

String Algorithms

String-Matching:

Definition

Given a text $T \in \Sigma^*$ and a string $P \in \Sigma^+$, the string matching problem is to determine all $s \in \mathbb{N}_0$, satisfying:

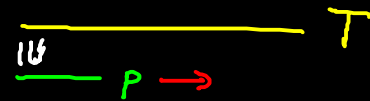
$$(\exists v \in \Sigma^s, w \in \Sigma^*)(T = vPw).$$

The number s from this definition is named *shift*.

A shift is called *feasible*, if P is found at the respective place in T , otherwise s is called *infeasible*.

Naïve algorithm: Try all shifts $s \in [0, |T| - |P|]$ one by one.

Worst case running time: $\mathcal{O}(|P| \cdot |T|)$. E.g. if $P = a^m$, $T = a^n$, $m, n \in \mathbb{N}$, $m < n$.



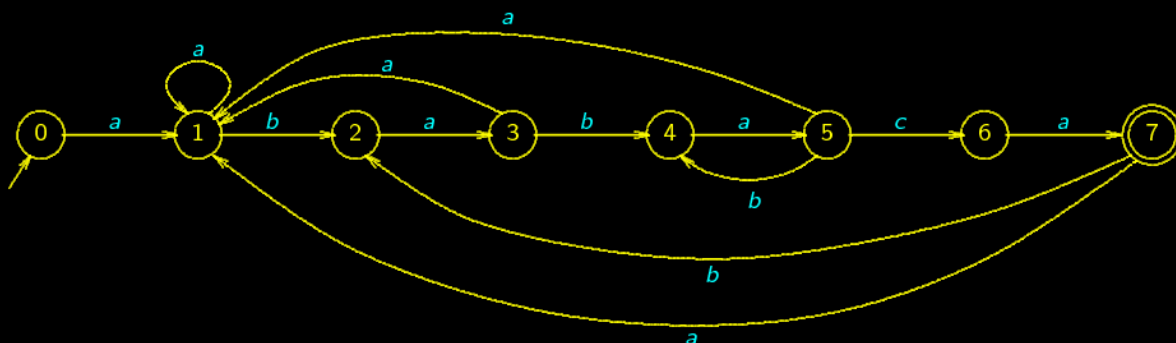
Reason of the slow running time: Knowledge about T gained in previous steps is not used. If e.g. $P = aaab$ and $s = 0$ is a feasible shift, we already know that $s = 1$, $s = 2$ and $s = 3$ are infeasible. Thus algorithm is implemented in Java runtime library!

Efficient String Matching Algorithms

Here we only give an overview:

1) Using finite automata

Example: $P = ababaca$ and $T = abababacaba$.

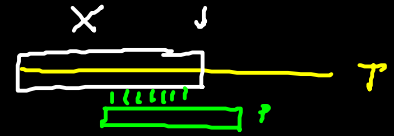


Fundamental definition:

Definition

The suffix function $\sigma_P : \Sigma^* \rightarrow \{0, 1, \dots, |P|\}$ of P is defined by

$$\sigma_P(X) := \max_{k \in \mathbb{N}_0} \{k \mid P_{0,k} \sqsupseteq X\},$$



i.e. $\sigma_P(X)$ is the length of the longest prefix of P being a suffix of X (where $P \sqsupseteq X$ denotes that P is a suffix of X and $P_{0,k}$ is the length k prefix of P).

Now with $\delta(q, a) := \sigma_P(P_{0,q} + a)$, $\forall q \in Q$ and $\forall a \in \Sigma$ a linear scan of the text is sufficient to find all feasible shifts.

Preprocessing: $\mathcal{O}(m^3 \cdot |\Sigma|)$ -algorithm to compute δ :

```
m := |P|;
for q := 0 to m do begin
  for a in Sigma do begin
    k := min(m+1, q+2); // P[0..k] should be
                        // suffix of P[0..q]+a
    repeat
      k := k-1;
    until (P[0..k] is suffix of (P[0..q]+a));
    delta[q, a] := k;
  end;
end;
```

- ▶ $P[i..j]$ denotes substring $P_{i,j}$ of P ,
- ▶ operator $+$ on strings describes concatenation.

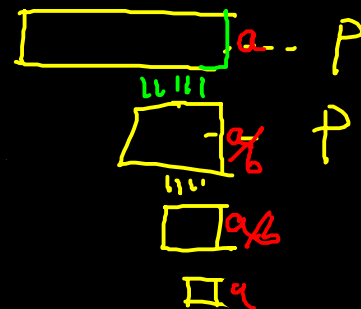
2) Knuth-Morris-Pratt Algorithm (KMP)

Definition

Let $P \in \Sigma^m$ a string. The prefix function

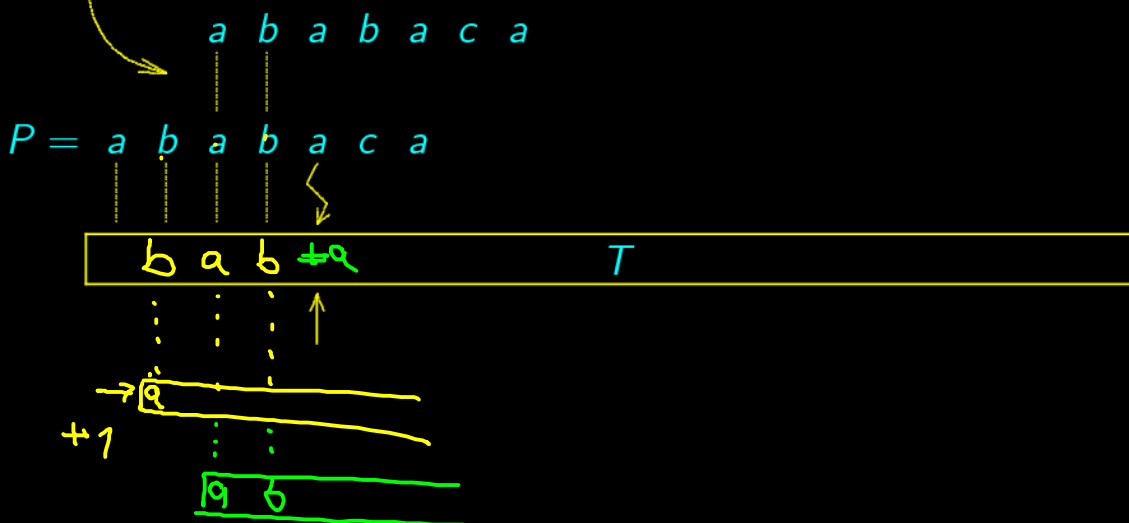
$\Pi_P : \{1, 2, \dots, m\} \rightarrow \{0, 1, \dots, m-1\}$ of P is defined by

$$\Pi_P(q) := \max_{k \in \mathbb{N}_0} \{k \mid k < q \wedge P_{0,k} \sqsupseteq P_{0,q}\}.$$



Example: For $P = ababaca$, $\Pi_P(4) = 2$ holds, since $k = 2$ is the maximum value for which $P_{0,k} \sqsupseteq P_{0,4}$, $k < 4$, holds. This leads to the following situation:

These two matches need not be verified, they can be inferred from $\Pi_P(4) = \underline{2}$.



```

1  m := |P|;
2  Pi[1] := 0;
3  k := 0;
4  for q := 2 to m do begin
5      while (k > 0) and (P[k+1] <> P[q]) do
6          k := Pi[k];
7      if P[k+1] = P[q] then
8          k := k+1;
9          Pi[q] := k;
10 end;
```

Running time: (amortized analysis using the potential method)

- ▶ Let the i -th operation Op_i be the i -th iteration of the **for**-loop. (Executing lines 7 through 9 yields constant cost c .)
- ▶ \Rightarrow Cost \mathcal{C}_i of Op_i is c plus number of iterations of the **while**-loop.
- ▶ **while**-loop iterated often only if k is large. (Assignment in line 6 strictly decreasing). **while**-loop iterated often leaves k small.

Hence we choose $\text{pot}(i) = k$.

Amortized cost

$$\mathcal{C}_i + \overbrace{\text{pot}(i) - \text{pot}(i-1)}^{\text{increase of potential during } Op_i}.$$

To reach j iterations of the **while**-loop, $k \geq j$ is required.

$\rightsquigarrow \geq j$ previous operations need to have gone without decreasing k during the **while**-loop but increasing k by 1 in line 8.

These operations have actual cost c , but are accounted with cost $c + 1$ in our analysis.

(\rightsquigarrow Overcharging of j to account for the cost of j iterations of the **while**-loop).

On the other hand $\mathcal{C}_i = c + j$ holds for the iteration, however the *increase of potential* is $-j$ (k is reduced by j , thus $\text{pot}(i) - \text{pot}(i-1) = -j$) resp. $-j + 1$, if line 8 is evaluated after the loop.

Hence amortized costs are $\leq c + j - j + 1 = c + 1$. (Here the previous overcharging and the cost of the **while**-loop are balanced, because in amortized analysis an operation including iterations of the **while**-loop is also rated with $c + 1$ at most.)

Our discussion therefor leads to

$$\mathcal{C}_i + \text{pot}(i) - \text{pot}(i-1) \leq c + 1 = \underline{\mathcal{O}(1)}.$$

Summing the amortized costs of all iterations of the **for**-loop, we get

$$\sum_{2 \leq i \leq m} (C_i + \text{pot}(i) - \text{pot}(i-1)) = \text{total cost} + \text{pot}(m) - \text{pot}(1).$$

Hence: $\text{pot}(m) - \text{pot}(1) \geq 0 \Rightarrow$ Summed amortized costs are upper bound of actual costs. This requirement is however fulfilled trivially as k never gets negative and starts with 0 in line 3.

\Rightarrow Upper bound of

$$\underline{(m-1)} \cdot \underline{\mathcal{O}(1)} = \mathcal{O}(m)$$

for the running time of our algorithm to compute the prefix function.

Knuth-Morris-Pratt (KMP) algorithm

```
1  n := |T|;
2  m := |P|;
3  // Compute prefix function Pi here
4  q := 0;
5  for i := 1 to n do begin
6      while (q > 0) and (P[q+1] <> T[i]) do
7          q := Pi[q];
8      if P[q+1] = T[i] then q := q+1;
9      if q = m then do begin
10         print('Occurrence at shift ', i-m);
11         q := Pi[q];
12     end;
13 end;
```

Remarks:

- ▶ KMP has (optimal) running time in $\mathcal{O}(m + n)$ which can be proven by a similar analysis.
- ▶ The knowledge of Π_P makes it possible to compute δ of $SMA(P)$ in linear time.
- ▶ Comparing the naïve method and the (optimized) KMP algorithm by dividing the expected number of comparisons both algorithms need on random texts we find

$$\text{KMP/NAIVE} = 1 - \frac{1}{c} + \frac{1}{c^2} + \frac{c-1}{c^m}.$$

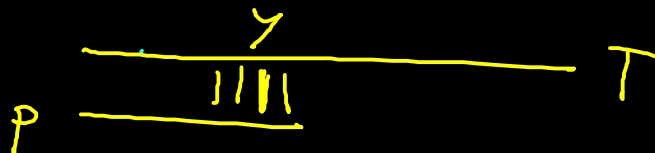
So if m and c are large enough both methods are almost equal.



3) The Boyer-Moore algorithm

Application: P long, Σ relatively large.

- ▶ **Core:** Naïve method: By setting $s := s+1$ in lines 12 and 14 we get an implementation of the naïve method.
- ▶ **Notable:** P is compared to the text from right to left.
- ▶ **Speed-up:** In case of a mismatch two heuristics (*bad character heuristic* (λ), *good-suffix heuristic* (γ)) give an increment for s which does not miss a feasible shift and is usually greater than 1.



Worst case running time of the Boyer-Moore algorithm is in

$\mathcal{O}((|T| - |P| + 1) \cdot |P| + |\Sigma|)$ (and usually in

$\Theta((|T| - |P| + 1) \cdot |P|)$), as

- ▶ the computation of lambda takes $\mathcal{O}(|P| + |\Sigma|)$ time,
- ▶ the computation of gamma takes $\Theta(|P|)$ time and
- ▶ the algorithm does not use more than $\Theta(|P|)$ time on each of the at worst $|T| - |P| + 1$ shifts.

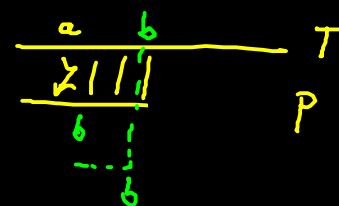
Practise: BM often the best choice as the worst case rarely occurs and the two heuristics give relatively large increments on the considered shifts. \Rightarrow **sublinear** (in in length of text) running time. BM *faster* than optimized KMP algorithm.

4) Boyer-Moore-Horspool algorithm

Variation of BM with only one heuristic similar to the bad-character heuristic. (Negative movement is avoided.)

Mismatch on comparing P with $T_{i-|P|+1,i} \Rightarrow P$ is moved to the right by $d(T_i)$ positions, where

$$d(x) := \min_{1 \leq k \leq |P|} \{k \mid k = |P| \vee P_{|P|-k} = x\}.$$



Intuition: T_i is brought to a match with a character of P (if possible). The minimizing guarantees that no potentially feasible shift is omitted.

Running time: Worst case $\Theta(|T| \cdot |P|)$, average case (sub)linear.

The constant of the linear term in the average running time is asymptotical ($|T| \rightarrow \infty$)

$$\underline{\underline{\frac{1}{|\Sigma|} + \mathcal{O}\left(\frac{1}{|\Sigma|^2}\right)}}.$$

5) Karp-Rabin algorithm

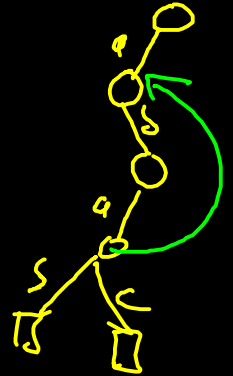
6) Algorithm of Aho and Corasick

This algorithm finds all occurrences of a set of search terms in a text (*set matching problem*) at the same time. This is achieved by organising the strings in a search term tree, a directed tree satisfying the following conditions:

- ▶ Each edge is labeled with a symbol from Σ .
- ▶ Edges leaving the same node are labeled with different symbols.
- ▶ For each search term w there is exactly one node such that the path from the root to this node is labeled with w .
- ▶ Each leaf is associated with a search term.

$P_1 = \underline{ada}g$

$P_2 = \underline{ada}c$



5) Karp-Rabin algorithm

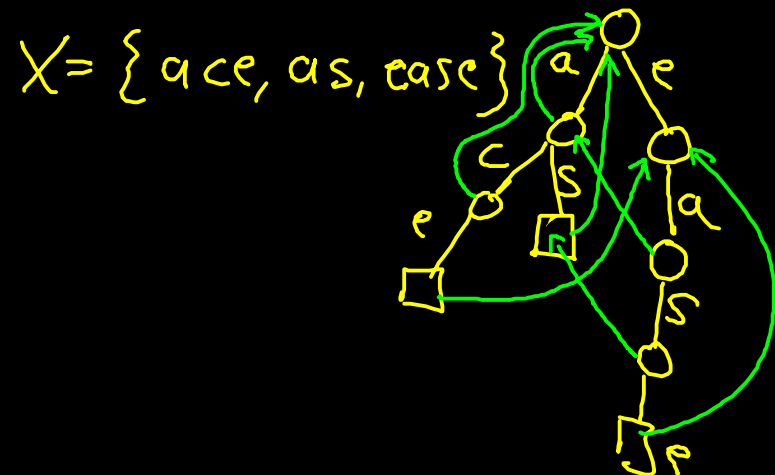
6) Algorithm of Aho and Corasick

This algorithm finds all occurrences of a set of search terms in a text (*set matching problem*) at the same time. This is achieved by organising the strings in a *search term tree*, a directed tree satisfying the following conditions:

- ▶ Each edge is labeled with a symbol from Σ .
- ▶ Edges leaving the same node are labeled with different symbols.
- ▶ For each search term w there is exactly one node such that the path from the root to this node is labeled with w .
- ▶ Each leaf is associated with a search term.

Searching in the text:

- ▶ Traverse the search term tree according to the letters of T .
- ▶ Reaching a node corresponding to a search term means we have found this term.
- ▶ If no outgoing vertex for the next symbol exists:
 - ⇒ *failure links*: Link from node v to node w such that a path from the root to w is equal to the longest suffix of the path from the root to v .



- ▶ Determining these links: Refer to $SMA(P)$, the search term tree is like a string matching automaton for a set of strings.
- ▶ **Difference:** failure links are not associated with symbols from the alphabet.
- ▶ Traversing a failure link does not consume a symbol of the text, but increase the current shift by the number of levels we went up in the tree.
- ▶ It is possible that multiple failure links are traversed in direct succession.
- ▶ If the current node is the root and there is no matching edge we stay at the root and advance to the next symbol of the text.

Suffix Trees

Idea: P appears in T , if and only if P is a prefix of a suffix of T .

Definition

Let $T \in \Sigma^n$ a text. A directed tree $B_T = (V, E)$ with root r is called a simple suffix tree for T , if it satisfies the following conditions:

1. B_T has exactly n leaves labeled with numbers 1 to n .
2. Every edge in B_T is labeled with a symbol from Σ .
3. All edges leaving an (internal) node are labeled differently.
4. The path from r to leaf i is labeled with $T_{i,n}$.

Method: Construction of a simple suffix tree B_T .

Input: Text $T \in \Sigma^n$.

Step 1: Let $T' = T \cdot \$$, $\$ \notin \Sigma$; let $\Sigma' = \Sigma \cup \{\$\}$.

Step 2: Initialize B_T with childless root r .

Step 3: For i from 1 to n repeat:

- ▶ Traverse B_T starting at r along the path $T_{i,n} \cdot \$$ until node x , reached by symbol T_k , has no leaving edge matching T_{k+1} .
- ▶ Append to x a linear list of nodes, the corresponding edges labeled with $T_{k+1,n} \cdot \$$.
- ▶ Label the new leaf with i .

String-Matching: Deciding with running time $\Theta(|P|)$.

Finding all matches: Additional effort proportional to the size of the subtree reached by P .

Problem: A simple suffix tree may have size in $\Omega(|T|^2 \cdot |\Sigma|)$.

Reason: Nodes with only one successor.

⇒ Allow each (nonempty) word as label and eliminate unary nodes. Words are represented by start- and end-position in the text.

$$|T| = n$$

$$T_{i,n}, \quad i=1, \dots, n \quad \rightsquigarrow \quad \sum_{i=1}^n i = \frac{1}{2} n(n+1)$$

Definition

Let $T \in \Sigma^n$ a text. A directed tree $B_T = (V, E)$ with root r is called compact suffix tree for T , if it satisfies the following conditions:

1. B_T has exactly n leaves, labeled with numbers 1 to n .
2. Each internal node of B_T has at least two successors.
3. The edges of B_T are labeled with substrings of T .
4. Labels of edges leaving the same node start with pairwise different symbols.
5. The path from the root to leaf i is labeled with $T_{i,n}$,
 $1 \leq i \leq n$.

Lemma

Let $T \in \Sigma^n$ a text. A compact suffix tree B_T for T has $\mathcal{O}(n)$ nodes. Labeling all edges takes $\mathcal{O}(n \log(n))$ bits.