University of Konstanz | Design and Analysis of Algorithms
Department of Computer & Information Science | WS 2010/2011
Prof. Dr. Ulrik Brandes / Dr. Sabine Cornelsen / Melanie Badent / Natalie Indlekofer

Contents

7.2 Finite Automaton Matcher 1

7.3 The Knuth-Morris-Pratt Matcher 6

7.4 Suffix-Trees 9
- 7.4.1 Trie . . . 10
- 7.4.2 Suffix Tree . . . 11
- 7.4.3 Ukkonen's Online Algorithm . . . 12
  - 7.4.3.1 Making Rules 1 and 3 Implicit . . . 14
  - 7.4.3.2 Suffix Links . . . 16

7.5 The Algorithm of Boyer & Moore 20
- 7.5.1 Bad Character Rule . . . 20
- 7.5.2 Good Suffix Rule . . . 21
- 7.5.3 Preprocessing . . . 22
  - 7.5.3.1 Preprocessing for the Bad Character Rule . . . 22
  - 7.5.3.2 Preprocessing for the Good Suffix Rule . . . 23

Exact string matching considers the problem of finding all occurrences of a pattern $P[1,\ldots,m]$ in a text $T[1,\ldots,n], n \gg m$, i.e., to find all *valid shifts* (all occurrences of P in T with *shift* s), i.e., all $s, 0 \leq s \leq n-m$ such that

$$P[1\ldots m] = T[s+1,\ldots,s+m].$$

Throughout these lecture notes, we assume that $T[j,\ldots,i]$ denotes the empty string if $j > i$.

7.2 Finite Automaton Matcher

The Finite-Automaton-Matcher was first described by Aho et al. (1974). The idea of the algorithm is to use the information obtained from the last characters in the already read text. E.g., in

T:	c	b	a	c	b	c
P:	c	b	c			

the naive string matcher will move ahead by only one, even though we already have the information to decide that moving forward by one or two will not lead to a match. The FINITE-AUTOMATON-MATCHER constructs a finite automaton $\mathcal{A}_P$ for the pattern P such that

- $\mathcal{A}_P$ reads the text T once.
- P occurs with shift s in T if and only if $\mathcal{A}_P$ is in an accepting state after reading $T[s+m]$.

Definition 1. *A* finite automaton *is a tuple* $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ *with*

Q:	*finite set of* states,
Σ:	*finite* alphabet,
$F \subseteq Q$:	*set of* accepting states,
$q_0 \in Q$:	start state,
$\delta : Q \times \Sigma \longrightarrow Q$:	transition function.

The transition function δ extends to $Q \times \Sigma^*$ by setting

$$\delta(q, \epsilon) = q \qquad \text{and} \qquad \delta(q, wa) = \delta(\delta(q, w), a).$$

(Σ^* is the set of words *(strings) over Σ and ϵ is the empty word.)* $\mathcal{A}$ accepts *all words $w \in \Sigma^*$ with $\delta(q_0, w) \in F$.*

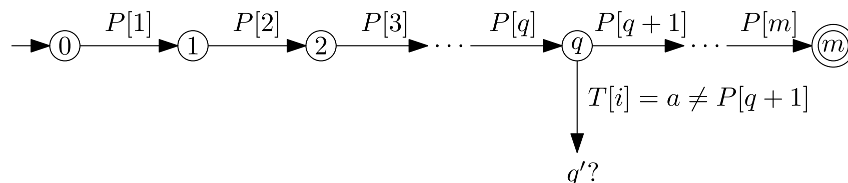
Example 1. *The finite automaton with $\Sigma = \{a, b\}$, $Q = \{0, 1\}$, $q_0 = 0$, $F = \{1\}$, and*

accepts all words ending with an odd number of a's.

The string matching automaton $\mathcal{A}_P$ for $P[1, \ldots, m]$ is defined as follows.

$$Q = \{0, \ldots, m\},\ q_0 = 0,\ F = \{m\}$$

and



δ should be defined such that $q' = \delta(q_0, T[1,\ldots,i])$ is the length of the largest prefix of P that is a suffix of the so far read text, i.e.

$$\begin{array}{rl} T[1,\ldots\ldots\ldots\ldots,i-1,i] & \text{known part of } T \\ P[1,\ldots,q'] & \end{array}$$

Definition 2.

- $x \in \Sigma^*$ *is a* suffix *of* $w \in \Sigma^*$, *if* $w = yx$ *for some* $y \in \Sigma^*$.
- $x \in \Sigma^*$ *is a* prefix *of* $w \in \Sigma^*$, *if* $w = xy$ *for some* $y \in \Sigma^*$.

Example 2. *The suffixes of "eggplant" are*

ϵ, t, nt, ant, lant, gplant, ggplant, eggplant.

A prefix of eggplant is, e.g., egg.

Definition 3. *Let*

$$\begin{array}{rrcl} \mathrm{suf}_P : & \Sigma^* & \longrightarrow & \{0,\ldots,m\} \\ & w & \longmapsto & \max\{k;\ P[1,\ldots,k] \textit{ is a suffix of } w\} \end{array}$$

Example 3. *If* $P = ab$, *then*

$$\mathrm{suf}_P(aca) = 1, \quad \mathrm{suf}_P(aab) = 2, \quad \textit{and} \quad \mathrm{suf}_P(acb) = 0.$$

We now want to define δ such that

$$\delta(q_0, T[1,\ldots,i]) = \mathrm{suf}_P(T[1,\ldots,i]), i = 0,\ldots,n. \tag{1}$$

From Eq. 1 it follows then especially that

$$\delta(q_0, T[1,\ldots,i]) = m \iff P \text{ occurs with shift } i-m \text{ in } T.$$

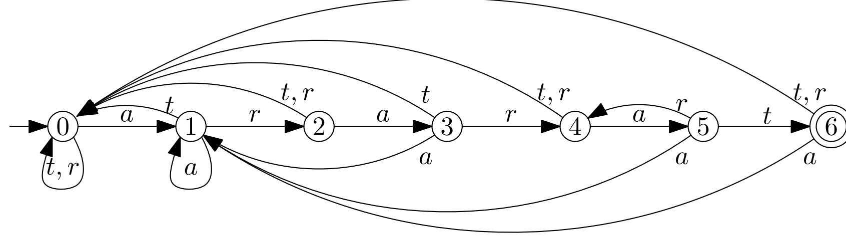
The definition of the transition function must not depend on the text but only on the pattern. Assume there is a δ fulfilling Eq. 1. Let $q = \mathrm{suf}_P(T[1,\ldots,i-1]) = \delta(q_0, T[1,\ldots,i-1])$, $a = T[i]$ and $q' = \delta(q,a)$. Then both, $P[1,\ldots\ldots,q]a$ and $P[1,\ldots,q']$ are suffixes of $T[1,\ldots,i]$:

$$\begin{array}{r} P[1,\ldots,q'] \\ T[1,\ldots\ldots\ldots\ldots,i-1,i] \\ P[1,\ldots\ldots,q]a \end{array}$$

This motivates us to define

$$\boxed{\delta(q,a) := \mathrm{suf}_P(P[1,\ldots,q]a).}$$

Example 4. $P = ararat, \Sigma = \{a, r, t\}$,



The text $T = arararat$ *traverses* $\mathcal{A}_P$ *along* $0, 1, 2, 3, 4, 5, 4, 5, 6$.

It remains to show that the definition of δ indeed yields Eq. 1.

Lemma 1. $\delta(q_0, T[1, \ldots, i]) = suf_P(T[1, \ldots, i]), i = 0, \ldots, n.$

Proof. We prove the lemma by induction on i. For $i = 0$ we have

$$\delta(q_0, \epsilon) = q_0 = 0 = \mathrm{suf}_P(\epsilon).$$

For $i > 0$, we have

$$\begin{aligned} \delta(q_0, T[1, \ldots, i]) &\overset{\text{Def. extension of } \delta}{=} \delta(\delta(q_0, T[1, \ldots, i-1]), T[i]) \\ &\overset{\text{inductive hypothesis}}{=} \delta(\underbrace{\mathrm{suf}_P(T[1, \ldots, i-1])}_{=:q}, T[i]) \\ &\overset{\text{Def. of } \delta}{=} \underbrace{\mathrm{suf}_P(P[1, \ldots, q]T[i])}_{=:q'} \end{aligned}$$

It remains to show that

$$q' = \underbrace{\mathrm{suf}_P(T[1, \ldots, i])}_{=:q''}.$$

$q' \leq q''$: $P[1, \ldots, q]$ is a suffix of $T[1, \ldots, i-1]$ and $P[1, \ldots, q']$ is a suffix of $P[1, \ldots, q]T[i]$:

$$\begin{array}{r} T[1, \ldots\ldots\ldots\ldots, i-1, i] \\ P[1, \ldots\ldots, q]T[i] \\ P[1, \ldots, q'] \end{array}$$

Hence, $P[1, \ldots, q']$ is a suffix of $T[1, \ldots, i]$. Since q'' is maximum with the property that $P[1, \ldots, q'']$ is a suffix of $T[1, \ldots, i]$ it follows that $q' \leq q''$.

$q'' \leq q'$: Assume that $q'' > q'$. Since q' is maximum with the property that $P[1,\ldots,q']$ is a suffix of $P[1,\ldots,q]T[i]$ it follows that $P[1,\ldots,q'']$ is a suffix of $T[1,\ldots,i]$ but not a suffix of $P[1,\ldots,q]T[i]$. Hence, $q'' > q+1$ and we have the following situation.

$$\begin{array}{r} T[1,\ldots\ldots\ldots\ldots,i-1,i] \\ P[1,\ldots\ldots\ldots\ldots,q''] \\ P[1,\ldots\ldots,q]T[i] \\ P[1,\ldots,q'] \end{array}$$

But then $P[1,\ldots,q''-1]$ would be a larger suffix of $T[1,\ldots,i-1]$ than $P[1,\ldots,q]$ contradicting the maximality of q. □

We summarize this section with the pseudocode of a straight-forward implementation of the FINITE-AUTOMATON-MATCHER.

Algorithm 1: NAIVE-TRANSITION-FUNCTION

Input : pattern $P[1,\ldots,m]$, alphabet Σ
Output: transition function δ of $\mathcal{A}_P$

```
for q ← 0 ... m do
    for a ∈ Σ do
        k ← min{m, q+1};
        while P[1,...,k] not suffix of P[1,...,q]a do
            k ← k − 1;
        δ(q,a) ← k;
```

Algorithm 2: FINITE-AUTOMATON-MATCHER

Input : text $T[1,\ldots,n]$, transition function δ of $\mathcal{A}_P$, length m of the pattern P
Output: all valid shifts

```
q ← 0;
for i ← 1,...,n do
    q ← δ(q, T[i]);
    if q = m then
        output i − m;
```

The run time of this implementation is in

$$\mathcal{O}(\underbrace{m^3|\Sigma|}_{\text{NAIVE-TRANSITION-FUNCTION}} + \underbrace{n}_{\text{FINITE-AUTOMATON-MATCHER}}).$$

Note that the time to compute the transition function can be improved to $\mathcal{O}(m|\Sigma|)$.

7.3 The Knuth-Morris-Pratt Matcher

In this section we discuss the linear time string matching algorithm of Knuth et al. (1977). A description of the algorithm can also be found in (Cormen et al., 2001, Section 32.4) or (Schöning, 2001, Section 10.3).

Recall that the finite automaton matcher had to compute $(m+1)|\Sigma|$ entries of the transition function δ

$$\delta(q,a) := \max\{k;\ P[1,\ldots,k] \text{ is a suffix of } P[1,\ldots,q]a\} = \mathrm{suf}_P(P[1,\ldots,q]a)$$

for $q = 0,\ldots,m, a \in \Sigma$. The Knuth-Morris-Pratt algorithm computes instead a *failure function*

$$\pi(q) := \max\{k < q;\ P[1,\ldots,k] \text{ is a suffix of } P[1,\ldots,q]\} = \mathrm{suf}_P(P[2,\ldots,q]),$$

$q = 1,\ldots,m$ that depends only on P. The failure function indicates how much the shift of P can be at least augmented if there had been a mismatch and we have the following properties.

Lemma 2. *Let $a \in \Sigma$ and $0 \le q \le m$ such that $q = m$ or $P[q+1] \ne a$. $1 \le q \le m$. Then*

(1) $\pi(q) + 1 \ge \delta(q,a)$*, and*

(2) $\delta(q,a) = \delta(\pi(q),a)$.

Proof. Let $q' := \delta(q,a)$.

(1) If $q' = 0$, then trivially $\pi(q) + 1 \ge q'$. If $q' \ne 0$, then $P[1,\ldots,q'-1]$ is a suffix of $P[1,\ldots,q]$. Since $P[1,\ldots,q']$ is a suffix of $P[1,\ldots,q]a$ and $P[q+1] \ne a$ it follows that $P[1,\ldots,q'-1]$ is even a suffix of $P[2,\ldots,q]$. Hence, $q'-1 \le \pi(q)$.

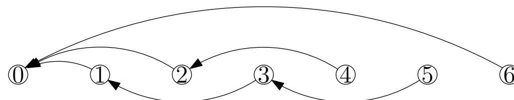
(2) By the definition of π it follows that $P[1,\ldots,\pi(q)]$ is a suffix of $P[1,\ldots,q]$. Further, $\pi(q) + 1 \ge q'$ by Item (1) of this lemma and we have the following situation

$$\begin{array}{rl} P[1,\ldots\ldots\ldots\ldots,q] & a \\ P[1,\ldots\ldots,\pi(q)] & a \\ P[1,\ldots,q'-1,q']. & \end{array}$$

It follows that $P[1,\ldots,q']$ is a suffix of $P[1,\ldots,\pi(q)]a$. Moreover, if there was a larger prefix of P that is a suffix of $P[1,\ldots,\pi(q)]a$, then this would also be a suffix of $P[1,\ldots,q]a$ – contradicting the maximality of q'. Hence, $\delta(\pi(q),a) = q'$. □

Example 5.

	a	*r*	*a*	*r*	*a*	*t*
q	*1*	*2*	*3*	*4*	*5*	*6*
$\pi(q)$	*0*	*0*	*1*	*2*	*3*	*0*



The Knuth-Morris-Pratt matcher uses the failure function to simulate the finite automaton matcher.

Algorithm 3: KNUTH-MORRIS-PRATT-MATCHER

Input : text $T[1,\dots,n]$, pattern $P[1,\dots,m]$
Output: Shifts s with which P occurs in T

```
1  π ← FAILURE-FUNCTION(P);
2  q ← 0;
3  for i ← 1,...,n do
4      while q > 0 and P[q+1] ≠ T[i] do
5          q ← π(q);
6      if P[q+1] = T[i] then
7          q ← q+1;
8      if q = m then
9          output i − m;
10         q ← π(m);
```

Line 10 is necessary, since otherwise we would ask for $P[m+1]$ in the next step. Note that Lines 4–7 replace the transition function δ, i.e. let $q_0 = 0$ and let $q_i, i = 1,\dots,n$ be the value of q after Line 7 in the ith iteration of the for-loop of Algorithm 3. Then we have the following lemma.

Lemma 3.

$$q_i = \delta(q_{i-1}, T[i]),\ i = 1,\dots,n$$

Proof. Let $i = 1,\dots,n$. Consider the number k of times that q is decreased in Line 10 in the $i-1$st iteration or in Line 5 in the ith iteration. Note that q is only decreased if $q < m$ or $P[q+1] \neq T[i]$ and it is decreased by replacing q by $\pi(q)$. Let now $q = \pi^k(q_{i-1})$ be the value after the last of the k decreases. Then we have that

$$q_i = \begin{cases} q+1 & \text{if } T[i] = P[q+1] \\ 0 & \text{if } q = 0 \text{ and } T[i] \neq P[1]. \end{cases}$$

In both cases it follows that

$$q_i = \delta(q, T[i]) = \delta(\pi^k(q_{i-1}), T[i]).$$

Lemma 2(2) applied to $a = T[i]$ and inductively to $q = \pi^j(q_{i-1}), 0 < j < k$ implies that

$$\delta(\pi^k(q_{i-1}), T[i]) = \cdots = \delta(\pi(q_{i-1}), T[i]) = \delta(q_{i-1}, T[i])$$

which concludes the proof. □

Corollary 4.

$$q_i = \delta(q_0, T[1,\ldots,i]) \overset{Lemma\ 1}{=} suf_P(T[1,\ldots,i]), i = 1,\ldots,n.$$

Example 6. *The table illustrates the behavior of the* KNUTH-MORRIS-PRATT MATCHER.

i	1	2	3	4	5	6	7	8	9	10	11	12	
T	a	r	a	r	a	a	r	a	r	a	t	a	
P	a	r	a	r	a	t							$q = 5, i = 6$
			a	r	a	r							$q = 3, i = 6$
					a	r							$q = 1, i = 6$
						a							$q = 0, i = 6$
						a	r						$q = 1, i = 7$
						a	r	a					$q = 2, i = 8$
						a	r	a	r				$q = 3, i = 9$
						a	r	a	r	a			$q = 4, i = 10$
						a	r	a	r	a	t		$q = 5, i = 11,$ *output shift* 5
												a	$q = 0, i = 12$

To compute the failure function, we apply the KNUTH-MORRIS-PRATT-MATCHER to $T = P[2,\ldots,n]$. Then we have

$$q_i = \mathrm{suf}_P(P[2,\ldots,i]) = \pi(i)$$

after Line 7. Observe that in Line 5 we only need $\pi(q), q < i$ during the step in which we compute $\pi(i)$.

Algorithm 4: FAILURE-FUNCTION

Input : pattern $P[1,\ldots,m]$
Output: failure function π

```
1 π(1) ← 0;
2 q ← 0;
3 for i ← 2,...,m do
4     while q > 0 and P[q+1] ≠ P[i] do
5         q ← π(q);
6     if P[q+1] = P[i] then
7         q ← q + 1;
8     π(i) ← q;
9 return π;
```

Theorem 5. *The Knuth-Morris-Pratt-Matcher finds all occurrences of a pattern* $P[1,\ldots,m]$ *in a text* $T[1,\ldots,n], n \geq m$ *in* $\mathcal{O}(n)$ *time.*

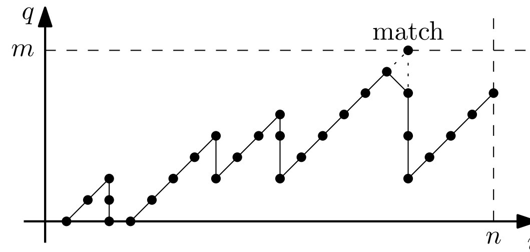


Figure 1: Typical behavior of q in Line 4 of the KNUTH-MORRIS-PRATT-MATCHER.

Proof. Consider the variation of q during a run of the KNUTH-MORRIS-PRATT-MATCHER.

- In the beginning $q = 0$.
- In Line 5 q is decreased by at least one.
- In Line 7 q is increased by one.

Hence, starting from $q = 0$, the value of q is totally increased by at most n and it is never negative. Thus, q can be decreased at most n times (see Fig. 1). Hence, the total run time of the KNUTH-MORRIS-PRATT-MATCHER after having computed the failure function is in $\mathcal{O}(n)$.

Since the computation of the failure function corresponds to applying the KNUTH-MORRIS-PRATT-MATCHER to $T = P[2, \ldots, m]$ it follows that the failure function can be computed in $\mathcal{O}(m) \subseteq \mathcal{O}(n)$ time. □

7.4 Suffix-Trees

In this section, we introduce suffix-trees and we discuss the algorithm of Ukkonen (1995) for constructing a suffix tree in linear time. A description of the algorithm can also be found in (Gusfield, 1997, Chapter 6).

The idea of a suffix tree is to represent the text in a tree such that the occurrence of several patterns can be recognized efficiently. Note that

$$\begin{gathered} P \text{ occurs in } T \\ \Longleftrightarrow \\ P \text{ is a prefix of a suffix of } T \end{gathered}$$

Hence, the idea is to represent the set of all suffixes of the text T in a tree.

(a) Trie (b) Suffix Trie for T = arara (c) impl. Suffix Trie for T = arara

Figure 2: a) Trie for the non-empty suffixes of "ararat". b) Suffix Trie for T = arara. The labels of the leaves indicate the shift with which the corresponding suffix occurs in T.

7.4.1 Trie

(originates from "information re*trie*val")

Definition 4. *A* trie *is a rooted tree such that*

- *edges are labeled with characters from a finite alphabet* Σ
- *edges to different children of a vertex have different labels.*

Let $\sigma_1, \ldots, \sigma_k$ *be the labels of the edges on the path from the root to a vertex* v *then* v represents *the string* $\overline{v} := \sigma_1 \ldots \sigma_k$.

A trie is a trie for $M \subset \Sigma^*$ *if* M *is the set of strings represented by the leaves.*

Example 7. *Fig. 2(a) indicates a trie for the set of all non-empty suffixes of the word ararat.*

Note that $M \subset \Sigma^*$ can only be represented by the leaves of a trie if no string in M is a prefix of another string in M. This problem can be solved by choosing a *sentinel character* $\$ \notin \Sigma$ and adding it at the end of each string. An *implicit trie* for M is a trie for the set of all strings in M that are not prefixes of other strings in M.

Example 8. *Fig. 2(b) indicates a trie with sentinel character \$ and Fig. 2(c) indicates an implicit trie for the set of all suffixes of the word arara.*

Definition 5. *An* implicit suffix trie *for a text* T *is an implicit trie for the set of all non-empty suffixes of* T. *A* suffix trie sTrie(T) *for a text* T *is an implicit suffix trie for* $T\$$.

Note the following relation to string matching.

$$\text{A pattern } P \text{ occurs in a text } T$$
$$\Longleftrightarrow$$
$$P \text{ is represented by a vertex } v \text{ of the (implicit) suffix trie for } T.$$

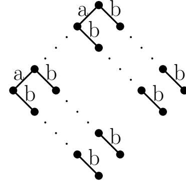
Moreover

$$\text{A pattern } P \text{ occurs with shift } s \text{ in a text } T$$
$$\Longleftrightarrow$$
$$P \text{ is represented by a vertex } v \text{ of sTrie}(T) \text{ and}$$
$$\text{there is a leaf } w \text{ in the subtree rooted at } v \text{ with } s = n - |\overline{w}| \underbrace{+1}_{\text{for \$}}.$$

Example 9. *Fig. 2(c) indicates the implicit suffix trie and Fig. 2(b) the suffix trie for $T =$ arara. The leaves of the suffix trie indicate that the pattern "ra" occurs in T with shift 1 and 3.*

Remark 6. *In the worst case the size of a suffix trie for a text $T[1,\ldots,n]$ is in $\Theta(n^2)$.*

Proof. A suffix trie for a text of length n represents $n+1$ strings of length at most $n+1$. Hence, the size of a suffix trie is in $\mathcal{O}(n^2)$. On the other hand does the implicit suffix trie for $T = a^{\frac{n}{2}}b^{\frac{n}{2}}$ have $\left(\frac{n}{2}+1\right)^2$ vertices.



□

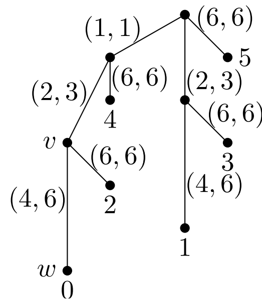
7.4.2 Suffix Tree

Definition 6. *The (implicit) suffix tree (sT(T)) sTree(T) for $T \in \Sigma^*$ is constructed from the (implicit) suffix trie by contracting all directed paths consisting of vertices of degree$= 2$ only as follows.*

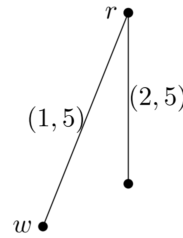
Let $v_j,\ldots,v_k$ be a maximal directed path in the (implicit) suffix trie such that $\deg(v_i) = 2, i = j+1,\ldots,k-1$ and let the label of (v_{i-1},v_i) be $T[i], i = j+1,\ldots,k$. Then the path $v_j,\ldots,v_k$ is replaced by the single edge (v_j,v_k) labeled $(j+1,k)$.

Example 10. *Fig. 3 indicates the (implicit) suffix tree for $T =$ arara.*

Note that the size of sTree($T[1,\ldots,n]$) and sT($T[1,\ldots,n]$), respectively, is in $\mathcal{O}(n)$. Further note that a substring P of T is not necessarily represented by a vertex of the (implicit) suffix tree for T, but by a vertex v and a prefix of the label of an outgoing edge of v.



(a) Suffix Tree for T = arara



(b) impl. Suffix Tree for T = arara

Figure 3: In b) the labels of the leaves indicate the shift with which the corresponding suffix occurs in T. The location of the pattern "arar" is $((v, w), (4, 4))$ in the suffix tree and $((r, w)(1, 4))$ in the implicit suffix tree.

Definition 7. *Let $P \neq \epsilon$ be a pattern that occurs in a text T. Let (v, w) be an edge of the (implicit) suffix tree and let $k \leq \ell$. We say that $((v, w), (k, \ell))$ is the* location *of the pattern P if*

- *the label of (v, w) is (k, ℓ') for an $\ell' \geq \ell$ and*
- *$P = \overline{v}T[k, \ldots, \ell]$.*

If $P = \overline{w}$, then we also say that w is the location of P.

The location of ϵ is the root r. We denote the location of ϵ also by $((r, r), (0, 0))$ and assume that the (implicit) suffix tree contains the loop (r, r) labeled $(0, 0)$.

Example 11. *The location of arar is $((v, w), (4, 4))$ in the suffix tree of arara (Fig. 3(a)) and $((r, w), (1, 4))$ in the implicit suffix tree of arara (Fig. 3(b)).*

A naive method for constructing a suffix tree adds suffixes in descending size to an initially empty tree. This algorithm has a run time that is quadratic in the length of the text – provided the alphabet has constant size.

7.4.3 Ukkonen's Online Algorithm

The algorithm of Ukkonen constructs a suffix tree in linear time – provided the alphabet has constant size. Starting from the empty string, Ukkonen's algorithm (UKK) constructs implicit suffix trees for all prefixes of T, i.e.,

$$\mathrm{sT}(T[1]), \ldots, \underbrace{\mathrm{sT}(T[1, \ldots, i])}_{\widehat{T}_i}, \ldots, \mathrm{sT}(T), \mathrm{sT}(T\$) = \mathrm{sTree}(T).$$

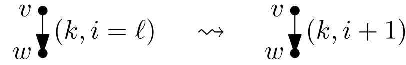
In Phase $i+1$, the implicit suffix tree $\widehat{T}_{i+1}$ is constructed from $\widehat{T}_i$ as follows.

Algorithm 5: Ukkonen's Online Algorithm (High level description)

```
1 for j ← 1,... i+1 do                                    // Step j of Phase i+1
2     Find the location of T[j,...,i];
3     Represent T[j,...,i+1] (according to the three extension rules);
```

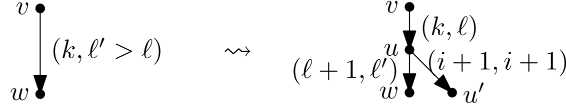
Let loc $= ((v,w),(k,\ell))$ be the location of $T[j,\dots,i]$. There are the following three *extension rules.*

(1) If loc is a leaf, i.e. if w is a leaf and $\ell = i$, then label$(v,w) \leftarrow (k, i+1)$.

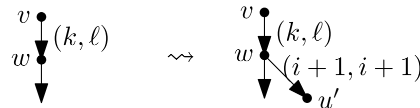


(2) If loc is not a leaf and no path starting from loc is labeled $T[i+1]$, then add a new edge starting from loc labeled $(i+1, i+1)$, i.e.

(a) if label$(v,w) = (k,\ell'), \ell' > \ell$ and $T[i+1] \neq T[\ell+1]$, then delete the edge (v,w) and add two new vertices u, u' and three new edges (v,u), (u,w), and (u,u') with labels (k,ℓ), $(\ell+1,\ell')$, and $(i+1,i+1)$, respectively.



(b) if label$(v,w) = (k,\ell)$, w is not a leaf and $T[i+1] \neq T[k']$ for each edge starting from w and labeled (k',ℓ'), then add a new leaf u' and a new edge (w,u') labeled $(i+1,i+1)$.



(3) If loc is not a leaf and some path starting from loc is labeled $T[i+1]$, then do nothing.

Example 12. *Let $T = acca$.*

Table 1 shows how to compute $\widehat{T}_4$ from $\widehat{T}_3$: r; acc (1, 3) → v; cc (2, 3) → w .

j	$T[j,\ldots,3]$	location		rule	$\widehat{T}$
1	acc	$((r,v),(1,3))$		1	r; acca $(1,4)$ → v; cc $(2,3)$ → w
2	cc	$((r,w),(2,3))$		1	r; acca $(1,4)$ → v; cca $(2,4)$ → w
3	c	$((r,w),(2,2))$	$T[4]=a \neq c = T[3]$	2	r; acca $(1,4)$ → v; $(2,2)$,c → u; $(4,4)$,a → u'; $(3,4)$,ca → w
4	ϵ	$((r,r),(0,0))$	$T[4]=a=T[1]$	3	r; acca $(1,4)$ → v; $(2,2)$,c → u; $(4,4)$,a → u'; $(3,4)$,ca → w

Table 1: Phase 4 of Computing the suffix tree for acca

7.4.3.1 Making Rules 1 and 3 Implicit

First we make the following observation.

Once a leaf always a leaf.

None of the three rules appends an edge to a leaf. So if we create a leaf in one step of the algorithm, it will remain a leaf during all later phases of the algorithm. The only cases where we create new leaves is an application of Rule 2. Further, we also know something about the labels of the edges to the leaves.

An edge to a leaf of $\widehat{T}_i$ is labeled $(.,i)$.

Indeed, the label of the only edge (r,w) of $\widehat{T}_1$ is $(1,1)$ and w represents $T[1]$. So assume that any edge to a leaf of $\widehat{T}_i$ is labeled $(.,i)$ and that a leaf w of $\widehat{T}_i$ represents the string $T[j_w,\ldots,i]$. Let now w be a leaf of $\widehat{T}_{i+1}$. Then w had already been a leaf of $\widehat{T}_i$ or w was created by an application of Rule 2 in Phase $i+1$. In the first case $(.,i)$ was replaced by $(.,i+1)$ in Step j_w of Phase $i+1$ and in the latter case the new edge was labeled $(i+1,i+1)$.

Hence, in order to make Rule 1 implicit, we label the edges to the new leaves by $(i+1,\infty)$ instead of $(i+1,i+1)$, where ∞ represents a global variable referring to the number of the current phase. To make Rules 1+3 completely implicit, we make another observation.

Lemma 7. *During a fixed phase of the algorithm of Ukkonen any application of Rule 1 occurs before any application of Rule 2 or 3 and any application of Rule 2 occurs before any application of Rule 3.*

Proof. Assume that Rule 2 or 3 is applied in Step j of Phase $i+1$, i.e., when inserting $T[j,\ldots,i+1]$. That means especially that the location of $T[j,\ldots,i]$ was not a leaf before inserting $T[j,\ldots,i+1]$. Hence, there is a $\ell < i$ such that $T[j,\ldots,i]$ is a suffix of $T[1,\ldots,\ell]$. Moreover, if Rule 3 was applied, then there is a $\ell < i$ such that $T[j,\ldots,i]$ is a suffix of $T[1,\ldots,\ell]$ and $T[\ell+1] = T[i+1]$.

$$\begin{array}{l} T[1,\ldots\ldots\ldots\ldots\ldots,\ell,\quad \ell+1,\ldots,j,\ldots,i,i+1,\ldots] \\ \qquad\quad T[j,\ldots\ldots\ldots,i]T[i+1] \\ \qquad\qquad\quad T[j',\ldots,i]T[i+1] \end{array}$$

Let now $j' > j$. Then $T[j',\ldots,i]$ is again a suffix of $T[1,\ldots,\ell]$. Hence, the location of $T[j',\ldots,i]$ is not a leaf since it is also the location of $T[j'+\ell-i,\ldots,\ell]$ which is a prefix of $T[j'+\ell-i,\ldots,i]$. Hence, we will again apply Rule 1 or 3. If we had applied Rule 3 at Step j, then we have again that $T[\ell+1] = T[i+1]$ and we will again apply Rule 3. □

Let j_i be the number of leaves of $\widehat{T_i}$. Since in Phase $i+1$ Rule 1 is applied to exactly the leaves of $\widehat{T_i}$ and each application of Rule 2 creates a new leaf, we have that the last application of Rule 1 in Phase $i+1$ is in Step j_i and the last application of Rule 2 in Phase $i+1$ (if any) is in Step j_{i+1}. So, in Phase $i+1$, we have the following situation:

$$\begin{array}{c|c|c} \ldots\ldots j_i & \ldots\ldots j_{i+1} & j_{i+1}+1\ldots\ldots \\ \text{Rule 1} & \text{Rule 2} & \text{Rule 3} \end{array}$$

Note that it is possible that $j_i = j_{i+1}$ meaning that Rule 2 is not applied in Phase i. Summarizing, we can refine Algorithm 5 as follows.

Algorithm 6: Ukkonen's Online Algorithm

Input : text $T[1,\ldots,n]$
Output: suffix tree $\widehat{T}$ of T

```
1  Extend T by T[n+1] = $;
2  Initialize T̂ with a single edge labeled (1, ∞);
3  j ← 2;
4  for i ← 1,...,n do
5  |   Rule3=false;
6  |   while j ≤ i+1 and Rule3 = false do          // Step j of Phase i+1
7  |   |   loc ← location(T[j,...,i]);
8  |   |   if there is a path starting from loc labeled T[i+1] then
9  |   |   |   Rule3 ← true;
10 |   |   else
11 |   |   |   apply Rule 2;
12 |   |   |   j ← j+1;
```

Example 13. *Fig. 4 illustrates Algorithm 6 applied to $T = pucupcupu$.*

Note that each application of Rule 2 creates a new leaf. The suffix tree of a text with n characters has exactly $n+1$ leaves and one leaf is created during the initialization. Hence Rule 2 is applied exactly n times. Each phase finds at most once an application of Rule 3. Hence, at most $2n$ locations have to be computed. In the next section, we will see how all these locations can be computed in totally linear time.

7.4.3.2 Suffix Links

Definition 8. *Let v be an internal vertex (i.e. v is not the root nor a leaf) of an implicit suffix tree $\widehat{T}$ and let $\overline{v} = x\sigma$, $x \in \Sigma, \sigma \in \Sigma^*$. The vertex $s(v)$ of $\widehat{T}$ with $\overline{s(v)} = \sigma$ is called the* suffix link *of v.*

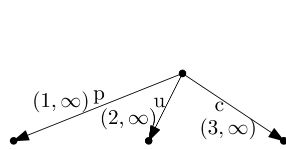
Example 14. *Fig. 5 shows the suffix tree for $T = pucupcupu$ with suffix links.*

Note that the suffix link of an internal vertex always exists: An internal vertex of v an implicit suffix tree has always more than one child. Let $\overline{v} = x\sigma$, $x \in \Sigma, \sigma \in \Sigma^*$. Then there is also more than one extension of the location of σ. Hence, the location of σ is a vertex. The following lemma indicates, how to efficiently construct the suffix links.

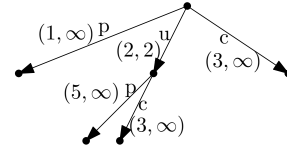
Lemma 8. *Let u be an inner vertex that was created in Step j of Phase $i+1$ and let w be the location computed in Step $j+1$ of the same phase. Then $s(u) = w$.*

	j / $i+1$	1	2	3	4	5	6	7	8	9	10
p	1	p									
u	2		u								
c	3			c							
u	4				$\boxed{\text{u}}_3$						
p	5				up	$\boxed{\text{p}}_3$					
c	6					pc	$\boxed{\text{c}}_3$				
u	7						$\boxed{\text{cu}}_3$				
p	8						$\boxed{\text{cup}}_3$				
u	9						cupu	upu	$\boxed{\text{pu}}_3$		
$	10								pu$	u$	$

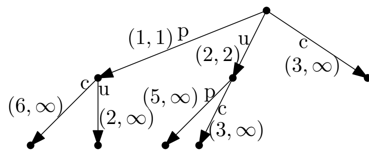
(a) Substrings to which Rule 2 or a first Rule 3 per phase ($\square_3$) is applied



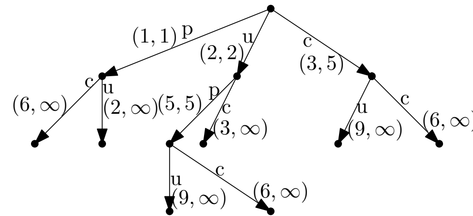
(b) after Phase 3



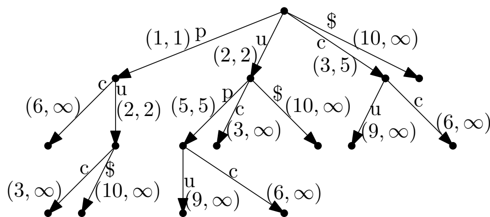
(c) after Phase 5



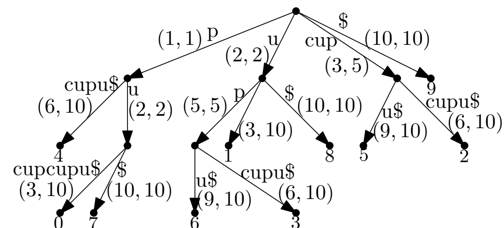
(d) after Phase 6



(e) after Phase 9



(f) after Phase 10



(g) Suffix Tree

Figure 4: Construction of the suffix tree for T = pucupcupu. In 4(g) the labels of the leaves indicate the shift of the corresponding suffix.

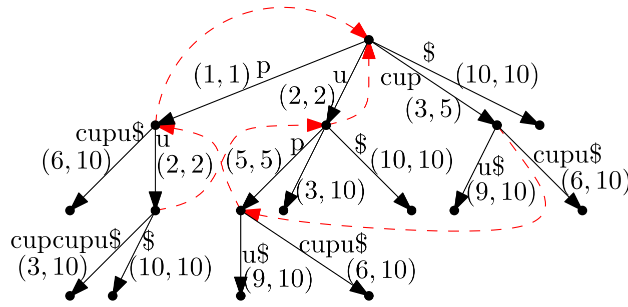
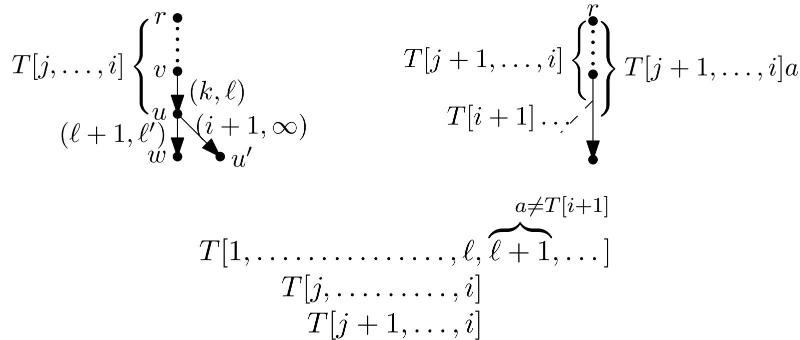


Figure 5: The suffix tree for T = pucupcupu. Dashed red edges indicate suffix links.

Proof. Observe that if an inner vertex u is created for $T[j, \ldots, i+1]$, then $\overline{u} = T[j, \ldots, i]$. The location computed in Step $j+1$ is the location of $T[j+1, \ldots, i]$ which is indeed $\overline{u}$ without the first character. It remains to show that the location of $T[j+1, \ldots, i]$ is indeed a vertex after Step $j+1$.



Note that since u is an inner vertex it follows that there is a $T[i+1] \neq a \in \Sigma$ such that $T[j, \ldots, i]a$ occurs in $T[1, \ldots, i]$. Hence, the substring $T[j+1, \ldots, i]a$ of $T[j, \ldots, i]a$ does also occur in $T[1, \ldots, i]$. Thus, there is a path starting from the location of $T[j+1, \ldots, i]$ which is not labeled $T[i+1]$. Hence, either the location of $T[j+1, \ldots, i]$ was already a vertex when it was found or a vertex at the location of $T[j+1, \ldots, i]$ was created in Step $j+1$ of Phase $i+1$ when appending the edge with label $(i+1, \infty)$.

Note that if a vertex was newly created this was not the last step of a phase: The last step of a phase is due to Rule 3 which does not create new vertices or because a new leaf is appended to the root which does not create new inner vertices. □

Let P and Q be two substrings of a text T such that the concatenation PQ does also occur in T. Let loc be the location of P in $\mathrm{sT}(T)$. Then we denote by "loc plus Q" the location of PQ in $\mathrm{sT}(T)$, i.e., the location that is reach from loc by following the path labeled Q.

Algorithm 7: LOCATION

Input : i, j with $2 \leq j \leq i+1 \leq n+1$
Output: location of $T[j,\ldots,i]$
Data : location loc $= ((v,w),(k,\ell))$ of the previous step

```
1  if first step of a phase then
2      if j = i + 1 then                              // T[j,...,i] = ϵ
3          loc ← root;
4      else                                           // loc = LOCATION(T[j,...,i−1])
5          loc ← loc plus T[i];
6  else                                               // loc = LOCATION(T[j−1,...,i])
7      if v ≠ root then
8          loc ← s(v) plus T[k,...,ℓ];
9      else                                           // T[j,...,i] = T[k+1,...,ℓ]
10         loc ← root plus T[j,...,i];
11 return loc;
```

We compute the suffix links in constant time per link according to Lemma 8 as follows. Let $((v,w),(k,\ell))$ be the location of $T[j-1,\ldots,i]$ after Step $j-1$ of Phase $i+1$. If a new inner vertex was created in Step $j-1$ of Phase $i+1$ it is w. In that case, we set $s(w)$ to be the location of $T[j,\ldots,i]$ after Step j of Phase $i+1$. Note that the suffix link of w is not needed to compute the location in the next step.

Further note that since we know that there is a path in $\widehat{T}$ starting from $s(v)$ and labeled $T[k,\ldots,\ell]$ (or starting from the root and labeled $T[k+1,\ldots,\ell]$, respectively) we can compute these paths in time linear in the number of traversed vertices of $\widehat{T}$ by just comparing the first and last character of the label of an edge instead of all characters in between.

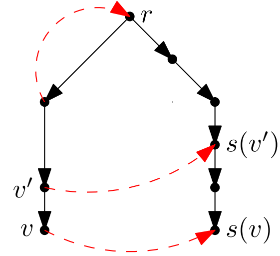
Theorem 9. *The algorithm of Ukkonen computes a suffix tree in linear time if the size of the alphabet is constant.*

Proof. It remains to show that all locations can be computed in linear time. Let r be the root of the suffix tree. As a potential function, we consider

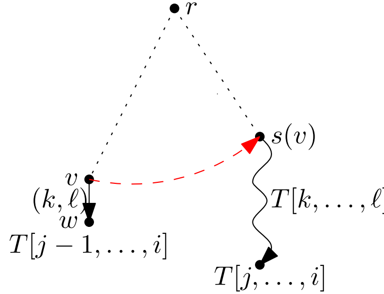
$$\text{depth}(v) := \text{ number of edges on the } rv-\text{path}$$

of the current location. First note that for any inner vertex v we have

$$\text{depth}(v) \leq \text{depth}(s(v)) + 1 :$$



Let v be an inner vertex and let $v' \neq r$ be a vertex on the rv-path. Then $s(v')$ is on the $rs(v)$-path. Moreover, two distinct vertices on a directed path have distinct suffix links. Hence, the $rs(v)$-path contains at least as many vertices as the rv-path without the root.



It follows that in each step the depth of the current location is first decreased by at most two (going possibly from the current location w to v and from v to $s(v)$) and then potentially increased by following the path labeled $T[i]$, $T[k,\dots,\ell]$, or $T[k+1,\dots,\ell]$. Since in the totally $2n$ steps the depth is totally decreased by at most $4n$ and it never exceeds n, it can be incremented at most $5n$ times. Hence, the total time needed to traverse the paths labeled $T[k,\dots,\ell]$ or $T[k+1,\dots,\ell]$, respectively, within all steps is linear in the length of the text. □

We conclude this section with a comparison of how to organize the children of a vertex in the suffix tree.

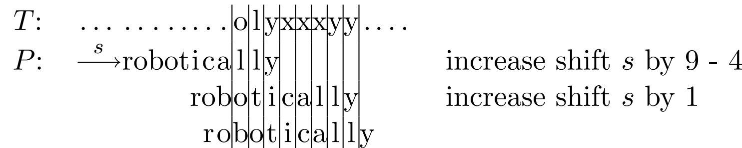
	memory space	access time
arrays	$\mathcal{O}(n\lvert\Sigma\rvert)$	$\mathcal{O}(1)$
lists	$\mathcal{O}(n)$	$\mathcal{O}(\lvert\Sigma\rvert)$
balanced search trees	$\mathcal{O}(n)$	$\mathcal{O}(\log\lvert\Sigma\rvert)$
hashing	$\mathcal{O}(n)$	$\mathcal{O}(n)$, depending on collisions, expect. $\mathcal{O}(1)$

7.5 The Algorithm of Boyer & Moore

The algorithm of Boyer and Moore (1977) has a sublinear expected run time and is very good in practice (especially for natural language texts). The idea is to start comparing the pattern with the text from the last character of the pattern. If the character of the text that is compared to the rightmost character of P does not occur in P, then $m-1$ characters of the text do not have to be examined at all.

7.5.1 Bad Character Rule

Consider the following example.



increase shift s by 9 - 4

increase shift s by 1

In the first case the rightmost mismatch is at $j = m - 2$ and the rightmost occurrence of $T[s + j] = \mathrm{o}$ in P is at position 4. Hence the shift s can be augmented by $j - 4$. In the second case the rightmost occurrence of the mismatching text character y in P is to the right of the mismatching pattern character. Hence the shift is only augmented by one. (In fact the pattern would allow a shift by $j = 10$, since y does not occur in P to the left of j but this would require a more involved and time consuming data structure, as described in this section. See also Sect. 7.5.3.1.)

Let $r_P(a)$ be the rightmost occurrence of a character a in P, i.e.,

$$\begin{array}{rcl} r_P : \Sigma & \longrightarrow & \{1, \ldots, m\} \\ a & \longmapsto & \begin{cases} 0 & \text{if } a \text{ does not occur in P} \\ \max\{j;\ P[j] = a\} & \text{otherwise} \end{cases} \end{array}$$

If the rightmost mismatch occurs at position j of P

$$\text{augment the shift } s \text{ by } \max(1, j - r_P(T[s + j]).$$

7.5.2 Good Suffix Rule

Consider the following example.

```
T:  ..........|d|b|l|e|x|x|a|b|l|e|............
P:  --s-->leblend|a|b|l|e| | | | | | |              increase shift by 11 - 5
             l|e|b|l|e|n|d|a|b|l|e|              increase shift by 11 - 2
              | | | | | | | | |l|eblendable
```

In the first row there is a mismatch at $j = m - 3 = 8$ and $P[3, \ldots, 5]$ is the rightmost substring of $P[1, \ldots, m-1]$ that matches $P[j+1, \ldots, m]$. Hence the shift can be increased by $m - 5$. In the second row there is a mismatch at $j = m - 4$, no substring of $P[1, \ldots, m - 1]$ matches $P[j + 1, \ldots, m]$ and $P[1, 2]$ is the largest prefix of P that matches a suffix of $P[j + 1, \ldots, m]$. Hence, the shift can be augmented by $m - 2$.

We define $s_P(j)$ to be the length of the largest prefix of P such that

- $P[j + 1, \ldots, m]$ is a suffix of $P[1, \ldots, s_P(j)]$ and $P[j] \neq P[s_P(j) - m + j]$

$\ldots$		$\ldots\ldots, s_P(j)$	$\ldots$	j	$j + 1, \ldots, m$
	$\not a$	////////////		a	////////////

or

- $P[1, \ldots, s_P(j)]$ is a suffix of $P[j + 1, \ldots, m]$.

$1, \ldots, s_P(j)$	$\ldots\ldots, j$	$j + 1, \ldots$	$\ldots, m$
///////			///////

If the rightmost mismatch occurs at position j of P we can

$$\text{augment the shift } s \text{ by } m - s_P(j).$$

The algorithm of Boyer and Moore combines the bad character rule and the good suffix rule and works as follows.

Algorithm 8: Algorithm of Boyer & Moore

Input : text $T[1, \ldots, n]$, pattern $P[1, \ldots, m]$ with functions r_P, s_P.
Output: Shifts s with which P occurs in T

```
1  s ← 0;
2  while s ≤ n − m do
3  |   j ← m;
4  |   while j > 0 and P[j] = T[s + j] do
5  |   |   j ← j − 1;
6  |   if j > 0 then
7  |   |   s ← s + max(m − s_P(j), j − r_P(T[s + j]));
8  |   else
9  |   |   output shift s;
10 |   |   s ← s + m − s_P(1);
```

One can show that the run time of the algorithm of Boyer & Moore is linear if the pattern does not occur in the text (Cole, 1994). The expected run time is sublinear even if only the bad character rule is applied. The algorithm of Boyer and Moore works especially good for large alphabets. Note that the worst case run time of the algorithm of Boyer & Moore in general, i.e., if P might occur in T, is in $\Theta(m(n - m + 1))$ (e.g. $T = a^n$, $P = a^m$). There are some modifications (Galil, 1979) such that the run time of the algorithm is in general linear. See also (Gusfield, 1997, Section 3.2).

7.5.3 Preprocessing

In this section, we show how the functions r_P and s_P can be computed efficiently.

7.5.3.1 Preprocessing for the Bad Character Rule

The function r_P can be computed in $\mathcal{O}(m + |\Sigma|)$ time using an array (see Algorithm 9 for details).

By storing all occurrences of a character in P from right to left in a list instead of storing only the rightmost one, the increase of the shift is potentially higher since we can always shift to the rightmost occurrence of a character to the **left** of the mismatch. In the worst case the run time of the algorithm of Boyer & Moore is at most doubled.

i	1	2	3	4	5	6	7
$P[i]$	a	b	b	a	b	a	b
$\pi(i)$	0	0	0	1	2	1	2
$s_P^{\text{pref}}(i)$	2	2	2	2	2	0	0
$\pi'(i)$	0	2	1	0	3	0	
$s_P^{\text{match}}(i)$	0	0	0	5	2	3	6

Table 2: Example of the indices used for the preprocessing of the good suffix rule.

Algorithm 9: Bad Character Preprocessing

Input : pattern P, alphabet Σ
Output: function r_P

```
1 for a ∈ Σ do
2     r_P(a) ← 0;
3 for j ← 1,...,m do
4     r_P(P[j]) ← j;
```

7.5.3.2 Preprocessing for the Good Suffix Rule

We split the computation of s_P into two parts. With $\max\emptyset := 0$ we define

$$s_P^{\text{match}}(j) = \max\{1 \le k < m;\ P[j+1,\dots,m] \text{ suffix of } P[1,\dots,k] \text{ and } (k-m+j=0 \text{ or } P[j] \ne P[k-m+j])\}$$

and

$$s_P^{\text{pref}}(j) = \underbrace{\max\{0 \le k < m;\ P[1,\dots,k] \text{ suffix of } P[j+1,\dots,m]\}}_{\text{suf}_P(P[j+1,\dots,m])}$$

for $j = 1,\dots,m$. Then

$$s_P(j) = \max(s_P^{\text{match}}(j), s_P^{\text{pref}}(j)).$$

We first show how to compute s_P^{pref}. Let π be the failure function of the Knuth-Morris-Pratt Matcher. Note that

$$s_P^{\text{pref}}(j) = \pi^i(m) \quad \text{if and only if} \quad \pi^i(m) \le m-j < \pi^{i-1}(m).$$

$1,\dots,\pi^i(m)$	$\dots\dots j$	$j+1,\dots$	$\dots,m$
///////			///////

Hence, we can compute s_P^{pref} in $\mathcal{O}(m)$ time as follows.

Algorithm 10: Good Suffix Preprocessing – The Prefix Case

Input : pattern P, alphabet Σ
Output: function s_P^{pref}
Data : failure function π

```
1 π ← FAILURE-FUNCTION(P);
2 j ← 1;
3 k ← m;
4 while k > 0 do
5 |   k ← π(k);
6 |   while k ≤ m − j do
7 |   |   s_P^pref(j) ← k;
8 |   |   j ← j + 1;
```

To construct s_P^{match}, we consider the length $\pi'(q)$ of the longest common suffix of $P[1,\ldots,q]$ and P, i.e.,

$$\begin{aligned} \pi' : \{1,\ldots,m-1\} &\longrightarrow \{0,\ldots,m-1\} \\ q &\longmapsto \max\{k;\ P[m-k+1,\ldots,m] \text{ suffix of } P[1,\ldots,q]\} \end{aligned}$$

Then, we have for $k = \pi'(q)$

$\ldots$	$q-k$	$q-k+1,\ldots,q$ (k)	$\ldots$	$m-k$	$m-k+1,\ldots,m$ (k)
	$\not a$	///////////////		a	///////////////

Hence, we have

$$s_P^{\text{match}}(j) = q \quad \text{if} \quad q \text{ maximum with } j = m - \pi'(q)$$

If there is no such q then $s_P^{\text{match}}(j) = 0$. We can compute π' in $\mathcal{O}(m)$ time as follows. Assume, we have already computed $\pi'(m-1),\ldots,\pi'(i+1)$. We maintain the first and the last index of the leftmost non-empty copy $P[\ell,\ldots,r]$ of a suffix of P that had so far been computed, i.e.,

$$\ell = \min\{q - \pi'(q) + 1;\ \pi'(q) > 0 \text{ and } i < q < m\}$$

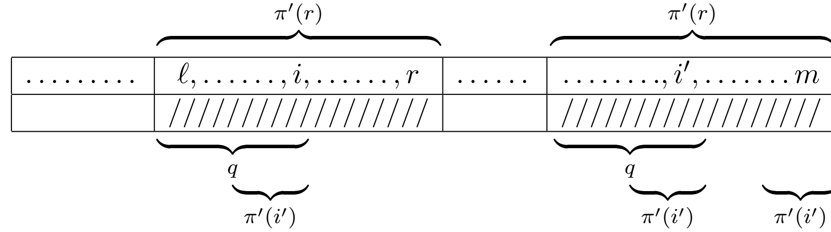
and

$$r = \min\{i < q < m;\ \pi'(q) > 0 \text{ and } \ell = q - \pi'(q) + 1\}$$

Initially, we set $\ell = m + 1$. To compute $\pi'(i)$, we distinguish three cases.

(1) If $i \geq \ell$ then $P[i]$ is contained in the non-empty suffix $P[\ell,\ldots,r]$ of P at position $i' = m - r + i$. Let $q \leftarrow i - \ell + 1$ be the length of the string $P[\ell,\ldots,i]$.

(a) If $\pi'(i') < q$ then $\pi'(i) = \pi'(i')$:



(b) If $\pi'(i') \geq q$ then $P[\ell, \ldots, i]$ is a suffix of P that can potentially be enlarged to the left of ℓ which is checked by comparing the corresponding characters from right to left starting from $P[\ell - 1]$ and $P[m - \pi'(r)]$ until a mismatch occurs.

(2) If $i < \ell$ we directly compare the corresponding characters from right to left starting from $P[i]$ and $P[m]$ until a mismatch occurs.

Algorithm 11 summarizes how to compute π' and s_P^{match} in $\mathcal{O}(m)$ time.

References

Aho, A. V., Hopcroft, J. E., and Ullman, J. D. (1974). *The Design and Analysis of Computer Algorithms.* Addison-Wesley.

Boyer, R. S. and Moore, J. S. (1977). A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772.

Cole, R. (1994). Tight bounds on the complexity of the Boyer-Moore pattern matching algorithm. *SIAM Journal on Computing*, 23(5):1075–1091.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, S. (2001). *Introduction to Algorithms.* MIT Press.

Galil, Z. (1979). On improving the worst case running time of the Boyer-Moore string searching algorithm. *Communications of the ACM*, 22(9):505–508.

Gusfield, D. (1997). *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press.

Knuth, D. E., Morris, Jr., J. H., and Pratt, V. (1977). Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323 – 350.

Schöning, U. (2001). *Algorithmik.* Spektrum Akademischer Verlag.

Ukkonen, E. (1995). On-line construction of suffix trees. *Algorithmica*, 14(3):249–260.

Algorithm 11: Good Suffix Preprocessing – The Matching Case

Input : pattern P
Output: function s_P^{match}
Data : lengths $\pi'(q)$ of the longest common suffix of P and $P[1, \ldots, q]$
after the step of the for-loop for computing $\pi'(i-1)$ we have

$$\ell = \min\{q - \pi'(q) + 1;\ \pi'(q) > 0 \text{ and } i < q < m\} \text{ and}$$
$$r = \min\{i < q < m;\ \pi'(q) > 0 \text{ and } \ell = q - \pi'(q) + 1\}$$

.

```
1  ▼ Computing π'
2      ℓ ← m;
3      for i ← m − 1, ..., 1 do
4          if i ≥ ℓ then
5              i' ← m − r + i;
6              q ← i − ℓ + 1;
7              if π'(i') < q then
8                  π'(i) ← π'(i');
9          else
10             q ← 0;
11         if i − q < ℓ then
12             while q < i and P[m − q] = P[i − q] do
13                 q ← q + 1;
14             π'(i) ← q;
15             if π'(i) > 0 then
16                 r ← i;
17                 ℓ ← r − π'(ℓ) + 1;
18 ▼ Computing s_P^match
19     for j ← 1, ..., m do
20         s_P^match(j) ← 0;
21     for q ← 1, ..., m − 1 do
22         s_P^match(m − π'(q)) ← q;
```