFactors. The time can be evaluated as the number of executions of the instruction at line 5. We get the following result, which shows that the algorithm treats the letters of y in real time when the alphabet is fixed.

Proposition 2.8.2. *When the algorithm* LengthsOfFactors *makes use of the suffix link $\hat{f}$ in place of f, the treatment of each letter of y takes a time $O(\text{Card}\, alph(x))$.*

Proof. The result is an immediate consequence of inclusions

$$Next(p) \subset Next(\hat{f}[p]) \subseteq \mathcal{A}$$

for each state p for which $\hat{f}[p]$ is defined. ∎

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

2.8.3. Search for conjugates

The sequence of lengths $\ell_0, \ell_1, \ldots, \ell_{n-1}$ of the preceding section provides very rich information on resemblances between the words x and y. It can be exploited in various ways by algorithms comparing words. It authorizes for example an efficient computation of $LCF(x, y)$, the maximum length of factors common to x and y. This is done in linear time on a bounded alphabet. This quantity intervenes for example in the definition of the distance between words:

$$d(x, y) = |x| + |y| - 2LCF(x, y).$$

We are interested in searching for conjugates (or rotation) of a word within a text. The solution put forward in this section is another consequence of the length computation described in the previous section. Let us recall that a conjugate of word x is a word of the form $v \cdot u$ for which $x = u \cdot v$.

Searching for conjugates Let $x \in \mathcal{A}^*$. Locate all the occurrences of conjugates of x (of length m) occurring in a word y (of length n).

A first solution applies a classical algorithm to search for a finite set of words after having built the trie of conjugates of x. The search time is then proportional to n (on a fixed alphabet), but the trie can have a quadratic size $O(n^2)$, as can be the size of the (noncompact) suffix trie of x.

The solution based on the use of a suffix automaton does not have this disadvantage while preserving an equivalent execution time. The technique is derived from the computation of lengths done in the preceding section. For this purpose, we consider the suffix automaton of the word $x \cdot x$, by noting that every conjugate of x is a factor of $x \cdot x$. One could even consider the word $x \cdot w\mathcal{A}^{-1}$ where w is the primitive root of x, but that does not change the following result.

Proposition 2.8.3. *Let $x, y \in \mathcal{A}^*$. Locating the conjugates of x in y can be done in time $O(|y| \times \log \operatorname{Card} \mathcal{A})$ within a memory space $O(|x|)$.*

Proof. We consider a variant of algorithm LengthsOfFactors that produces the positions of the occurrences of factors having a length not smaller than a given integer k. The transformation is immediate since at each stage of the algorithm the length of the current factor is stored in variable ℓ.

The modified algorithm is applied to the automaton $\mathfrak{A}(x^2)$ and the word y with parameter $k = |x|$. The algorithm thus determines factors of length $|x|$ of x^2 which appear in y. The conclusion follows, noting that factors of length $|x|$ of x^2 are conjugate of x, and that all conjugates x appear in x^2. ∎

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

Problems

Section 2.2

2.2.1 Check that the execution of SUFFIXTREE(a^n) ($a \in \mathcal{A}$) takes a time $O(n)$. Check that the execution time of SUFFIXTREE(y) is $\Omega(n \log n)$ when Card $alph(y) = |y| = n$.

2.2.2 How many nodes are there in the compact suffix tree of a de Bruijn word? How many for a Fibonacci word? Same question for their compact and noncompact suffix automata.

2.2.3 Let $\mathcal{T}_{k,l}(y)$ be the compact trie that accepts the factors of word y that have a length ranging between the two natural integers k and ℓ ($0 \leq k \leq \ell \leq |y|$). Design an algorithm, to build $\mathcal{T}_{k,l}(y)$, that uses a memory space proportional to the size of the tree (and not $O(|y|)$), and that runs in the same asymptotic time as the construction of the suffix tree of y.

2.2.4 Design an algorithm for the computation of $LCF(x, y)$ ($x, y \in \mathcal{A}^*$), maximum length of factors common to x and y, based on the tree $\mathfrak{S}(x \cdot c \cdot y)$, where $c \in \mathcal{A}$ and $c \notin alph(x \cdot y)$. What is the time and space complexity of the computation? Compare with the solution in Section 2.8.

2.2.5 Give a bound on the number of cubes of primitive words occurring in a word of length n. Same problem for squares. (Hint: use the suffix tree of the word.)

2.2.6 Design an algorithm for the fusion of two suffix trees.

2.2.7 Describe a linear time and space algorithm (on a fixed alphabet) for the construction of the suffix tree of a finite set of words.

Section 2.3

2.3.1 Let y be a word in which the last letter does not appear elsewhere. Show that $\mathfrak{F}(y)$, the minimal deterministic automaton accepting the factors of y, has the same states and same edges as $\mathfrak{A}(y)$ (only the terminal states differ).

2.3.2 Give the precise number of states and edges in the factor automaton $\mathfrak{F}(y)$.

Section 2.4

2.4.1 Design an on-line algorithm for the construction of the factor automaton $\mathfrak{F}(y)$. The algorithm should run in linear time and space on a finite and fixed alphabet.

2.4.2 Design a linear-time algorithm (on a fixed alphabet) for the construction of the suffix automaton of a finite set of words.

Section 2.5

2.5.1 Describe an algorithm for constructing $\mathfrak{A}^c(y)$ from $\mathfrak{S}(y)$.

2.5.2 Describe an algorithm for constructing $\mathfrak{A}^c(y)$ from $\mathfrak{A}(y)$.

2.5.3 Write in details the code of the algorithm for the direct construction of $\mathfrak{A}^c(y)$.

2.5.4 Design an on-line algorithm for constructing $\mathfrak{A}^c(y)$.

Section 2.7

2.7.1 Let $k > 0$ be an integer. Implement an algorithm, based on one of the automata of suffixes of $y \in \mathcal{A}^*$, which determines factors that appear at least k times in y.

2.7.2 For $y \in \mathcal{A}^*$, design an algorithm for computing the maximum length of factors of y which have two nonoverlapping occurrences (that is if u is such a factor, it appears in y at two positions i and j such as $i + |u| \leq j$).

2.7.3 It is said that a language $M \subseteq \mathcal{A}^*$ avoids a word $u \in \mathcal{A}^*$ if u is not a factor of any word of M. Let M be the language of words that avoid all the words of a finite set $I \subseteq \mathcal{A}^*$. Show that M is accepted by a finite automaton. Give an algorithm that builds an automaton accepting M given the trie of I.

2.7.4 Design a construction of the automaton $\mathfrak{F}(y)$ given the trie of forbidden words $I(y)$.

Section 2.8

2.8.1 Provide an infinite family of words for which each word has a trie of its conjugates that is of quadratic size.

2.8.2 Design an algorithm for locating conjugates of x in y ($x, y \in \mathcal{A}^*$), given the tree $\mathfrak{S}(x \cdot x \cdot c \cdot y)$, where $c \in \mathcal{A}$ and $c \notin alph(x \cdot y)$. What is the complexity of the computation?

Notes

The concept of position tree is due to Weiner (1973) who presented an algorithm to compute its compact version. The algorithm of Section 2.2 is from McCreight (1976). A strictly sequential version of the suffix tree construction was described by Ukkonen (1995).

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press

For questions referring to formal languages, like concepts of syntactic congruences and minimal automata, one can refer to the books of Berstel (1979) and Pin (1986).

The suffix automaton of a text with unmarked terminal states is also known by the acronym DAWG, *Directed Acyclic Word Graph*. Its linearity was discovered by Blumer, Blumer, Enhrenfeucht, Haussler, and McConnel (1983) who gave a linear construction of it on a fixed alphabet (see also Blumer *et al.* 1985). The minimality of the structure as an automaton is due to Crochemore (1986), who showed how to build within the same complexity the factor automaton of a text (see Problems 2.3.1, 2.3.2, and 2.4.1).

The notion of compact suffix automaton appears in Blumer, Ehrenfeucht, and Haussler (1989). An algorithm for compacting suffix automata, as well as a direct construction of compact suffix automata, is presented in Crochemore and Vérin (1997). An on-line construction of compact suffix automata has been designed by Inenaga, Hoshino, Shinohara, Takeda, Arikawa, Mauri, and Pavesi (2001).

For the average analysis of sizes of the various structures presented in the chapter, one can refer to Szpankowski (1993b) and to Jacquet and Szpankowski (1994), who corrected a previous analysis by Blumer *et al.* (1989), extended by Raffinot (1997). These analyses rely on methods described in the book of Sedgewick and Flajolet (1995).

On special integer alphabets, Farach (1997) has designed a linear time construction of suffix trees.

Indexes can also be realized efficiently with the use of suffix arrays. This data structure may be viewed as an implementation of a suffix tree. The notion has been introduced by Manber and Myers (1993) who designed the first efficient algorithms for its construction and use. On special integer alphabets, a suffix array can be built in linear time by three independent algorithms provided by Kärkkäinen and Sanders (2003), Kim, Sim, Park, and Park (2003), and Ko and Aluru (2003).

The concept of index is strongly used in questions related to data retrieval techniques. One can refer to the book of Baeza-Yates and Ribero-Neto (1999) to go deeper into the subject, or to that of Salton (1989). Apostolico (1985) describes several algorithmic applications of suffix trees that often apply to other suffix structures.

Personal searching systems, or indexes used by search engines, often use simpler techniques like the constitution of lexicons of rare words or k-grams (that is factors of length k) with k relatively small.

The majority of topics covered in this chapter is classical in string algorithmics. The book of Gusfield (1997) contains a good number of problems, and especially those grounded on questions in computational

molecular biology, whose algorithmic solutions rest on the use of data structures for indexes, including questions related to repetitions.

Forbidden words of Section 2.7.2 are used in the DCA compression method of Crochemore, Mignosi, Restivo, and Salemi (2000).

The use of suffix automata as searching machines is due to Crochemore (1987). Using suffix trees for this purpose produces an immediate but less efficient solution (see Problem 2.2.4).

https://doi.org/10.1017/CBO9781107341005.003 Published online by Cambridge University Press