4

String Algorithms

In this chapter we will present some basic algorithms and data structures for dealing with strings. This includes methods for the exact comparison of strings and for finding repetitive structures in strings. While some of the presented approaches can be used directly for problems arising in molecular biology, for example, for the search for repeats in a DNA sequence, other methods presented here will serve as subprocedures for solving more complex problems in later chapters of this book.

The chapter is organized as follows: We introduce the basic problem of string matching in Section 4.1, and we present a simple algorithm for it. Sections 4.2 and 4.3 contain improved algorithmic approaches to the string matching problem. Section 4.4 is devoted to the presentation of an important data structure for handling strings, called suffix tree. We will consider in Section 4.5 some variants of the string matching problem that can be solved efficiently using suffix trees. Subsection 4.5.4 deals with another basic problem: the search for identical substrings in a given string. Finally, Section 4.6 introduces another powerful tool for dealing with strings, the suffix array. The chapter concludes with a summary in Section 4.7 and some bibliographic notes in Section 4.8.

4.1 The String Matching Problem

The most elementary problem when dealing with strings is probably the string matching problem, which consists of finding a (usually short) string, the *pattern* , as a substring in a given (usually very long) string, the *text* . This problem arises in many different applications, also outside of molecular biology, for example, in text editors or search engines for the Internet. An important application in molecular biology is the search for a newly sequenced DNA fragment, possibly coding for a gene, in a genome database. Although one usually prefers to allow a certain error rate for this search, an algorithm for

Algorithm 4.1 Naive string matching algorithm

Input: A pattern $p = p_1 \dots p_m$ and a text $t = t_1 \dots t_n$.

```
I := ∅
for j := 0 to n − m do
    i := 1
    while p_i = t_{j+i} and i ⩽ m do
        i := i + 1
    if i = m + 1 then {p_1 ... p_m = t_{j+1} ... t_{j+m}}
        I := I ∪ {j + 1}
```

Output: The set I of positions, where an occurrence of p as a substring in t starts.

the exact string matching problem can be used as a subroutine, as we will see in the context of the FASTA method in Subsection 5.2.1.

First, we give a formal definition of the string matching problem.

Definition 4.1. *Let Σ be an arbitrary alphabet. The (exact)* string matching problem *is the following computing problem:*

Input: Two strings $t = t_1 \dots t_n$ and $p = p_1 \dots p_m$ over Σ.
Output: The set of all positions in the text t, where an occurence of the pattern p as a substring starts, i.e., a set $I \subseteq \{1, \dots, n - m + 1\}$ of indices, such that $i \in I$ if and only if $t_i \dots t_{i+m-1} = p$.

The first naive approach for solving the string matching problem consists of sliding a window with the pattern p over the text t, and testing for each position of the window whether this substring of the text coincides with the pattern. More formally speaking, the algorithm tests for each position $i \in \{1, \dots, n - m\}$ if the condition $t_i \dots t_{i+m} = p$ holds (see Algorithm 4.1).

Figure 4.1 shows an example of the work of the naive string matching algorithm. In the worst case, this algorithm needs m comparisons for each i, which sums up to an overall running time in $O(m \cdot (n - m))$. The strings $t = a^n$ and $p = a^m$ make up a worst-case example for this algorithm. Even if we consider a modification of the algorithm that outputs only the first position j where $t_j \dots t_{j+m-1} = p$ holds, the naive string matching algorithm has a running time in $O(m \cdot (n-m))$, as the input instance $t = a^{n-1}b$ and $p = a^{m-1}b$ shows.

Our goal in the following is to find more efficient methods for solving the string matching problem. We will reach this goal by exploiting the structure of the pattern or of the text in order to save some comparisons. This general idea is illustrated by the following example.

Example 4.1. Let $t = ababb$ and $p = abb$. When comparing p with $t_1t_2t_3$, we recognize that $p_1 = t_1$ and $p_2 = t_2$, but $p_3 \neq t_3$. Since we know that $p_1 \neq p_2$, $p_2 = t_2$ implies that shifting the pattern by one position cannot be successful. ◊

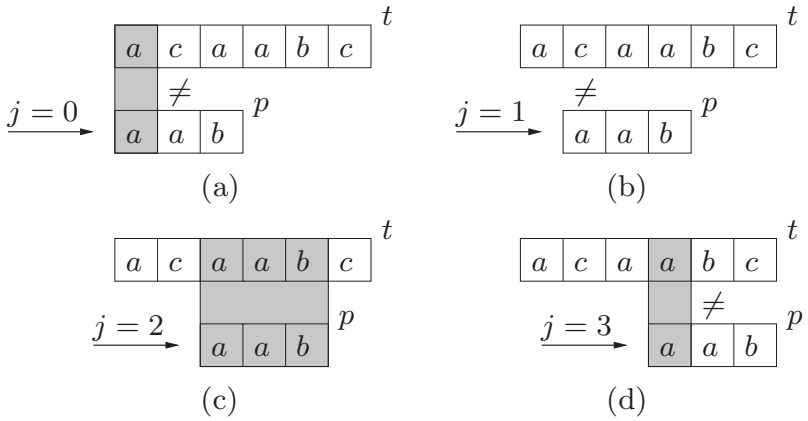

Fig. 4.1. An example for the work of the naive string matching algorithm

In the following sections we will present several different algorithms which use some preprocessing before starting the actual comparison of the strings. We will distinguish two classes of methods, depending on whether the preprocessing is done on the pattern p or on the text t. We will start with some methods using a preprocessing of p. In Section 4.4 we will then present the concept of string matching via suffix trees, which is based on preprocessing the text.

4.2 String Matching Automata

With the first approach, which we will follow in this section, we will show that, after a clever preprocessing of the pattern, one scan of the text from left to right will suffice to solve the string matching problem. Furthermore we will see that the preprocessing can also be realized efficiently; it is possible in time in $O(|p| \cdot |\Sigma|)$, where Σ is the alphabet over which the text t and the pattern p are given. This approach is based on the concept of finite automata. We will now give a very short definition of finite automata, as much as is needed to understand the string matching method. A more detailed presentation can be found, for example, in the textbooks by Hromkovič [104] or Hopcroft et al. [103].

Informally speaking, a finite automaton can be described as a machine that reads a given text once from left to right. At each step, the automaton is in one of finitely many internal states, and this internal state can change after reading every single symbol of the text, depending only on the current state and the last symbol read. By choosing the state transitions appropriately, one can determine from the current state after reading a symbol whether the text contains the pattern as a suffix. This means that a special automaton can find all positions in the text t where the pattern p ends.

Definition 4.2. *A* finite automaton *is a quintuple* $M = (Q, \Sigma, q_0, \delta, F)$*, where*

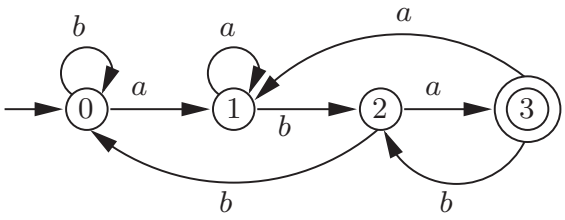

Fig. 4.2. A finite automaton for the string matching problem with the pattern $p = aba$. The states are shown as circles and the transition function is given by arrows labeled with the corresponding symbols from the alphabet Σ. The accepting state is marked by a double circle and the initial state is marked by an incoming unlabeled arrow

- *Q is a finite set of states,*
- *Σ is an input alphabet,*
- *$q_0 \in Q$ is the initial state,*
- *$F \subseteq Q$ is a set of accepting states , and*
- *$\delta : Q \times \Sigma \to Q$ is a transition function describing the transitions of the automaton from one state to another.*

We define the extension $\hat{\delta}$ of the transition function to strings over Σ by $\hat{\delta}(q, \lambda) = q$ and $\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$ for all $q \in Q, a \in \Sigma$ and $x \in \Sigma^*$. Thus, $\hat{\delta}(q, x)$ is the state M reaches from the state q by reading x.

We say that the automaton M accepts *the string $x \in \Sigma^*$, if $\hat{\delta}(q_0, x) \in F$.*

To solve the string matching problem we construct a finite automaton, called *string matching automaton*, that, for a given pattern $p = p_1 \dots p_m$, accepts all texts ending with the pattern p. We first illustrate the idea of the construction with an example.

Example 4.2. We construct a string matching automaton M for the pattern $p = aba$. We define $M = (Q, \Sigma, q_0, \delta, F)$, where $\Sigma = \{a, b\}$, $Q = \{0, \dots, 3\}$, $q_0 = 0$, $F = \{3\}$, and δ is given by the following table:

state	input symbol	following state
0	a	1
0	b	0
1	a	1
1	b	2
2	a	3
2	b	0
3	a	1
3	b	2

The automaton M is shown graphically in Figure 4.2.

On the input $t = bababaa$ the automaton passes through the following states:

Read prefix	state of the automaton
λ	0
b	0
ba	1
bab	2
$b\boldsymbol{aba}$	**3**
$babab$	2
$bab\boldsymbol{aba}$	**3**
$bababaa$	1

We note that, after reading a prefix of t, the automaton is in the accepting state 3 if and only if the prefix ends with the pattern $p = aba$. ◊

In the following, we describe a systematic way to construct a finite automaton for any given pattern p that solves the string matching problem for p and an arbitrary text t, based on the method shown in the above example.

The idea behind this string matching automaton is the following: For a pattern p of length m we define a sequence of $m + 1$ states, connected by a path of transitions labeled with p. In addition to these transitions, which are directed from the initial state to the accepting state, we add outgoing transitions for every state labeled with the respective missing symbols from Σ. These transitions point in the backward direction or from a state to itself. The endpoints of these transitions can be determined from the overlap structure of the pattern with itself.

To describe the overlap structure, we need a variant of the overlap of two strings as defined in Section 3.1.

Definition 4.3. *Let s, t be strings. If there exist some strings x, y, and z from Σ^* satisfying the conditions*

(i) $s = xy$,
(ii) $t = yz$, *and*
(iii) $|y|$ *is maximal with (i) and (ii),*

then $\overline{Ov}(s,t) := y$ *is called the* generalized overlap *of s and t. We denote the length of* $\overline{Ov}(s,t)$ *by* $\overline{ov}(s,t)$.

The generalized overlap needed here differs from the overlap defined in Definition 3.5 in that any one of the strings s and t is allowed to be a substring of the other. Now we can determine the transition function of a string matching automaton as follows.

Definition 4.4. *Let $p = p_1 \dots p_m \in \Sigma^m$ for an arbitrary alphabet Σ. We define the* string matching automaton for p *as the finite automaton $M_p = (Q, \Sigma, q_0, \delta, F)$, where $Q = \{0, \dots, m\}$, $q_0 = 0$, $F = \{m\}$, and the transition function δ is defined by*

$$\delta(q, a) = \overline{ov}(p_1 \dots p_q a, p) \quad \text{for all } q \in Q \text{ and } a \in \Sigma.$$

Algorithm 4.2 Construction of a string matching automaton

Input: A pattern $p = p_1 \dots p_m$ over an alphabet Σ.

for $q := 0$ **to** m **do**
  for all $a \in \Sigma$ **do**
    {Compute $\delta(q, a) = \overline{ov}(p_1 \dots p_q a, p)$}
    $k := \min\{m, q+1\} + 1$
    repeat
      $k := k - 1$
    until $p_1 \dots p_k = p_{q-k+2} \dots p_q a$
    $\delta(q, a) := k$

Output: The string matching automaton $M_p = (\{0, \dots, m\}, \Sigma, 0, \delta, \{m\})$.

Thus, the transition function of the string matching automaton for a pattern p can be constructed as shown in Algorithm 4.2.

In the following we will examine the connection between string matching automata and the calculation of overlaps more formally. We first need the following lemma.

Lemma 4.1. *Let Σ be an alphabet and let $n, m \in \mathbb{N}$. Let $x = x_1 \dots x_n \in \Sigma^n, y = y_1 \dots y_m \in \Sigma^m$, and $a \in \Sigma$. If $i = \overline{ov}(x, y)$, then*

$$\overline{ov}(xa, y) = \overline{ov}(y_1 \dots y_i a, y).$$

Proof. We start by proving the following inequality:

$$\overline{ov}(xa, y) \leqslant \overline{ov}(x, y) + 1. \tag{4.1}$$

To prove Inequality (4.1) we distinguish two cases: If $\overline{ov}(xa, y) = 0$ holds, then the proposition is obvious. Thus, let $\overline{ov}(xa, y) = r > 0$. Then $y_1 \dots y_r$ is a suffix of xa, and thus $y_1 \dots y_{r-1}$ is a suffix of x. This implies $\overline{ov}(x, y) \geqslant r-1$, from which Inequality (4.1) follows.

We will now prove the claim of the lemma. From $i = \overline{ov}(x, y)$ we know that $x = x' y_1 \dots y_i$ for some $x' \in \Sigma^*$. Furthermore, $\overline{ov}(x' y_1 \dots y_i a, y) = \overline{ov}(y_1 \dots y_i a, y)$ holds, since we know that $\overline{ov}(x' y_1 \dots y_i a, y) \leqslant i + 1$ from Inequality (4.1). This immediately implies the claim. □

With Lemma 4.1 we are now able to prove that it is possible to compute the overlap of all prefixes of an arbitrary text with a pattern p using a string matching automaton.

Lemma 4.2. *Let $p = p_1 \dots p_m \in \Sigma^m$ be a pattern and let $M_p = (Q, \Sigma, q_0, \delta, F)$ be the string matching automaton for p. Let $t = t_1 \dots t_n \in \Sigma^n$ be an arbitrary text. Then, for all $i \in \{0, \dots, n\}$,*

$$\hat{\delta}(q_0, t_1 \dots t_i) = \overline{ov}(t_1 \dots t_i, p).$$

Proof. We will prove the claim by induction on i. For $i = 0$ the claim obviously holds, since $\hat{\delta}(q_0, \lambda) = q_0 = 0 = \overline{ov}(\lambda, p)$. For the induction step from i to $i+1$ we denote by q the state the automaton has reached after reading $t_1 \dots t_i$, i.e., $q = \hat{\delta}(q_0, t_1 \dots t_i)$. Then,

$$\begin{aligned}\hat{\delta}(q_0, t_1 \dots t_{i+1}) &= \delta(\hat{\delta}(q_0, t_1 \dots t_i), t_{i+1}) \\ &= \delta(q, t_{i+1}).\end{aligned}$$

Following the definition of the transition function of M_p, this implies

$$\hat{\delta}(q_0, t_1 \dots t_{i+1}) = \overline{ov}(p_1 \dots p_q t_{i+1}, p).$$

From the induction hypothesis we know that $q = \overline{ov}(t_1 \dots t_i, p)$, which together with Lemma 4.1 implies

$$\begin{aligned}\hat{\delta}(q_0, t_1 \dots t_{i+1}) &= \overline{ov}(p_1 \dots p_q t_{i+1}, p) \\ &= \overline{ov}(t_1 \dots t_i t_{i+1}, p).\end{aligned}$$

This completes the proof of the claim. □

Using Lemma 4.2 we are now able to prove that the string matching automaton for a pattern p really solves the string matching problem for an arbitrary text $t \in \Sigma^*$ and the pattern p.

Theorem 4.1. *Let $p = p_1 \dots p_m \in \Sigma^m$ be a pattern and let $M_p = (Q, \Sigma, q_0, \delta, F)$ be the string matching automaton for p. Let $t = t_1 \dots t_n \in \Sigma^n$ be an arbitrary text. Then, for all $i \in \{1, \dots, n\}$,*

$$p \text{ is a suffix of } t_1 \dots t_i \iff \hat{\delta}(q_0, t_1 \dots t_i) \in F.$$

Proof. The claim immediately follows from Lemma 4.2 and the fact $F = \{m\}$. □

This theorem enables us to solve the string matching problem using finite automata, as shown in Algorithm 4.3.

Now that we have seen how the string matching automata can solve the string matching problem, we will analyze the time complexity of the method. The construction of the string matching automaton M_p for a given pattern $p = p_1 \dots p_m \in \Sigma^m$ by Algorithm 4.2 needs time in $O(|\Sigma| \cdot m^3)$, since the automaton for p has exactly $m+1$ states, and the computation of a transition needs $O(m^2)$ time for every pair of state and input symbol. Nevertheless, there is a method known to construct the string matching automaton M_p in time $O(|\Sigma| \cdot m)$, but it is quite technical. Therefore we will not present it here, but refer the reader to the references given in Section 4.8.

The application of M_p on a text $t = t_1 \dots t_n$ needs time in $O(n)$ since the automaton reads each symbol of t exactly once. Overall, the string matching problem for p and t is solvable in time $O(n + |\Sigma| \cdot m)$ using finite automata.

Algorithm 4.3 String matching with finite automata

Input: A text $t = t_1 \dots t_n \in \Sigma^n$ and a pattern $p = p_1 \dots p_m \in \Sigma^m$.

Compute the string matching automaton $M_p = (Q, \Sigma, q_0, \delta, F)$ using Algorithm 4.2.
$q := q_0$
$I := \emptyset$
for $i := 1$ **to** n **do**
  $q := \delta(q, t_i)$
  if $q \in F$ **then**
    $I := I \cup \{i - m + 1\}$

Output: The set I of those positions where p starts as a substring in t.

This solution to the string matching problem is especially efficient if the task is to search for a given pattern p of length m in k different texts $t^{(1)}, \dots, t^{(k)}$. In this case one has to construct the string matching automaton for p only once in time $O(|\Sigma| \cdot m)$, and after that the actual search within the text $t^{(i)}$ is possible in time $O(|t^{(i)}|)$.

4.3 The Boyer–Moore Algorithm

In this section we will present the Boyer–Moore algorithm for the string matching problem. It is based on a similar idea as the naive algorithm, but saves a lot of comparisons by using clever preprocessing.

Although the time complexity of the Boyer–Moore algorithm can in the worst case be as high as that of the naive algorithm, in many practical applications its running time usually is very good. Therefore, it is often used in practice.

The Boyer–Moore algorithm utilizes the following basic idea. Similarly to the naive algorithm, it shifts the pattern along the text from left to right and compares $p_1 \dots p_m$ with $t_{j+1} \dots t_{j+m}$ for $0 \leqslant j \leqslant n - m$. But, in contrast to the naive algorithm, the comparison is done from right to left, i.e., p_m is compared to t_{j+m} first. As soon as a position i occurs where p_i and t_{j+i} differ, the pattern is shifted to the right. The number of positions the pattern can be shifted without missing an occurrence of p in t is computed using two rules. To apply these rules efficiently, preprocessing of the pattern is necessary.

The two rules are as follows. The *bad character rule* says that the pattern can be shifted to the rightmost occurrence of the symbol t_{j+i} in the pattern. The *good suffix rule* claims that the pattern can be shifted to the next occurrence of the suffix $p_{i+1} \dots p_m$ in p. On an intuitive level, this means that as we have already matched a suffix of the pattern, we know the following symbols in t, and thus we may shift the pattern to a position where this particular sequence of symbols occurs again. Please note that the bad character rule

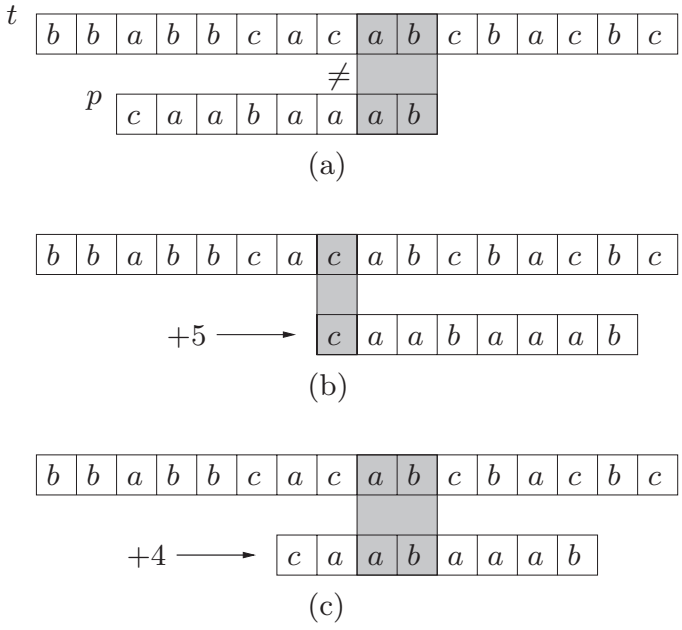

Fig. 4.3. An example for the application of the rules in the Boyer–Moore algorithm: (a) the comparison of the pattern p with a substring of the text t, the longest identical suffix of p is shaded grey; (b) the shift of the pattern for the next comparison according to the bad character rule; (c) the shift according to the good suffix rule

Algorithm 4.4 Preprocessing for the bad character rule

Input: A pattern $p = p_1 \dots p_m$ over an alphabet Σ.

for all $a \in \Sigma$ **do** $\beta(a) := 0$
for $i := 1$ **to** m **do** $\beta(p_i) := i$

Output: The function β.

might well propose a shift to the left, but such a shift will never be executed since the good suffix rule will always propose a shift to the right.

An example for the application of these rules is shown in Figure 4.3. At each step, the possible shift is calculated according to both rules and the shift with the larger value is realized.

We show that these rules are correct and efficiently implementable. Let us start with the bad character rule. For implementing this rule, it suffices to compute, in a preprocessing step, a function β that assigns the position of its last occurrence in p to each symbol from Σ (or the value 0, if the symbol does not occur in p). This computation can be done using Algorithm 4.4.

Algorithm 4.4 obviously has a running time in $O(m + |\Sigma|)$.

Lemma 4.3. *Let $p_{i+1} \dots p_m = t_{j+i+1} \dots t_{j+m}$ and $p_i \neq t_{j+i} = a$. Then the pattern can be shifted for the next comparison, according to the bad character rule, by $i - \beta(a)$ positions without missing an occurrence of p in the text t.*

Proof. For the proof we distinguish three cases:

(1) If the symbol a does not occur in p, i.e., if $\beta(a) = 0$ holds, the pattern p can obviously be shifted completely past t_{j+i}; thus, a shift by $i - \beta(a) = i$ positions is possible.
(2) If the last occurrence of a in p is at a position $k < i$, the pattern can be shifted by $i - k = i - \beta(a)$ positions to the right.
(3) If the last occurrence of a in p is at a position $k > i$, the bad character rule requires a shift of the pattern to the right by $i - k$ positions. Since $i - k$ is negative in this case, a shift to the left is indicated, which obviously is not helpful for the algorithm. Such a shift will never be executed since the good suffix rule always allows for a shift to the right, as we see below. □

We now consider the good suffix rule. For this rule, an efficient preprocessing step is also possible, for which we need the notion of suffix similarity of strings.

Definition 4.5. *Let s and t be two strings. We say that s is* suffix similar *to t, or $s \sim t$, if s is a suffix of t or t is a suffix of s.*

Intuitively, Definition 4.5 says that s and t can be aligned at their right ends such that all corresponding symbols are equal.

Using this notation, we reformulate the good suffix rule as follows:

Good Suffix Rule If $p_{i+1} \dots p_m = t_{j+i+1} \dots t_{j+m}$ and $p_i \neq t_{j+i}$, then shift the pattern for the next comparison to the right by $m - \gamma(i+1)$ positions, where

$$\gamma(i) = \max\{0 \leqslant k < m \mid p_i \dots p_m \sim p_1 \dots p_k\}.$$

We will show in the following that we can compute all values $\gamma(i)$ for all $2 \leqslant i \leqslant m$ in time $O(|\Sigma| \cdot m)$ using string matching automata.

We will do the computation of γ in two steps. By definition, the following holds for all $2 \leqslant i \leqslant m$:

$$\begin{aligned}\gamma(i) &= \max\{0 \leqslant k < m \mid p_i \dots p_m \sim p_1 \dots p_k\} \\ &= \max\{\max\{0 \leqslant k < m \mid p_i \dots p_m \text{ is a suffix of } p_1 \dots p_k\}, \\ &\qquad \max\{0 \leqslant k < m \mid p_1 \dots p_k \text{ is a suffix of } p_i \dots p_m\}\}.\end{aligned}$$

These two cases are shown in Figure 4.4.

We start with determining, for all $2 \leqslant i \leqslant m$, the value

$$\gamma'(i) = \max\{0 \leqslant k < m \mid p_i \dots p_m \text{ is a suffix of } p_1 \dots p_k\}.$$

This is equivalent to solving the following subproblem.

> Given a pattern $p = p_1 \dots p_m$, compute the last occurrence of each suffix $p_i \dots p_m$ of p within $p_1 \dots p_{m-1}$, for $2 \leqslant i \leqslant m$.

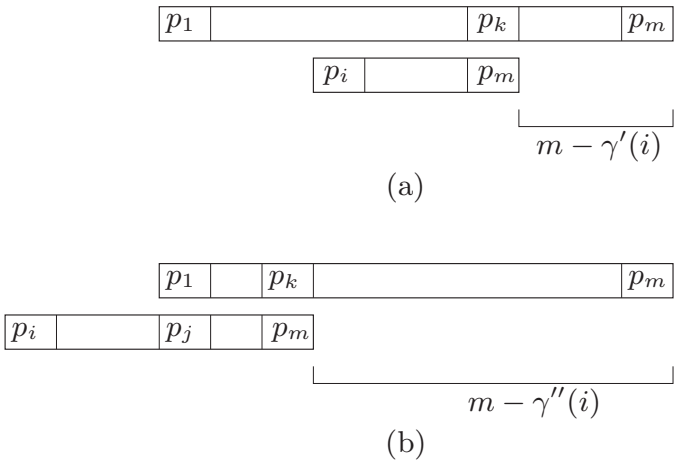

Fig. 4.4. (a) The case where $p_i \dots p_m$ is a substring of $p_1 \dots p_{m-1}$, where $\gamma'(i)$ denotes the last position $k \leqslant m-1$, such that $p_i \dots p_m$ is a suffix of $p_1 \dots p_k$. (b) The case where a suffix of $p_i \dots p_m$ is a prefix of $p_1 \dots p_{m-1}$, where $\gamma''(i)$ denotes the last position $k \leqslant m-1$, such that $p_1 \dots p_k$ is a suffix of $p_i \dots p_m$

To solve this subproblem we compute the function $\gamma' : \{2, \dots, m\} \to \{0, \dots, m-1\}$, where $\gamma'(i) = \max\{1 \leqslant k \leqslant m-1 \mid p_i \dots p_m = p_{k-m+i} \dots p_k\}$, if such a k exists, and $\gamma'(i) = 0$ otherwise.

To compute γ' using string matching automata, we reformulate the above problem.

> Given a pattern $p = p_1 \dots p_m$, compute the first occurrence of each prefix $p_m \dots p_i$ of $p^R = p_m \dots p_1$ in $p_{m-1} \dots p_1$ for $2 \leqslant i \leqslant m$.

This obviously is an equivalent formulation for all $1 \leqslant i \leqslant m-1$, $\gamma'(i) = \min\{1 \leqslant k \leqslant m-1 \mid p_m \dots p_i = p_k \dots p_{k-m+i}\}$, if such a k exists.

A first, naive approach for solving this problem could be to construct the string matching automata for $p_m \dots p_i$ for all $i \in \{2, \dots, m\}$ and to apply them to the text $p_{m-1} \dots p_1$. But this approach is far too time consuming, as it needs time in $O(|\Sigma| \cdot m^2)$. But the string matching automata for $p_m \dots p_i$ are very similar for all i. The next lemma shows that it suffices to construct the automaton for $p_m \dots p_2$.

Lemma 4.4. *Let $M_i = (\{0, \dots, m-i+1\}, \Sigma, \delta_i, 0, \{m-i+1\})$ be the string matching automaton for $p_m \dots p_i$ and let $t = t_1 \dots t_n$ be an arbitrary text. If $t_k \dots t_{k-m+i}$ is the first occurrence of $p_m \dots p_i$ in t, the following holds for all $1 \leqslant j \leqslant k-m+i$:*

$$\hat{\delta}_i(q_0, t_1 \dots t_j) = \hat{\delta}_2(q_0, t_1 \dots t_j). \tag{4.2}$$

If $p_m \dots p_i$ does not occur as a substring in t, Equation (4.2) holds even for all $1 \leqslant j \leqslant n$.

Proof. The claim of the lemma directly follows from the construction of the string matching automata. The outgoing transitions of the first $m-i+1$ states

are the same in both automata, M_i and M_2. To get into one of the additional states in M_2, the automaton must have reached the state $m - i + 1$ before. But this implies that the automaton has read the substring $p_m \dots p_i$. □

Since we are only interested in the first occurrence of each prefix $p_m \dots p_i$, Lemma 4.4 allows us to simulate the computation of all string matching automata M_i for $2 \leqslant i \leqslant m$ by one computation of M_2 by just storing the sequence of reached states while reading the text $p_{m-1} \dots p_1$. Let $q_0, q_{m-1}, \dots, q_1$ be this sequence of states. Then $\gamma'(i)$ can for all $2 \leqslant i \leqslant m$ be determined as $\gamma'(i) = \max\{j \mid q_j = i\}$ if the state i was reached while reading the string $p_{m-1} \dots p_1$, and as $\gamma'(i) = 0$ otherwise.

We will now show how we can use the sequence $q_0, q_{m-1}, \dots, q_1$ of states to compute for all $2 \leqslant i \leqslant m$ the value of

$$\gamma''(i) = \max\{0 \leqslant k < m \mid p_1 \dots p_k \text{ is a suffix of } p_i \dots p_m\},$$

and thus also the function γ we are actually interested in.

If the string matching automaton M_2 for $p_m \dots p_2$ ends in state $q_1 = j$ after reading the text $p_{m-1} \dots p_1$, then $p_m \dots p_j$ are the last symbols read due to the construction of the automaton, and j is minimal with this property. In other words, $p_m \dots p_j$ is a suffix of $p_{m-1} \dots p_1$, and $p_m \dots p_{j'}$ is not a suffix of $p_{m-1} \dots p_1$ for all $j' < j$. This means that $p_j \dots p_m$ is a prefix of $p_1 \dots p_{m-1}$ and that j is minimal with this property. Thus, we can set $\gamma''(i) = m - j + 1$ for all $2 \leqslant i \leqslant j$ (see Figure 4.4 (b)). For all other values of i, $\gamma''(i) = 0$ holds.

These considerations can be put together to yield Algorithm 4.5 for computing the function γ for the good suffix rule.

The construction of the string matching automaton needs time in $O(|\Sigma| \cdot m)$ as we have seen in Section 4.2; all further steps of Algorithm 4.5 can obviously be done in time $O(m)$. Thus, the preprocessing for the good suffix rule needs time in $O(|\Sigma| \cdot m)$ overall.

The Boyer–Moore algorithm, as shown in Algorithm 4.6, now combines the above preprocessing steps with the scanning of the text according to the two rules.

We now analyze the running time of the Boyer–Moore algorithm. The preprocessing of the two functions β and γ needs time in $O(|\Sigma| \cdot m)$ as shown above. After each comparison, the pattern is shifted according to these two rules. Since $\gamma(i) < m - 1$ holds for all i, the shift proposed by the good suffix rule is always positive. But in the worst case it is possible that the pattern is shifted exactly one position to the right in every step, and the computation of this shift might even need a comparison of the complete pattern in every step. This means that the Boyer–Moore algorithm has a worst-case running time in $O(|\Sigma| \cdot m + n \cdot m)$, which does not improve over the naive algorithm. On the other hand, the Boyer–Moore algorithm is quite fast in practice; the worst case occurs very rarely. By modifying the preprocessing, it is also possible to guarantee a worst-case running time in $O(n + m)$ (see the bibliographical notes at the end of this chapter).

Algorithm 4.5 Preprocessing for the good suffix rule

Input: A pattern $p = p_1 \dots p_m$ over an alphabet Σ.

1. Construct the string matching automaton $M = (Q, \Sigma, q_0, \delta, F)$ for $p_m \dots p_2$.
2. Determine the sequence $q_0, q_{m-1}, \dots, q_1$ of states M is traversing while reading the input $p_{m-1} \dots p_1$.
3. Compute the function γ':
   for $i := 2$ **to** m **do** $\gamma'(i) := 0$
   for $j := 1$ **to** $m - 1$ **do** $\gamma'(q_j) := j$
4. Compute the function γ'':
   for $i := 2$ **to** m **do**
   if $i \leqslant q_1$ **then**
   $\gamma''(i) := m - q_1 + 1$
   else
   $\gamma''(i) := 0$
5. Compute the function γ:
   for $i := 2$ **to** m **do** $\gamma(i) := \max\{\gamma'(i), \gamma''(i)\}$

Output: The function γ.

Algorithm 4.6 Boyer–Moore algorithm

Input: A pattern $p = p_1 \dots p_m$ and a text $t = t_1 \dots t_n$ over an alphabet Σ.

1. Compute from p the function β for the bad character rule.
2. Compute from p the function γ for the good suffix rule.
3. Initialize the set I of positions, where p starts in t, by $I := \emptyset$.
4. Shift the pattern p along the text t from left to right:
   $j := 0$
   $\gamma(m + 1) := m$ {Good suffix rule not applicable for $p_m = t_{j+m}$}
   while $j < n - m$ **do**
   {Compare $p_1 \dots p_m$ and $t_{j+1} \dots t_{j+m}$ starting from the right}
   $i := m$
   while $p_i = t_{j+i}$ **and** $i > 0$ **do**
   $i := i - 1$
   if $i = 0$ **then**
   $I := I \cup \{j\}$ {The pattern p starts in t at position j}
   {Compute the shift of the pattern according to the bad character rule and the good suffix rule}
   $j := j + \max\{i - \beta(t_{j+i}), m - \gamma(i + 1)\}$

Output: The set I of all positions j in t where the pattern p starts.