

Advanced Algorithms

General references:

- Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman,
The Design and Analysis of Computer Algorithms
Addison - Wesley, 1974.
- Thomas H. Cormen, Charles E. Leiserson,
Ronald L. Rivest, Clifford Stein, Introduction
to Algorithms, 2nd edition, MIT Press, 2001.
- Norbert Blum, Algorithmen und Daten-
Strukturen, 2. Auflage, Oldenbourg, 2013.

Goals:

- to learn further fundamental methods
of the development of algorithms
- the development of efficient solutions
for some important problems.

Contents

- 1 Algorithms on graphs
 - 1.1 Bipartite matching
 - 1.1.1 Definitions and the general method
 - 1.1.2 The unweighted case
 - 1.1.3 The weighted case

1.2 Strong connectivity

2. Algorithms on strings

2.1 The algorithm of Knuth, Morris and Pratt.

2.2 Suffix trees

2.2.1 The data structure

2.2.2 The construction algorithm

2.2.3 Applications of suffix trees.

3. Linear programming

3.1 Foundations

3.2 The simplex method

3.3 Duality

1.1 Bipartite Matching

References

C.H. Papadimitriou, K. Steiglitz, Combinatorial Optimization: Algorithms and Complexity Prentice - Hall 1982.

N. Blum, Algorithmen und Datenstrukturen, Oldenbourg 2004 (in German).

1.1.1 Definitions and the general method

A graph $G = (V, E)$ consists of a finite, non-empty set of nodes V and a set of edges E . G is either directed or undirected. In the (un-) directed case, each edge is an (un-) ordered pair of distinct nodes. A graph $G = (V, E)$ is bipartite if V can be partitioned into disjoint nonempty sets A and B such that for all $(u, v) \in E$, $u \in A$ and $v \in B$, or vice versa. Then we often write $G = (A, B, E)$.

A path P from $v \in V$ to $w \in V$ is a sequence of nodes

$$P = v = v_0, v_1, \dots, v_k = w$$

which satisfies $(v_i, v_{i+1}) \in E$, for $0 \leq i < k$. The length $|P|$ of P is the number k of edges on P .

P is simple if $v_i \neq v_j$, for $0 \leq i < j \leq k$.

If there exists a path from v to w (of length 1) v is called a (direct) predecessor of w , and w is called a (direct) successor of v .

Let $G = (V, E)$ be an undirected graph. $M \subseteq E$ is a matching of G if no two edges in M have a common node. A matching M is maximal if there exists no $e \in E \setminus M$ such that $M \cup \{e\}$ is a matching. A matching M is maximum if there exists no matching $M' \subseteq E$ of larger size.

Given an undirected graph $G = (V, E)$, the maximum matching problem is finding a maximum matching $M \subseteq E$.

A path $P = v_0, v_1, \dots, v_k$ is M -alternating if it contains alternately edges in M and in $E \setminus M$. A node $v \in V$ is M -free if v is not incident to an edge in M .

Let $P = v_0, v_1, \dots, v_k$ be a simple M -alternating path. P is M -augmenting if v_0 and v_k are M -free. Let P be an M -augmenting path in G . Then $M \oplus P$ denotes the symmetric difference of M and P ; i.e.

$$M \oplus P := M \setminus P \cup P \setminus M.$$

Note that $M \oplus P$ is a matching of G , and $|M \oplus P| = |M| + 1$.

Theorem 1.1 (Berge 1957)

Let $G = (V, E)$ be an undirected graph and $M \subseteq E$ be a matching of G . Then M is maximum if and only if there exists no M -augmenting path in G .

Proof:

" $\Rightarrow$ "

If M is maximum then no M -augmenting path P can exist. Otherwise, $M \oplus P$ would be a matching of size $|M| + 1$, a contradiction.

" $\Leftarrow$ "

Let M be a matching of G . Assume that in G no M -augmenting path exists.

Let M' be any maximum matching of G . We have to prove $|M| = |M'|$.

Assume

$$|M| < |M'|.$$

Let us consider the subgraph

$$G' := (V, M \oplus M').$$

M and M' matchings $\Rightarrow$

$$\deg_{G'}(v) \leq 2 \quad \forall v \in V.$$

Furthermore

$$\deg_{G'}(v) = 2 \Rightarrow$$

One of the two edges with end node v is in M and the other such an edge is in M' .

Hence, the connected components of G' are of the following kinds.

- i) isolated nodes,
- ii) cycles of even length with alternately edges in M and in M' , or
- iii) paths with alternately edges in M and in M' .

$$|M'| > |M| \Rightarrow$$

At least one of the connected components is a path P with more edges in M' than in M .

$\Rightarrow$

The first edge of P and also the last edge of P are in M' .

$\Rightarrow$

Both end nodes of P are M -free.

$\Rightarrow$

⑥

P is an M -augmenting path.

This is a contradiction to the assumption that no M -augmenting path in G exists.

Berge's theorem implies the following general method for finding a maximum matching in a graph G .

Algorithm MAXIMUM MATCHING

Input: An undirected graph $G = (V, E)$, and a matching $M \subseteq E$ (possibly $M = \emptyset$)

Output: A maximum matching $M_{\max}$

Method:

while there exists an M -augmenting path

do

construct such a path P ;

$M := M \oplus P$

od;

$M_{\max} := M$.

The key problem is now this:

How to find an M -augmenting path P , if such a path exists?

Next we will define the maximum weighted matching problem.

Let $G = (V, E)$ be an undirected graph. If we associate with each edge $(i, j) \in E$ a weight $w_{ij} > 0$ then we obtain a weighted undirected graph $G = (V, E, w)$.

The weight $w(M)$ of a matching M is the sum of the weights of the edges in M . A matching $M \subseteq E$ has maximum weight if

$$\sum_{(i,j) \in M} w_{ij} \leq \sum_{(i,j) \in M'} w_{ij} \quad \forall \text{ matchings } M' \subseteq E.$$

Given a weighted undirected graph $G = (V, E, w)$, the maximum weighted matching problem is finding a matching $M \subseteq E$ of maximum weight.

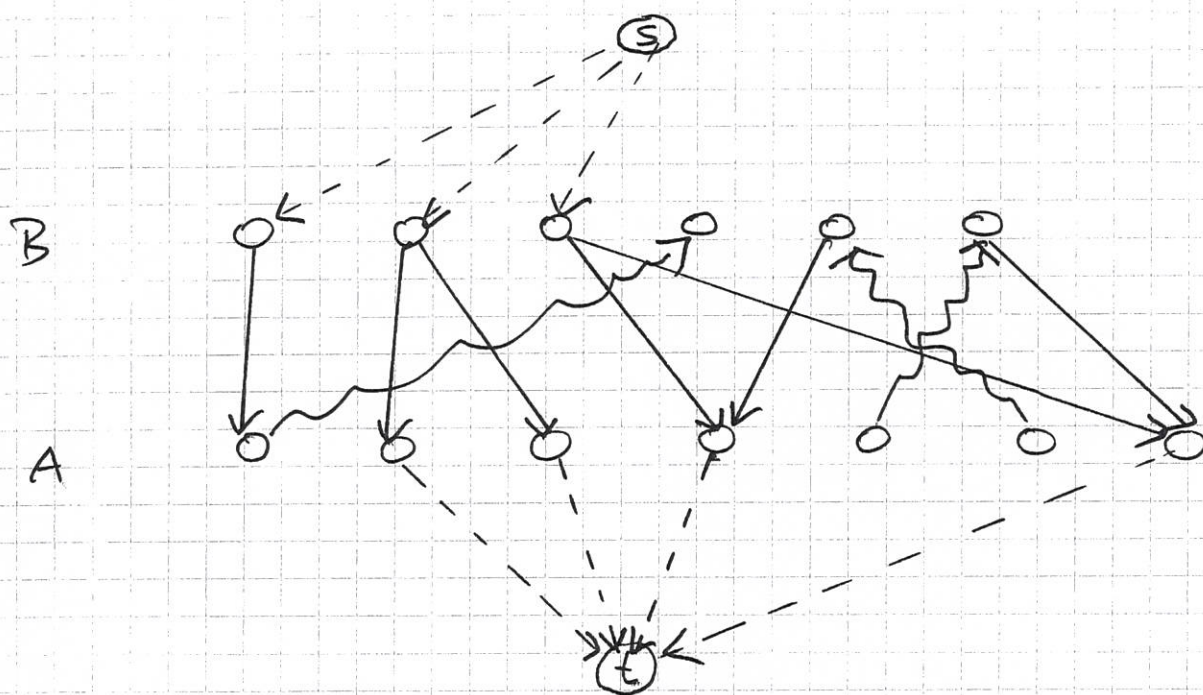
1.1.2 The unweighted case

We solve the key problem above in the following way.

- 1) We reduce the key problem to a reachability problem in a directed, bipartite graph $G_M = (A', B', E_M)$.
- 2) We solve this reachability problem constructively.

Given the bipartite graph $G = (A, B, E)$ and the matching $M \subseteq E$, we direct the edges in M from A to B and the edges in $E \setminus M$ from B to A . Additionally, we add two new nodes s and t to $A \cup B$, add for each M -free node $b \in B$ the edge (s, b) to E_M , and add for each M -free node $a \in A$ the edge (a, t) to E_M .

~>



~> edge in M

—> edge in $E \setminus M$

----> additional edge

$G_M = (A', B', E_M)$ where

$A' = A \cup \{s\}$, $B' = B \cup \{t\}$, $s, t \notin A \cup B$
 $s \neq t$

and

$$\begin{aligned}
 E_M = & \{ (u,v) \mid (u,v) \in E \text{ and } u \in A, v \in B \} \\
 & \cup \{ (x,y) \mid (x,y) \in E \setminus M \text{ and } x \in B, y \in A \} \\
 & \cup \{ (s,b) \mid b \in B \text{ } M\text{-free} \} \\
 & \cup \{ (a,t) \mid a \in A \text{ } M\text{-free} \}
 \end{aligned}$$

Lemma 1.1

Let $G = (A, B, E)$ be a bipartite graph, $M \subseteq E$ be a matching and $G_M = (A', B', E_M)$ constructed as above. Then there exists an M -augmenting path in G if and only if there is a simple path from s to t in G_M .

Proof:

" $\Rightarrow$ "

Let $P = v_0, v_1, v_2, \dots, v_k$ be an M -augmenting path in G .

W.l.o.g. we can assume $v_0 \in B$.

$\Rightarrow$

- 1, $v_i \in A$ for i odd,
- 2, $v_i \in B$ for i even,
- 3, $v_0 \in B$ M -free and $v_k \in A$ M -free,
- 4, $(v_i, v_{i+1}) \in M$ if i odd, and
- 5, $(v_i, v_{i+1}) \in E \setminus M$ if i even or zero

$\Rightarrow$

By the construction of G_M , the path

$$P' := s, v_0, v_1, v_2, \dots, v_k, t$$

is a simple path from s to t in G_M .

" $\Leftarrow$ " analogously

The following algorithm constructs an M -augmenting path if such a path exists.

Algorithm FINDAUGPATH

Input: A bipartite graph $G = (A, B, E)$ and a matching $M \subseteq E$.

Output: An M -augmenting path P if such a path exists

Method:

- (1) Construct $G_M = (A', B', E_M)$.
- (2) Apply a depth first search with start node s to G_M .
- (3) If depth first search reaches the node t then put out the path P contained in the stack without the nodes s and t .

Next we shall analyze the time used by the algorithm FINDAUGPATH.

Let $n = |A| + |B|$ and $m = |E|$.

Obviously, G_M can be constructed in $O(n+m)$ time. Depth first search uses also only $O(n+m)$ time. Hence, the algorithm `FINDAUGPATH` constructs an M -augmenting path in $O(n+m)$ time if such a path exists.

The symmetric difference $M := M \oplus P$ can be performed in $O(|P|) = O(n)$ time. At most $\lfloor \frac{n}{2} \rfloor$ augmentations can be performed. Altogether, we have proved the following theorem.

Theorem 1.2.

Let $G = (A, B, E)$ be a bipartite graph which contains no isolated nodes. Then we can compute a maximum matching of G in $O(n \cdot m)$ time where $n = |A| + |B|$ and $m = |E|$.

In general graphs $G = (V, E)$, the solution of the key problem above is much more complicated.

1.1.3 The weighted case

Let $G = (V, E, w)$ be a weighted undirected graph and $M \subseteq E$ be a matching of G . Let P be a simple M -alternating path, or a simple M -alternating cycle of even length

The gain $\Delta(P)$ with respect to P denotes the weight change of the matching after the performance of the symmetric difference $M' := M \oplus P$; i.e.,

$$\Delta(P) = \sum_{(i,j) \in P \cap E \setminus M} w_{ij} - \sum_{(i,j) \in P \cap M} w_{ij}.$$

The following theorem gives an exact characterization of a maximum weighted matching of a weighted undirected graph.

Theorem 1.3

Let $G = (V, E, w)$ be a weighted undirected graph and $M \subseteq E$ be a matching of G . Then M is of maximum weight if and only if none of the following two cases is fulfilled:

- 1) There is an M -alternating ^{simple} cycle P of even length with $\Delta(P) > 0$.
- 2) There is a simple M -alternating path P such that
 - i) $(v,w) \in M \Rightarrow (v,w) \in P$ or $v,w \notin P$ and
 - ii) $\Delta(P) > 0$.

Proof:

" $\Rightarrow$ "

Let $M \subseteq E$ be a matching of maximum weight. Obviously, none of the both cases can be

fulfilled. Otherwise, after $M := M \oplus P$ we would obtain a matching of larger weight.

⇐ "

Let $M' \subseteq E$ be any matching of G . Consider the graph

$$G' := (V, M \oplus M', w).$$

Each connected component of G' which is not an isolated node is either an simple M -alternating cycle P of even length or a simple M -alternating path P . Hence, for each connected component P we have

$$\Delta(P) \leq 0.$$

Claim: $w(M') = w(M \oplus M' \oplus M) \leq w(M)$

Proof of claim:

To prove that

$$M' = M \oplus M' \oplus M$$

consider

$$M \oplus M' = \underbrace{(M \cup M') \setminus (M \cap M')}_{\bar{M}}$$

and

$$\bar{M} \oplus M = (\bar{M} \cup M) \setminus (\bar{M} \cap M)$$

Since



$$\bar{M} \cup M = (M \cup M') \setminus (M \cap M') \cup M = M' \cup M$$

and

$$\bar{M} \cap M = (M \cup M') \setminus (M \cap M') \cap M = M \setminus (M \cap M')$$

we obtain

$$\bar{M} \oplus M = (M' \cup M) \setminus (M \setminus (M \cap M')) = M'$$

To prove that

$$w(M \oplus M' \oplus M) \leq w(M)$$

consider the connected components of G' which are not an isolated node.

Note that

$$w(M) = w(M \setminus (M \cap M')) + w(M \cap M')$$

and

$$w(M') = w(M' \setminus (M \cap M')) + w(M \cap M')$$

Since for each connected component P of G' which is not an isolated node $\Delta(P) \leq 0$ there holds

$$w(M' \setminus (M \cap M')) \leq w(M \setminus (M \cap M'))$$

and hence

$$\begin{aligned} w(M') &= w(M' \setminus (M \cap M')) + w(M \cap M') \\ &\leq w(M \setminus (M \cap M')) + w(M \cap M') \\ &= w(M). \end{aligned}$$

Applying Theorem 1.3, we have to consider M -augmenting paths, also the other simple M -alternating paths, and also the simple M -alternating cycles.

The following theorem gives us the possibility to restrict us to the consideration of M -augmenting paths.

Theorem 1.4

Let $G = (V, E, w)$ be a weighted undirected graph and $M \subseteq E$ be a matching of G such that $|M| = k$ and $w(M)$ is maximum with respect to matchings of size k . Let P be an M -augmenting path with maximal gain $\Delta(P)$. Then $M' := M \oplus P$ is a matching with maximum weight within the matchings of size $k+1$ of G .

Proof:

Let M'' be any matching of G with $|M''| = k+1$. It suffices to prove

$$w(M'') \leq w(M \oplus P).$$

Consider the subgraph

$$G' := (V, M'' \oplus M, w).$$

Let R be any M -augmenting path in G' . $|M''| > |M|$ implies that R exists.

$\Rightarrow$

$$w(M) + \Delta(R) = w(M \oplus R) \leq w(M \oplus P)$$

$$w(M'') - \Delta(R) = w(M'' \oplus R) \leq w(M)$$

After the addition of both inequalities, we obtain

$$w(M) + w(M'') \leq w(M \oplus P) + w(M)$$

$$\Leftrightarrow w(M'') \leq w(M \oplus P).$$



Applying Theorems 1.3 and 1.4 we obtain the following general method for

the computation of a maximum weighted matching of a graph $G = (V, E, w)$.

Algorithm MAXWEIMATCHING

Input: A weighted undirected graph $G = (V, E, w)$

Output: A maximum weighted matching $M_{\max}$ of G .

Method:

$M := \emptyset$;

while there exists an M -augmenting path P
with $\Delta(P) > 0$

do

construct such a path P with maximal
gain $\Delta(P)$;

$M := M \oplus P$

od;

$M_{\max} := M$.

The following theorem proves the correctness of the algorithm above.

Theorem 1.5

The algorithm MAXWEIMATCHING computes a matching of maximum weight of G .

Proof:

Let $|M_{\max}| = t$. Then by Theorem 1.4

- For $1 \leq k \leq t$, the matching M_k computed during the k th performance of the body of the while-loop has maximum weight within all matchings of size k of G .

Assume that $M_{\max}$ has not maximum weight.

Theorem 1.3 $\Rightarrow$

One of the following cases is fulfilled:

- 1) $\exists M_{\max}$ -alternating cycle P of even length with $\Delta(P) > 0$.
- 2) $\exists$ simple $M_{\max}$ -alternating path P such that
 - i) $(v, w) \in M_{\max} \Rightarrow (v, w) \in P$ or $v, w \notin P$ and
 - ii) $\Delta(P) > 0$.

If there is an $M_{\max}$ -alternating cycle P of even length with $\Delta(P) > 0$ then $M_{\max} \oplus P$ would be a matching of size t with

$$w(M_{\max} \oplus P) > w(M_{\max}).$$

This contradicts that $M_{\max}$ is a matching of size t of maximum weight within all matchings of size t .

If there is an $M_{\max}$ -alternating path P such that

- i) $(v, w) \in M_{\max} \Rightarrow (v, w) \in P$ or $v, w \notin P$ and
 ii) $\Delta(P) > 0$

then P cannot be $M_{\max}$ -augmenting. Otherwise, the algorithm would not terminate with the matching $M_{\max}$.

If the length of P is even then $M_{\max} \oplus P$ would be a matching of size t with

$$w(M_{\max} \oplus P) > w(M_{\max}),$$

a contradiction.

If the length of P is odd then $M_{\max} \oplus P$ would be a matching of size $t-1$ with

$$w(M_{\max} \oplus P) > w(M_{t-1}),$$

a contradiction.

Altogether, Theorem 1.3 implies that the computed matching $M_{\max}$ is of maximum weight.

The key problem is now this:

How to find an M -augmenting path P with maximal gain $\Delta(P) > 0$ if such a path exists?

Note that the considerations above holds also for nonbipartite graphs.

Goal:

Development of a solution of the key problem for bipartite graphs.

Let $G = (A, B, E, w)$ be a weighted undirected bipartite graph and $M \subseteq E$ be a matching of G .

Analogously to the unweighted case, we construct the weighted directed graph

$$G_M = (A \cup \{s\}, B \cup \{t\}, E_M, w)$$

where the new edges (s, b) and (a, t) obtain the weight 0.

Observation:

A depth first search on G_M with start node s finds an M -augmenting path P if such a path exists. But $\Delta(P)$ must not be maximal within all M -augmenting paths.

$\Rightarrow$

We need a mechanism which guarantees that $\Delta(P)$ is maximal within all M -augmenting paths.

$\leadsto$

The primal-dual method for the weighted bipartite matching problem.

The primal-dual method can be separated into rounds. Each round divides into two steps, the search step and the extension step.

The input of a search step will always be a subgraph G_M^* of G_M and an upper bound U such that

- 1) $\Delta(P) \leq U$ for all M -augmenting paths P in G_M
- 2) $\Delta(P) = U$ for all M -augmenting paths P in G_M^* ,
and
- 3) G_M^* contains all M -augmenting paths P of G_M with $\Delta(P) = U$.

For the construction of an M -augmenting path in G_M^* if such a path exists the search step will use depth first search. If G_M^* does not contain any M -augmenting path, the extension step will compute the input for the next search step.

Given G_M^* , the search step will be performed in the same way as in the unweighted case. It remains the description of the extension step.

Given $G = (A, B, E, w)$ we construct the graph

$$G_\phi = (A \cup \{s\}, B \cup \{t\}, E_\phi, w).$$

Let

$$W := \max \{w_{ij} \mid (i, j) \in E\}.$$

We initialize the upper bound U and the input graph $G_\emptyset^*$ for the first search step as follows:

$$U := W \quad \text{and}$$

$$G_\emptyset^* := (A \cup \{s\}, B \cup \{t\}, E_\emptyset^*, w)$$

where

$$E_\emptyset^* = \{ (i,j) \mid (i,j) \in E_\emptyset \text{ and } w_{ij} = W \} \\ \cup \{ (s,i) \mid i \in B \} \cup \{ (j,t) \mid j \in A \}.$$

It is easy to prove that U and $G_\emptyset^*$ fulfill the Properties 1-3.

Exercise:

Prove that U and $G_\emptyset^*$ fulfill the properties 1-3.

Assume that the current search step terminates with

- current upper bound U ,
- a matching M ,
- a weighted directed graph

$$G_M = (A \cup \{s\}, B \cup \{t\}, E_M, w)$$

and

- the input graph

$$G_M^* = (A \cup \{s\}, B \cup \{t\}, E_M^*, w)$$

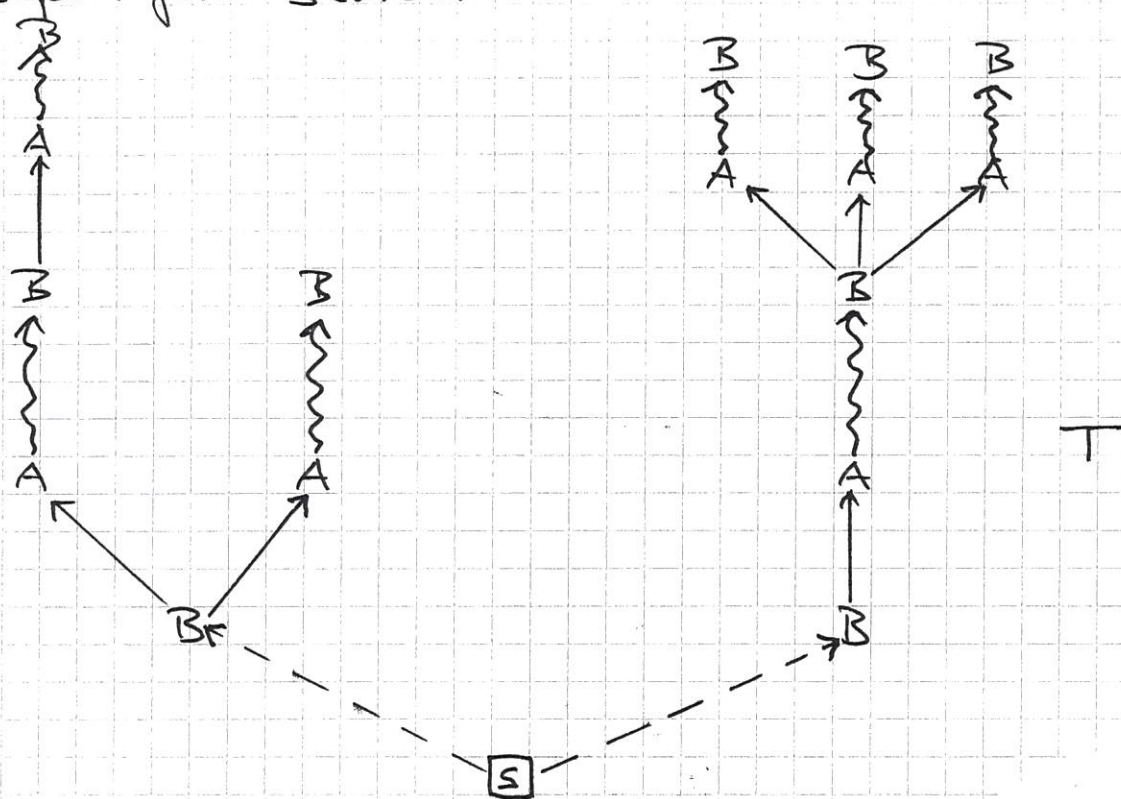
of the last unsuccessful depth first search.

Note that every M -augmenting path P in G_M has gain $\Delta(P) < U$.

Question:

How to get the reduced upper bound and the input graph for the next search step?

For getting an answer of this question let us consider the depth first search tree T which has been constructed during the last unsuccessful depth first search.



Let

$$B_T := B \cap T \quad \text{and} \quad A_T := A \cap T.$$

Goal:

The reduction of the current upper bound U by the appropriate value δ and the extension of E_M^* by edges from $E_M \setminus E_M^*$ such that the

Properties 1-3 will be fulfilled with respect to the new upper bound $U-\delta$ and the constructed input graph for the next search step.

Question:

How to get the appropriate δ and in dependence of δ those edges which we have to add to E_M^* ?

Idea

We associate with all nodes of the graph node weights such that the gain of a path P can be computed only by the consideration of the end nodes of the path P .

More exactly, we associate with each node $j \in A$ a node weight v_j and with each node $i \in B$ a node weight u_i such that always the following invariants are fulfilled:

- 1) $u_i + v_j \geq w_{ij}$ for all $(i, j) \in E \setminus M$,
- 2) $v_j + u_i = w_{ji}$ for all $(j, i) \in M$,
- 3) $u_i = U$ for all M -free $i \in B$, and
- 4) $v_j = 0$ for all M -free $j \in A$.

Then we define

$$E_M^* := M \cup \{(i, j) \in E \setminus M \mid u_i + v_j = w_{ij}\}.$$

For the computation of the appropriate value δ we define for $i \in B$ and $j \in A$ unbalances

π_{ij} and π_{ji} in the following way.

$$\pi_{ij} := u_i + v_j - w_{ij} \quad \text{and}$$

$$\pi_{ji} := v_j + u_i - w_{ji}.$$

Goal:

Prolongation of paths in the depth first search tree T by adding appropriate edges (i, j) with $i \in B_T$ and $j \in A \setminus A_T$. The upper bound u should be reduced as few as possible.

Note that $(i, j) \notin E_M^*$.

$\Rightarrow$

$$u_i + v_j > w_{ij} \quad \Rightarrow \quad \pi_{ij} > 0.$$

Question:

How to change the node weights u_i and v_j such that the edge (i, j) is added to E_M^* ; i.e., $\pi_{ij} = 0$?

After the reduction of u_i by $\delta := \pi_{ij}$ we obtain $(u_i - \pi_{ij}) + v_j = w_{ij}$ such that the edge (i, j) would be added to E_M^* . Since we do not wish to delete some edge from E_M^* , we have to modify the node weights of the other nodes in T in an appropriate manner.



We perform

$$\begin{aligned} u_i &:= u_i - \delta & \forall i \in B_T & \text{ and} \\ v_j &:= v_j + \delta & \forall j \in A_T \end{aligned}$$

Furthermore, we reduce the upper bound.

$$U := U - \delta.$$

Properties:

- At the beginning

$$u_i = U \quad \forall \text{ M-free nodes } i \in B.$$

Always we have $i \in B_T \quad \forall \text{ M-free } i \in B.$

$\Rightarrow$

After the extension step, we have

$$u_i = U \quad \forall \text{ M-free nodes } i \in B.$$

- At the beginning

$$v_j = 0 \quad \forall \text{ M-free nodes } j \in A.$$

Always we have $j \notin A_T \quad \forall \text{ M-free } j \in A$

$\Rightarrow$

After the extension step, we have

$$v_j = 0 \quad \forall \text{ M-free nodes } j \in A$$

Since δ should be chosen such that

- i) at least one edge is added to E_M^* and
 ii) δ should be chosen as small as possible,
 we define

$$\delta := \min \{ \pi_{ij} \mid i \in B_T \text{ and } j \in A \setminus A_T \}.$$

If $\delta \geq u$ then the algorithm terminates.

Exercise

Work out the algorithm in detail.

Let P be a path in G_M . The unbalance $\pi(P)$ of the path P is defined as follows.

$$\pi(P) := \sum_{e \in P} \pi_e.$$

Next we will show that after the extension step with respect to the new upper bound and the new input graph for the next search step the invariants 1-3 are fulfilled.

Lemma 1.2

Let $P = v_0, v_1, \dots, v_k$ with $v_0 \neq s$ and $v_k \neq t$ be a simple alternating path in G_M . Then

$$\Delta(P) = \begin{cases} u_{v_0} + v_{v_k} - \pi(P) & \text{if } v_0 \in B, v_k \in A \\ u_{v_0} - u_{v_k} - \pi(P) & \text{if } v_0, v_k \in B \\ -v_{v_0} - u_{v_k} - \pi(P) & \text{if } v_0 \in A, v_k \in B \\ -v_{v_0} + v_{v_k} - \pi(P) & \text{if } v_0, v_k \in A. \end{cases}$$

Proof:

Definition $\Rightarrow$

$$\begin{aligned}\Delta(P) &= \sum_{e \in P \cap E_M \setminus M} w_e - \sum_{e \in P \cap M} w_e \\ &= \sum_{(i,j) \in P \cap E_M \setminus M} (u_i + v_j - \pi_{ij}) \\ &\quad - \sum_{(j,i) \in P \cap M} (v_j + u_i - \pi_{ji})\end{aligned}$$

Since $\pi_{ji} = 0$ for $(j,i) \in M$ there holds

$$= \underbrace{\left(\sum_{(i,j) \in P \cap E_M \setminus M} (u_i + v_j) - \sum_{(j,i) \in P \cap M} (v_j + u_i) \right)}_D - \pi(P)$$

Since P is an alternating path, up to the first summand with respect to the first edge on P and the second summand with respect to the last edge on P all summands vanish. Hence

$$D = \begin{cases} u_{v_0} + v_{v_k} & \text{if } v_0 \in B, v_k \in A \\ u_{v_0} - u_{v_k} & \text{if } v_0, v_k \in B \\ -v_{v_0} - u_{v_k} & \text{if } v_0 \in A, v_k \in B \\ -v_{v_0} + v_{v_k} & \text{if } v_0, v_k \in A \end{cases}$$

This proves the lemma.

Lemma 1.3

The algorithm fulfills always the following invariants:

- i) $u_i = u$ for all M -free nodes $i \in B$,
- ii) $v_j = 0$ for all M -free nodes $j \in A$, and
- iii) $\pi_e \geq 0$ for all edges $e \in E_M$.

Proof:

Definition $\Rightarrow$

At the beginning, i.e., $M = \emptyset$ all invariants are fulfilled.

Goal:

To prove the following:

invariants fulfilled before the search and the extension step, respectively

$\Rightarrow$

The invariants are fulfilled after the search and the extension step, respectively.

The search step never changes the node weights or the upper bound. Hence, the invariants remain fulfilled after the search step.

We have shown above that the extension step maintains the invariants i) and ii).

δ is chosen in such a way such that the modification of the node weights never violate the invariant (ii). ■

Theorem 1.6

The algorithm MAXWEIGHTMATCHING implemented as described above terminates with a maximum weighted matching $M_{\max}$.

Proof:

It is clear that the algorithm terminates with a matching $M_{\max}$.

Theorem 1.4 $\Rightarrow$

It suffices to show that always an augmenting path P with maximal gain $\Delta(P)$ is augmented by the algorithm and that after the termination of the algorithm no augmenting path P with $\Delta(P) > 0$ exists.

Assume that the algorithm augment the M -augmenting path P with $\Delta(P) > 0$ although an M -augmenting path Q with $\Delta(Q) > \Delta(P)$ exists.

Let

$$P = i, \dots, j \quad \text{and} \quad Q = p, \dots, k.$$

Lemma 1.2 $\Rightarrow$

$$\Delta(Q) = u_p + v_z - \pi(Q) \quad \text{and}$$

$$\Delta(P) = u_i + v_j - \pi(P).$$

Lemma 1.3 $\Rightarrow$

$$\pi(Q) \geq 0, \quad u_i = u_p \quad \text{and} \quad v_j = v_z = 0.$$

Since $\pi(P) = 0$ we obtain

$$\Delta(Q) + \pi(Q) = \Delta(P)$$

and hence,

$$\Delta(Q) \leq \Delta(P)$$

or contradiction.

After the termination of the algorithm there hold

$$u_i \leq 0 \quad \text{for all } M_{\max}\text{-free nodes } i \in B \text{ and}$$

$$v_j = 0 \quad \text{for all } M_{\max}\text{-free nodes } j \in A.$$

Lemmas 1.2 and 1.3 $\Rightarrow$

$$\Delta(P) = u_i + v_j - \pi(P) \leq -\pi(P) \leq 0$$

for all $M_{\max}$ -augmenting paths $P = (i, \dots, j)$.

$\Rightarrow$

$\nexists$ $M_{\max}$ -augmenting path P with $\Delta(P) > 0$. ■

Exercise:

Show that the algorithm MAXWEIMATCHING can be implemented such that its run time is $O((|A| + |B|)^3)$.

1.3. Strong connectivity

Let $G = (V, E)$ be a directed graph. Two nodes $v, u \in V$ are strongly connected if there is a path from v to u and also a path from u to v in G .

The relation "strongly connected" is an equivalence relation. Hence, we can partition V into the equivalence classes V_i , $1 \leq i \leq r$ with respect to the relation "strongly connected".

Let

$$E_i := \{ (v, w) \in E \mid v, w \in V_i \}.$$

The graphs $G_i := (V_i, E_i)$, $1 \leq i \leq r$ are the strongly connected components of $G = (V, E)$.

G is strongly connected if it has only one strongly connected component.

Properties:

- Each node in V is in exactly one strongly connected component.
- Edges with the property that its end nodes are in distinct connected components are in no strongly connected component. Such an edge is called 2c-edge.

The reduced graph $G_{\text{red}} = (V_{\text{red}}, E_{\text{red}})$ is defined by

(3)

$$V_{\text{red}} := \{ v_i \mid G_i = (V_i, E_i) \text{ is a strongly connected component} \}$$

$$E_{\text{red}} := \{ (v_i, v_j) \mid \exists u \in V_i, v \in V_j : (u, v) \in E \}$$

Note that G_{red} is acyclic. Hence, G_{red} can be sorted topologically.

Goal:

The computation of the strongly connected components $G_i = (V_i, E_i)$, $1 \leq i \leq r$ of a given directed graph $G = (V, E)$.

Assume that G_{red} is topological sorted and that the strongly connected components are numbered with respect to the topological sorting of G_{red} .

$\Rightarrow$

If there is a 2c-edge from component V_i to component V_j then $i < j$.

Idea:

(1) $k := r$;

(2) while $k > 0$

do

- Compute an arbitrary node $u_k \in V_k$, the reference node of V_k ;

- $\text{DFS}(u_k)$;

od: $k := k - 1$

The nodes visited during $\text{DFS}(u_k)$ are exactly the nodes in V_k .

~)

backward construction of the strongly connected components of G .

Question:

How to obtain the reference nodes in the correct order?

We say that the nodes in $G = (V, E)$ are well numbered if starting with number 1 a node $v \in V$ obtains the next free number at the moment when the node v leaves the DFS-stack.

Lemma 1.4

Let $G = (V, E)$ be a directed graph and let the nodes of G be well numbered by $\text{num}: V \rightarrow \{1, 2, \dots, n\}$. Then for all $u, v \in V$ with $\text{num}(u) > \text{num}(v)$ the following holds:

If there is in G a path P from v to u such that $\text{num}(w) \leq \text{num}(u)$ for all w on P then both nodes u and v are in the same strongly connected component.

Proof:

- 1) We have to prove that there is a path from u to v in G .

Assume that there is no such a path in G .

$\Rightarrow$

v is not a node in the subtree with root u of the DFS-tree.

- 2) $\text{num}(v) < \text{num}(u)$ and $\nexists$ path from u to v

$\Rightarrow$

At the moment when v enters the DFS-stack the node u cannot be already considered.

Let

$$P = v = w_1, w_2, \dots, w_k = u$$

Let $i \geq 1$ maximal such that

- directly after the entering of v there holds: w_i is already considered.

Since $v = w_1$ is already considered such i exists.

Since $u = w_k$ is not already considered, we have $i < k$. Two cases can arise:

Case 1:

At the moment when v enters the DFS-stack the node w_i is not in the stack.

Then w_i has left the DFS-stack before the entering of v . But before the performance of $\text{Pop}(w_i)$ the edge (w_i, w_{i+1}) is considered by DFS

$\Rightarrow$

w_{i+1} is already considered

a contradiction to the choice of w_i .

Case 2:

w_i is in the DFS-stack when v enters the stack.

Since there is a path from w_i to the node u , $\text{Push}(u)$ is performed by DFS before $\text{Pop}(w_i)$ and hence, also $\text{Pop}(u)$ is performed before $\text{Pop}(w_i)$

$\Rightarrow$

$\text{num}(u) < \text{num}(w_i)$ a contradiction.

■

Let $G' = (V, E')$ be the backward graph of G ; i.e.,

$$E' := \{(u, w) \mid (w, v) \in E\}.$$

Observation:

The strongly connected components of G and G' are the same.

Goal:

The development of a rule for the determination of the reference node for the computation of the strongly connected components of G' .

Lemma 1.5

Let $G = (V, E)$ be a directed graph and let the nodes in V be well numbered by $\text{num}: V \rightarrow \{1, \dots, n\}$

Apply the following rule (R) for the determination of the reference nodes for the computation of the strongly connected components of $G' = (V, E')$.

(R) Choose the node u with $\text{num}(u)$ is maximal within all nodes which are not assigned to a component.

Then u is in a strongly connected component such that all successor components with respect to the reduced graph G'_{red} are already constructed

Proof:

(R) $\Rightarrow$

For all nodes v visited during DFS on G' with start node u for the first time exists in G a path P from v to u with

$$\text{num}(w) \leq \text{num}(u) \quad \forall w \text{ on } P.$$

Lemma 1.4 $\Rightarrow$

If $\text{num}(u) > \text{num}(x)$ then both nodes u and v are in the same connected component.

Otherwise, we obtain by rule (R) that the strongly connected component containing v is already constructed.

~>

- DFS on G for the computation of num .
- DFS on G' for the computation of the strongly connected components.

Theorem 1.7

Let $G = (V, E)$ be a directed graph. Then we can compute the strongly connected components of G in time $O(|V| + |E|)$.

2. Algorithms on strings

References:

- Maxime Crochemore, Wojciech Rytter, Text Algorithms, Oxford University Press, 1994.
- Maxime Crochemore, Wojciech Rytter, Jewels of Stringology, World Scientific, 2003.
- Bill Smyth, Computing Patterns in Strings, Pearson - Addison Wesley, 2003.
- Maxime Crochemore, Christophe Hancart, Thierry Lecroq, Algorithms on Strings, Cambridge University Press, 2007.
- Norbert Blum, Algorithmen und Datenstrukturen, 2. Auflage, Oldenbourg, 2013.
(in German).

Finding all occurrences of a pattern in a text is a problem that arises frequently in text-editing programs. Also with respect to the investigation of biological sequences such questions have to be solved. The development of solutions of such questions will be the topic of this part of the lecture.

We shall consider the following string matching problem.

(3)

Given a text string $x \in \Sigma^+$ where Σ is a finite alphabet, $|x| = n$ and a pattern string $y \in \Sigma^+$, $|y| = m$, we wish to find the first (or all) occurrence of the pattern string y in the text string x .

2.1 The algorithm of Knuth, Morris and Pratt

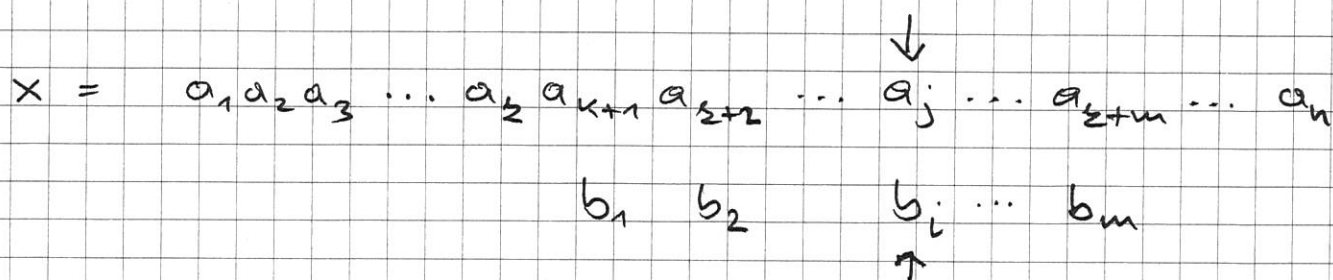
It is obvious how to solve the string matching problem in $O(n \cdot m)$ time. For doing this it suffices to test all possible $n - m + 1$ positions in x if a substring z with $z = y$ begins.

Each test can be performed in $O(m)$ time such that a total time of $O(n \cdot m)$ is needed.

Question:

Can this naive method be improved?

To answer this question let us analyze the naive approach more in detail. We shift simultaneously two pointers over the text and the pattern string



We distinguish two cases:

Case 1: $a_j = b_i$

$$\Rightarrow a_{j-i+1} a_{j-i+2} \dots a_j = b_1 b_2 \dots b_i$$

Both pointers are shifted to the next position

Case 2: $a_j \neq b_i$

$$\Rightarrow a_{j-i+1} a_{j-i+2} \dots a_{j-1} = b_1 b_2 \dots b_{i-1}$$

and

$$a_j \neq b_i$$

The naive approach pushes the pattern string one position to the right. This means that k is increased by one. Then, both pointers are shifted back by $i-1$ positions.

$\leadsto$

The knowledge $a_{j-i+1} a_{j-i+2} \dots a_{j-1} = b_1 b_2 \dots b_{i-1}$ is not taken into account. Moreover, this knowledge is forgotten.

Question:

What would be the maximal gain if the knowledge would be used?

Idea:

Do not shift back the text string pointer.

$\Rightarrow$

The pattern string has to be pushed to the right

such that

- 1) to the left of the text string pointer, pattern and text are the same and
- 2) the first property is not fulfilled with respect to each shorter right shift of the pattern string.

Note that a right shift of the pattern string corresponds to a left shift of the pattern string pointer. This means that we have to

- choose the shortest possible left shift of the pattern string pointer such that to the left of both pointers, both strings are identical.

Goal:

Computation of the correct new position of the pattern string pointers within the pattern string.

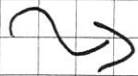
For getting this position we define for $1 \leq r \leq m$

$$H(r) := \max_{e \leq r} \{ e \mid b_1 b_2 \dots b_{e-1} \text{ is suffix of } b_1 b_2 \dots b_{r-1} \}.$$

$H(r) - 1$ is the length of the longest prefix of $b_1 b_2 \dots b_{r-2}$ which is also suffix of $b_1 b_2 \dots b_{r-1}$.

Assume that $H(r)$, $1 \leq r \leq m$ is already computed. Then we can refine the naive method in the following way:

- ④
- Move the pattern string such that $b_{H(i)}$ is below a_i and do not change the text string pointer. After doing this, the pattern string pointer points to $b_{H(i)}$.



Algorithm KMP

Input: text string $x = a_1 a_2 \dots a_n$,
pattern string $y = b_1 b_2 \dots b_m$,
table H .

Output: k minimal such that

$$a_{k+1} a_{k+2} \dots a_{k+m} = b_1 b_2 \dots b_m$$
 if such a k exists and "no occurrence" otherwise.

Method:

$i := 1; j := 1;$

while $i \leq m$ and $j \leq n$

do

while $i > 0$ and $b_i \neq a_j$

do

$i := H(i)$ (* Note $H(1) = 0$ *)

od

$i := i + 1; j := j + 1$

od;

if $i > m$

then

$k := j - (m+1)$

else

"no occurrence"

fi.

The cond in the inner while-loop is a "conditional and" and compares b_i with a_j only if $i > 0$. Each iteration of the inner while-loop moves the pattern string about $(i - H(i))$ position to the right until $i = 0$ or $b_i = a_j$.

In the first case, no nonempty prefix of y is suffix of $a_1 a_2 \dots a_j$. In the second case there holds

$$b_1 b_2 \dots b_i = a_{j-i+1} a_{j-i+2} \dots a_j.$$

The outer while-loop increases synchronously both pointers.

Lemma 2.1

The algorithm KMP is correct.

Proof:

We have to show that the algorithm KMP computes the smallest k with

$$a_{k+1} a_{k+2} \dots a_{k+m} = b_1 b_2 \dots b_m$$

if such a k exists. and otherwise establishes

"no occurrence".

) If KMP leaves the outer while-loop because of $i = m+1$ then it follows by construction

a) $j \leq n+1$ and

b) $a_{k+1} a_{k+2} \dots a_{k+m} = b_1 b_2 \dots b_m$.

This can be proved by induction.

We have to show that there is no $k' \leq n-m$ with

i) $a_{k'+1} a_{k'+2} \dots a_{k'+m} = b_1 b_2 \dots b_m$ and

ii) $k' < k$ if the output of KMP is k .

Assume that such a k' exists. To obtain a contradiction we investigate the run of the algorithm KMP.

) Since the output of KMP is $k > k'$ or "no occurrence", the following situation arises:

$x = a_1 a_2 \dots a_k a_{k+1} \dots a_{k'} \dots a_q \dots a_j \dots a_{k+m} \dots a_n$	
	$\neq$
before shift	$b_1 b_2 \dots b_i \dots b_m$
after shift	$b_1 b_2 \dots b_m$

- Immediately before the execution of the block of the inner while-loop, the first symbol of the pattern string y is to the left of $a_{k'+1}$

and after that behind $a_{k'+1}$.

$$\Rightarrow k < k' < j-1.$$

Because of

$$a_{k'+1} a_{k'+2} \dots a_j \dots a_{k'+m} = b_1 b_2 \dots b_m$$

and

$$a_{k+1} a_{k+2} \dots a_{k+(i-1)} = b_1 b_2 \dots b_{i-1}$$

there holds:

$b_1 b_2 \dots b_{j-k'-1}$ is suffix of $b_1 b_2 \dots b_{i-1}$.

But because of the situation above there holds

$$j - k' > H(i).$$

This contradicts the definition of $H(i)$. Hence, our assumption is wrong such that the lemma is proved. ■

Next we shall analyze the needed time and the needed space of the algorithm KMP.

- The text string pointer j is never reduced.
- An increase of the pattern string pointer i by one implies the increase of the text string pointer j by one.

$\Rightarrow$

④

The pattern string pointer is increased by one at most n times.

- A decrease of the pattern string pointer in the inner while-loop requires the corresponding increase of i .

⇒

The total time used by the algorithm KMP is $O(n)$.

The algorithm KMP needs additional space for

- the two pointers and
- the table H .

⇒

additional space is $O(m)$.

It remains to explain how to compute the table H efficiently. The idea is to compute H inductively.

1) By the definition there hold
 $H(1) = 0$ and $H(2) = 1$.

2) Assume that $H(1), H(2), \dots, H(i)$, $i \geq 2$ are already computed. To compute $H(i+1)$, we distinguish two cases:

a) $b_i = b_{H(i)}$.

It follows directly $H(i+1) = H(i) + 1$.

b) $b_i \neq b_{H(i)}$.

We have the following situation:

$b_1 b_2 \dots b_{H(i)-1}$ is suffix of $b_1 b_2 \dots b_{i-1}$ but $b_i \neq b_{H(i)}$

i.e., $b_1 b_2 \dots b_{H(i)}$ is not a suffix of $b_1 b_2 \dots b_i$.

We iterate the approach above and check $H(H(i))$.

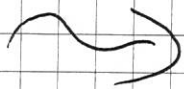
Assumption $\Rightarrow H(H(i))$ is already computed.

• If $b_i = b_{H(H(i))}$ then $H(i+1) = H(H(i)) + 1$.

Otherwise, we repeat the approach above.

This is done as long until

- $H(i+1)$ is computed or
- the beginning of the pattern string y is reached.



Algorithm COMPUTATION OF H

Input: $y = b_1 b_2 \dots b_m$

Output: table H .

Method:

$H(1) := 0; H(2) := 1;$

for i from 2 to $m-1$
do

$j := H(i);$

while $j > 0$ and $b_i \neq b_j$
do

$j := H(j)$

od;

$H(i+1) := j+1$

od.

Analogously to the analysis of the algorithm KMP we can prove that the needed time is $O(m)$.

Exercise:

a) Prove the correctness of the algorithm COMPUTATION OF H . Give a formal proof that the needed time is $O(m)$.

b) Give an exact analysis of the number of comparisons performed by the algorithm KMP.

c) We have computed the values

$$H(r) = \max_{e < r} \{e \mid b_1 b_2 \dots b_{e-1} \text{ is suffix of } b_1 \dots b_{r-1}\}.$$

for $1 \leq r \leq m$. It is obvious that in the case $b_{H(i)} = b_i$ there holds $b_{H(i)} \neq a_i$.

Hence, instead $H(r)$ it would be better to compute the value $\text{Next}(r)$ where

$$\text{Next}(r) = \max_{e < r} \{ l \mid b_1 b_2 \dots b_{e-1} \text{ is suffix of } b_1 b_2 \dots b_{r-1} \text{ and } b_e \neq b_r \}$$

Develop an efficient algorithm for the computation of $\text{Next}(r)$ for $1 \leq r \leq m$. Prove the correctness of your algorithm and analyze its needed time. Modify the algorithm KMP such that the table Next is used instead of the table H .

The algorithm KMP computes the first occurrence of the pattern string y in the text string x . How to compute all occurrences of y in x ? The naive approach which after finding y in x moves the pattern y one position to the right and to start KMP again can need $O(m \cdot n)$ time.

Example:

$$x = \underbrace{aa \dots a}_{n \text{ times}}$$

$$y = \underbrace{aa \dots a}_{m \text{ times}}$$

The above approach would perform m comparisons for each of the $n - m + 1$ occurrences of y in x . Hence, the total number of comparisons would be

$$m(n - m + 1) = mn - m^2 + m$$

Goal:

To extend the algorithm KMP such that all occurrences of y in x can be computed in $O(n + m)$ time.

Idea:

Extend the table H by

$$H(m+1) := \max_{\ell \leq m+1} \{ \ell \mid b_1 b_2 \dots b_{\ell-1} \text{ is suffix of } b_1 b_2 \dots b_m \}$$

Now we can use $H(m+1)$ for the computation of the minimal right shift of the pattern string such that the algorithm KMP can be continued at the termination point of the last run.

Exercise:

Modify the algorithm KMP such that all occurrences of y in x are computed in $O(n + m)$ time. Extend also the algorithm COMPUTATION OF H such that $H(m+1)$ is also computed.

Reference:

- D.E. Knuth, J.H. Morris, V.R. Pratt, Fast Pattern Matching in Strings, SIAM J. Comput 6 (1977), 323 - 350.

Altogether, we have obtained the following theorem.

Theorem 2.1

- 1) The algorithm KMP solves the string matching problem in $O(n+m)$ time where n is the length of the text string and m is the length of the pattern string. An additional space of size $O(m)$ is used.

Our goal is now to improve the algorithm KMP. Note that in the inner while-loop of the algorithm KMP, the pattern string pointer i can be reduced without increasing the text string pointer j . This happens as long as $b_i \neq a_j$.

~>

Question:

Is it possible to compute a table F such that the pattern string pointer obtains directly the correct position and after doing that, both pointers are moved one position to the right?

To get such a table, we need for $1 \leq r \leq m, c \in \Sigma$

$$F(r, c) := \max_{e \leq r} \{ e \mid b_1 b_2 \dots b_e \text{ is suffix of } b_1 \dots b_{r-1} c \}$$

Exercise

Assume that the table F is already computed. Modify the algorithm KMP such that the table F is used instead of the table H .

Analyze the number of comparisons performed by the modified algorithm. (5)

We have to explain how to compute the table F efficiently. The definition of F implies directly

- $F(0, c) = 0$ for all $c \in \Sigma$,
- $F(1, b_1) = 1$ and
- $F(1, c) = 0$ for all $c \in \Sigma \setminus \{b_1\}$.

For $r > 1$ it follows from the definition of F

$$F(r, c) = \begin{cases} H(r) & \text{if } b_{H(r)} = c \\ F(H(r), c) & \text{otherwise.} \end{cases}$$

$\leadsto$

Algorithm COMPUTATION OF F

Input: $y = b_1 b_2 \dots b_m$, table H

Output: table F

Method:

(1) $F(0, b_1) := 0$; $F(1, b_1) := 1$;

(2) for all $c \in \Sigma \setminus \{b_1\}$
do

$F(0, c) := 0; F(1, c) := 0$

od;

(3) for i from 2 to m

do

for all $c \in \Sigma$

do

if $b_{H(i)} = c$

then

$F(i, c) := H(i)$

else

$F(i, c) := F(H(i), c)$

fi

od

od.

time analysis:

Steps (1) and (2)

$O(|\Sigma|)$

Step (3)

$O(|\Sigma| \cdot m)$

in total

$O(|\Sigma| \cdot m)$

The modified algorithm uses less comparisons than the algorithm KMP. What we pay is the additional time $O(|\Sigma| \cdot m)$ for the computation of the table F and the additional space $(|\Sigma| \cdot m)$ to store the table F .

2.2 Suffix trees

- 1) The algorithm KMP prepares the pattern string such that all occurrences of the pattern string can be found in the text string in linear time. In the case the occurrences of many pattern string have to be searched in a text string it could be better to prepare the text string such that each pattern string can be found efficiently in the text string.

Goal:

The development of a data structure for the text string which enables the solution of string matching problems. In particular, all occurrences of a pattern string y with $|y| = m$ should be found in time $O(m + A(y))$ where $A(y)$ is the number of occurrences of y in the text string x .

References:

- Dan Gusfield, Algorithms on Strings, Trees, and Sequences, Cambridge University Press, 1997, 89 - 107.
- Esko Ukkonen, Online - Construction of Suffix Trees, Algorithmica 14 (1995), 248 - 260.

٤

First we need some definitions.

A trie with respect to an alphabet Σ , $|\Sigma|=k$ is a tree $T = (V, E)$ such that:

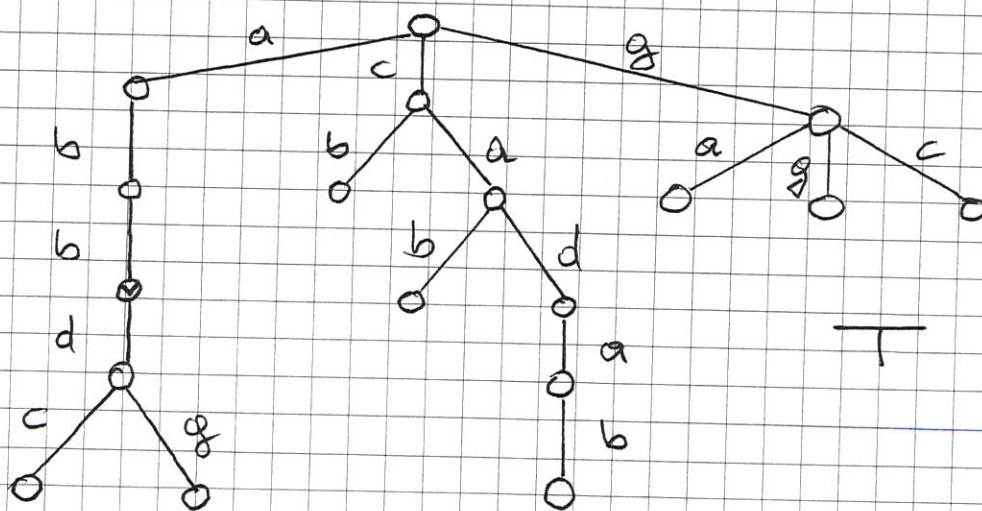
- (i) Each inner node has outdegree $\leq k$.
- (ii) The outgoing edges of an inner node are marked with pairwise distinct elements of Σ .

A path P from the root of the trie T to a leaf v corresponds to the string which is obtained by the concatenation of edge labels.

A tree T represents a set $S(T)$ of strings.

These strings correspond to the paths from the root of T to a leaf.

Example:



$$S(T) = \{abbd, abbdg, cb, cab, cadab, ga, gg, gc\}$$

The set $S(T)$ above is prefix free, i.e., no element of $S(T)$ is prefix of another element of $S(T)$. If we add to $S(T)$ the string abb then we obtain a set of strings which is not prefix free since abb is a prefix of the string $abbd$.

If we mark inner nodes of the trie where such a prefix ends then we can also represent a set S of strings which is not prefix free. The paths from the root of the trie to a marked node or to a leaf correspond uniquely to the elements of S .

Example (continuation):

If we mark in the trie T above the node v then we obtain a trie T' representing the set

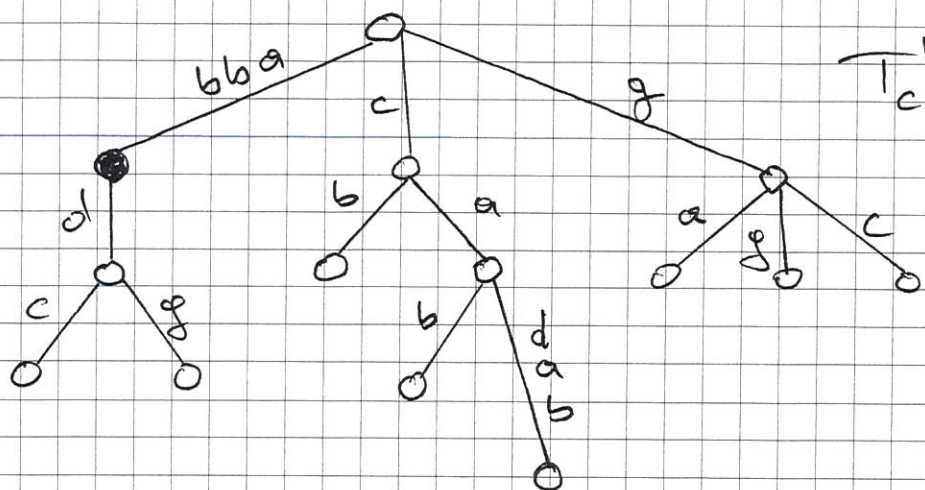
$$S(T') = \{abb, abbd, abbdg, cb, cab, cadab, ga, gg, gc\}$$

of strings.

◇

Given a trie T we obtain the compact trie T_c by combining all paths P which contain only nonmarked nodes of outdegree one to one edge and marking this edge with the string which corresponds to P .

Example (continuation):

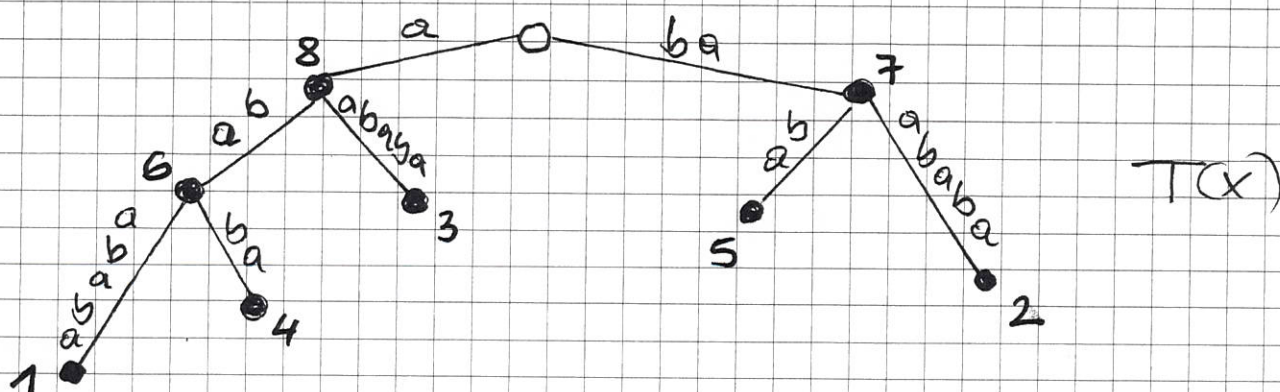


The compact tree T_c' corresponding to T' .

A suffix tree $T(x)$ for a string $x = a_1 a_2 \dots a_n \in \Sigma^n$ is a compact trie with respect to the alphabet Σ which contains n marked nodes. The nodes have pairwise distinct numbers in $\{1, 2, \dots, n\}$. The path from the root to the node with number i corresponds to that suffix of x which starts in the i -th position of x , i.e., $a_i a_{i+1} \dots a_n$.

Example:

Let $x = abaababa$



Next we shall show how to solve the string matching problem efficiently with help of a suffix tree $T(x)$.

Instance: text string $x := a_1 a_2 \dots a_n$, a suffix tree $T(x)$ and a pattern string $y := b_1 b_2 \dots b_m$.

task: Compute all position in x in which the pattern string y begins.

To solve this problem, we start in the root of $T(x)$ and construct the longest path P in $T(x)$ with edge marking $b_1 b_2 \dots b_j$, $0 \leq j \leq m$. Two cases can arise:

Case 1: $j < m$

Then y is not a substring of x .

Case 2: $j = m$

We consider that node v in $T(x)$ such that

- the path P ends on the incoming edge of v or in the node v .

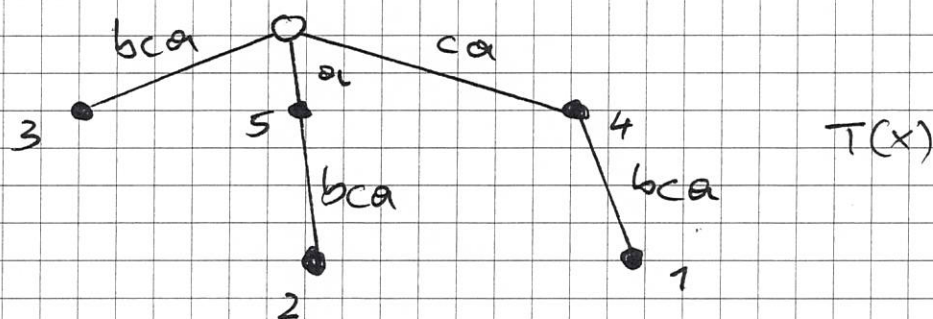
The numbers of the marked nodes in the subtree with root v denote exactly those positions in x in which the pattern string y begins.

Exercise

Give a formal proof of the correctness of the solution of the string matching problem which uses a suffix tree for the considered text string.

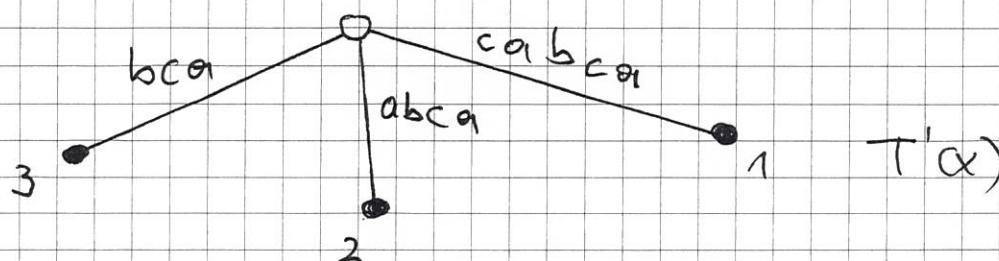
Before we shall develop a method for the construction of a suffix tree $T(x)$ for a given string x we shall simplify the problem a little bit.

For doing this let us consider the following suffix tree $T(x)$ for the string $x = cabca$.



$T(x)$ contains some nodes of outdegree one. Since $T(x)$ is compact all these nodes are marked.

By collecting in $T(x)$ all maximal paths on which there are only nodes of outdegree one to one edge attaching to each such created edge that string which corresponds to the collected path, we obtain the following tree $T'(x)$:



$T'(x)$ is the implicit suffix tree for the string x . (6)

2.2.2 The construction algorithm

Given a string $x := a_1 a_2 \dots a_n$ we shall develop an algorithm for the construction of a suffix tree $T(x)$. For doing this an implicit suffix tree $T'(x)$ for the string x is constructed first. Then $T'(x)$ will be extended to a suffix tree $T(x)$ for x .

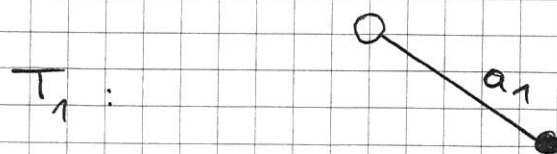
Starting with the prefix a_1 , for each prefix $a_1 a_2 \dots a_i$ of x an implicit suffix tree T_i is constructed. Then, given an implicit suffix tree T_n for x , a suffix tree $T(x)$ for x is constructed.



The algorithm partitions into n phases. In Phase i , $1 \leq i \leq n$ an implicit suffix tree T_i for the prefix $a_1 a_2 \dots a_i$ of x is constructed.

Phase 1:

T_1 contains only one edge which is marked by a_1 ; i.e.,



Assume that for $a_1 a_2 \dots a_i$, $i \geq 1$ an implicit suffix tree is constructed.

Phase $i+1$:

The $(i+1)$ st phase is separated into $i+1$ extension steps. The j th extension step treats the suffix $a_j a_{j+1} \dots a_{i+1}$ of $a_1 a_2 \dots a_{i+1}$.

Extension step j :

Our goal is to take care that starting in the root of the current implicit suffix tree a path with mark $a_j a_{j+1} \dots a_{i+1}$ exists. Such a path can terminate on an edge, in an inner node or in a leaf. To reach this goal we start in the root of the current tree and determine the end of the path P with mark $a_j a_{j+1} \dots a_i$. Since T_i contains such a path there is such a path in the current tree as well. In dependence where this path terminates, we distinguish three cases.

Case 1: P ends in a leaf u .

Konkatenate a_{i+1} at the end of the mark of that edge with end node u .

Case 2: P ends in an inner node v

There are two subcases.

(62)

2.1 The marking of an outgoing edge of v has prefix a_{i+1} .

Then a path with marking $a_j a_{j+1} \dots a_{i+1}$ is already contained in the current suffix tree.

$\leadsto$

Do nothing.

2.2 The marking of no outgoing edge of v has prefix a_{i+1} .

$\leadsto$

v obtains a new direct successor w . The edge (v, w) obtains the mark a_{i+1} . The node w obtains the number j . Then w is a leaf of the current suffix tree.

Case 3: P ends on an edge e .

Let $\beta \in \gamma$ be the marking of the edge e w
 β is suffix of $a_j a_{j+1} \dots a_i$.

$\Rightarrow$

c is the first symbol of the marking of e which does not correspond to $a_j a_{j+1} \dots a_i$.

Since P ends on e and not in a node, the symbol c exists.

Two subcases can arise:

3.1 $c = a_{i+1}$

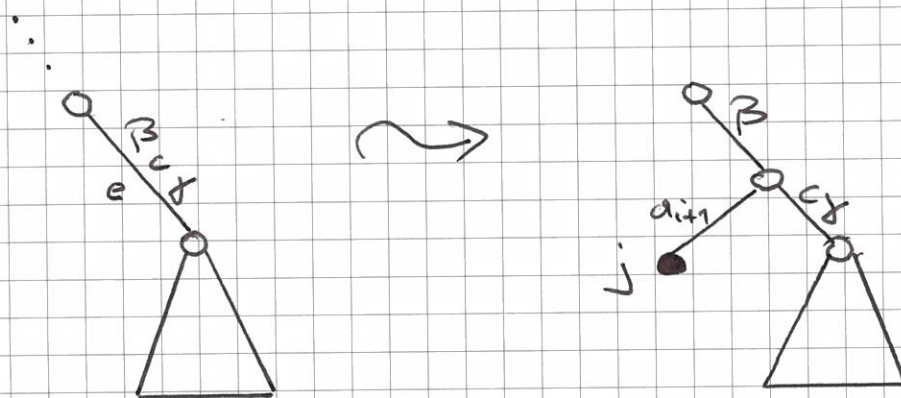
Then a path with marking $a; a_{i+1} \dots a_{i+1}$ already exists.

~>

Do nothing

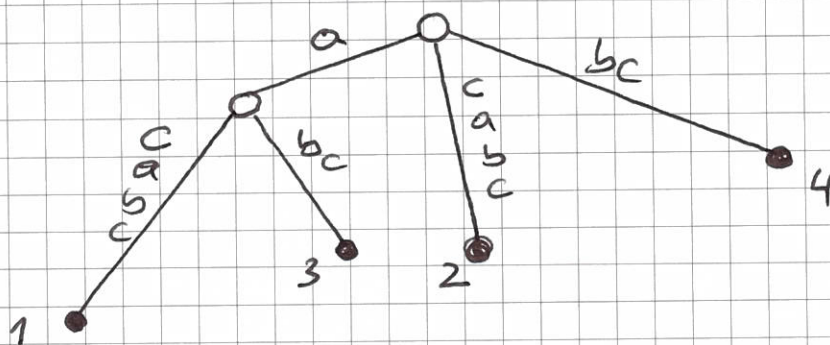
3.2 $c \neq a_{i+1}$

Perform the following transformation:



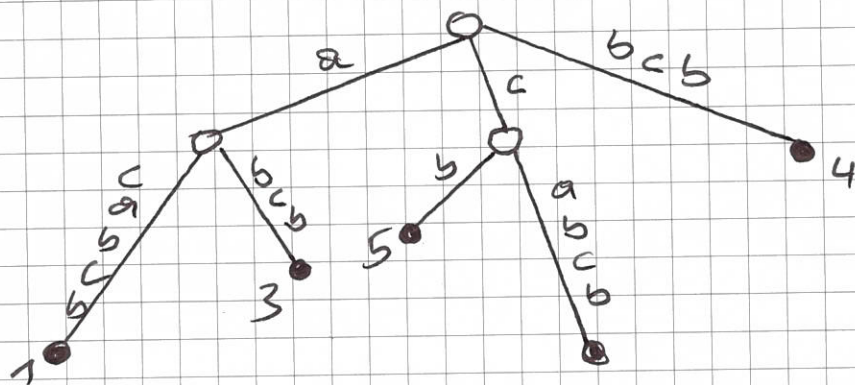
Example:

Consider the following implicit suffix tree for $acabc$:



After adding the symbol b , the implicit suffix tree above is extended to the following suffix tree

for $a < a b c b$:



Goal:

An Estimate of the time and space complexity of the above algorithm.

Space:

- In every leaf of T_n one of the possible n suffixes ends.

$\Rightarrow$

T_n contains at most n leaves and hence, at most $n-1$ inner nodes and at most $2n-2$ edges.

An edge is marked with a substring of x of length $\leq n$.

$\Rightarrow$

The needed space for T_n is $O(n^2)$.

Time analysis:

- Time $L(T_{i+1})$ for the construction of T_{i+1} :

For the extension step j , we have to find the end of the path with marking $a_j a_{j+1} \dots a_i$.

If we start in the root of the current tree and run on the path until the end of the marking $a_j a_{j+1} \dots a_i$ is found the needed time is $O(\text{length of the path})$, i.e.,

$$O(i - (j-1)) = O(i+1 - j).$$

The additional needed time is constant.

$\Rightarrow$

$$\begin{aligned} L(T_{i+1}) &= O\left(\sum_{j=1}^i i+1-j\right) \\ &= O\left(\sum_{j=0}^{i-1} j+1\right) \\ &= O(i^2). \end{aligned}$$

total time:

$$O\left(\sum_{i=1}^n i^2\right) = O(n^3).$$

Goal:

Reduction of the needed time and the needed space.

Subgoal 1: time $O(n^2)$, space $O(n^2)$

If we could find the end of the path with marking $a_j a_{j+1} \dots a_i$ in constant time then the needed time for the construction of T_i would be reduced to $O(i)$

$\Rightarrow$

total time

$$O\left(\sum_{i=1}^n i\right) = O(n^2).$$

Observation:

After the construction of T_i , $1 \leq i \leq n$ we know for $1 \leq j \leq i$ the end of the paths from the root starting with marking $a_j a_{j+1} \dots a_i$.

$\leadsto$

If we store these ends then we obtain these during the construction of T_{i+1} in constant time.

Only for $j = i+1$, we have to find the end of the path with marking a_{i+1} explicitly by starting in the root and running on the path until its end is found. Since its length is one if the path exists, constant time suffices.

$\Rightarrow$

We have reduced the needed time to $O(n^2)$.

The structure of T_n is unchanged

$\Rightarrow$

The needed space is still $O(n^2)$.

Subgoal 2: time $O(n^2)$, space $O(n)$

The following example shows that there is a suffix tree which needs $\Omega(n^2)$ space.

Example:

$x = \text{abcdefghijklmnopqrstuvwxyz}$.

Each suffix starts with a different symbol.

$\Rightarrow$

The root of the implicit suffix tree has 26 sons.

The corresponding edge is marked with the whole suffix.

$\Rightarrow$

The markings need

$$\sum_{j=1}^{26} j = \frac{26 \cdot 27}{2}$$

space.

Idea:

(68)

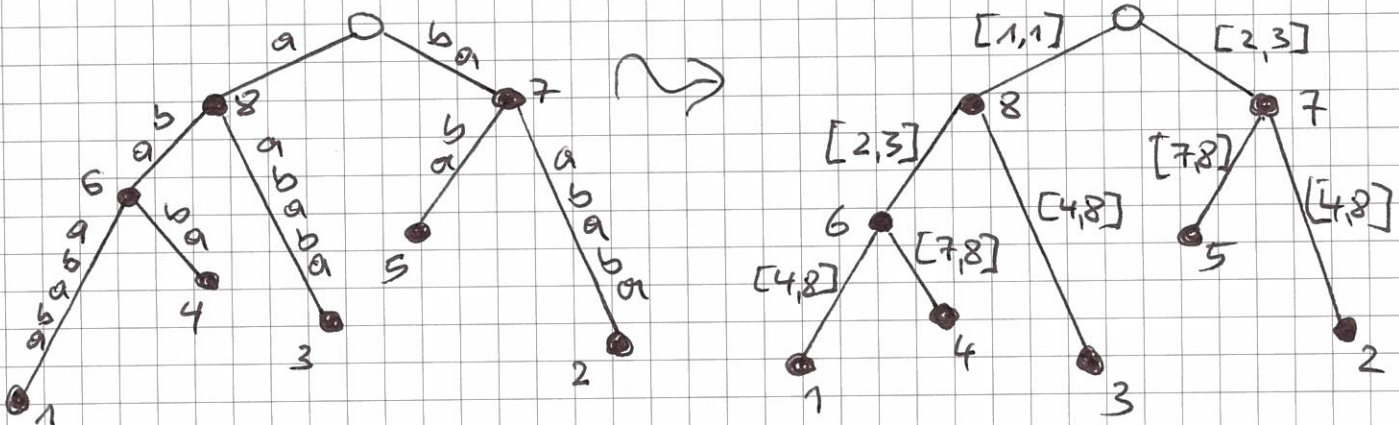
Each marking of an edge corresponds to a substring of the text string x .

~>

Instead of explicitly writing the substring, it suffices to specify the beginning and the end of the substring.

Example:

Let $x = abaababa$. We obtain the following suffix trees:



⇒

Under the assumption that we can store a position in the string in constant space, we have reduced the needed space from $O(n^2)$ to $O(n)$.

Subgoal 3: time $O(n)$, space $O(n)$.

We consider the extension step j again and combine the cases and subcases in dependence of the action performed by the extension step j .

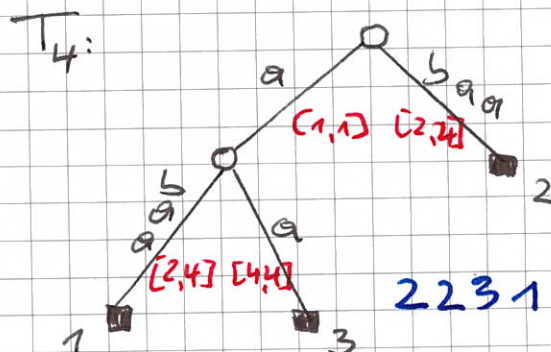
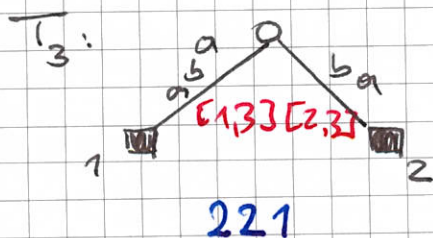
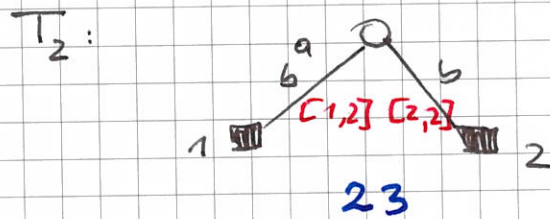
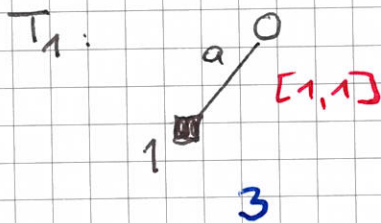
action	Case and subcase
1) Do nothing	2.1 and 3.1
2) Extend leaf marking	1
3) Eventually add an inner node and a leaf	2.2 and 3.2

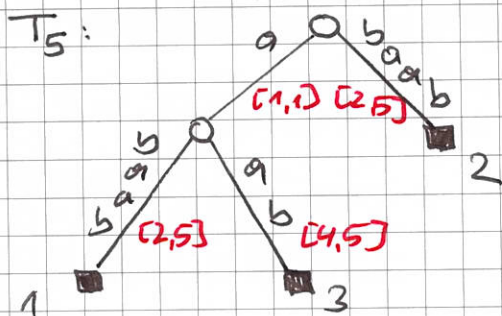
To see what happens, we use the example above and perform the extension steps.

Example (continuation):

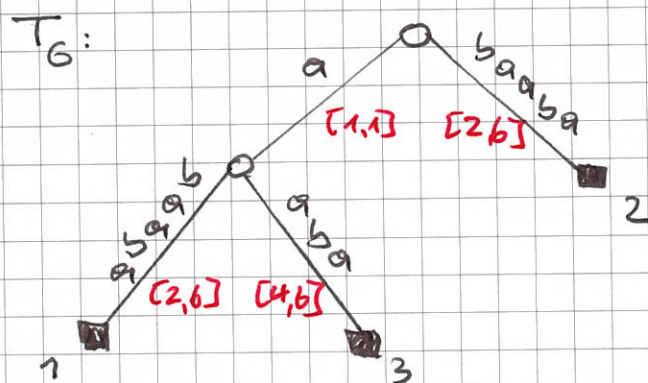
$x = abaaababa$

■ leaf ○ inner node

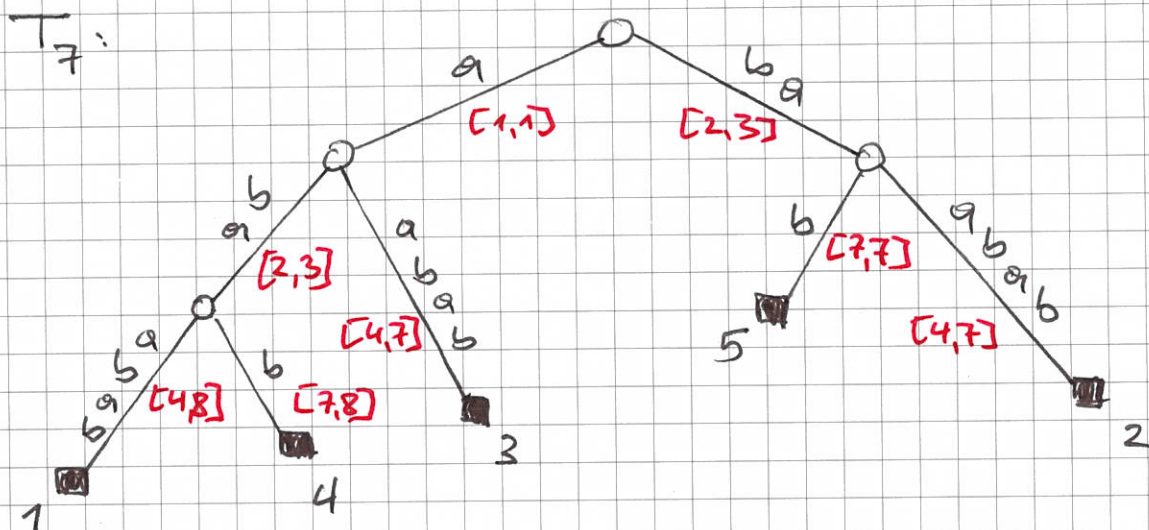




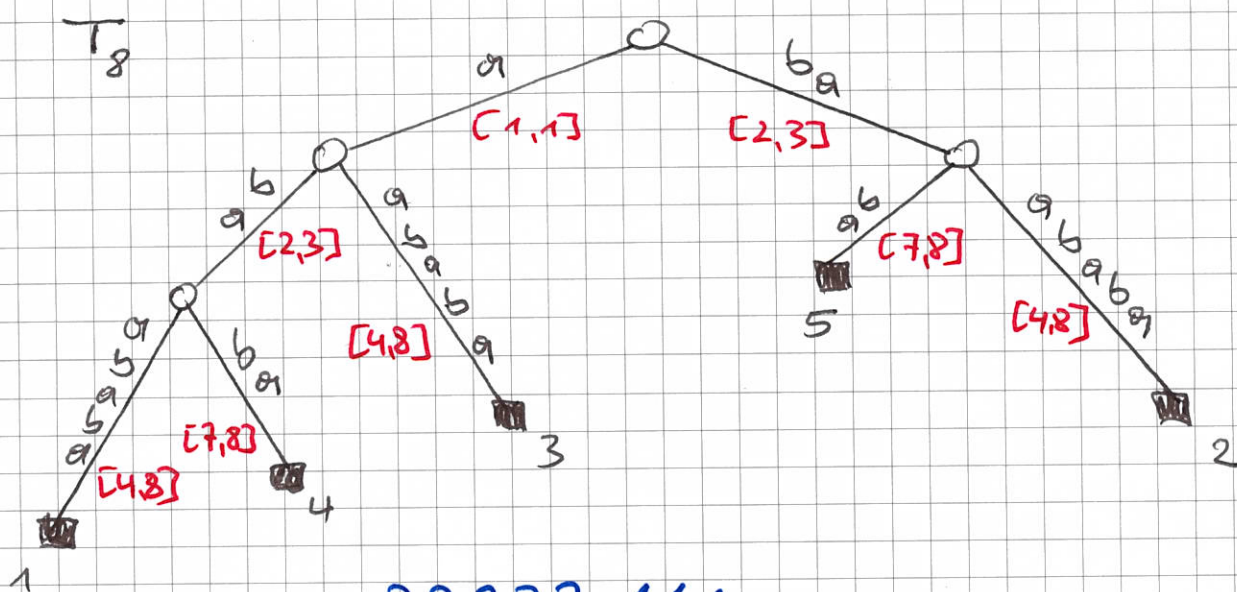
222 11



222 111



222 33 11



22222 111



Observation:

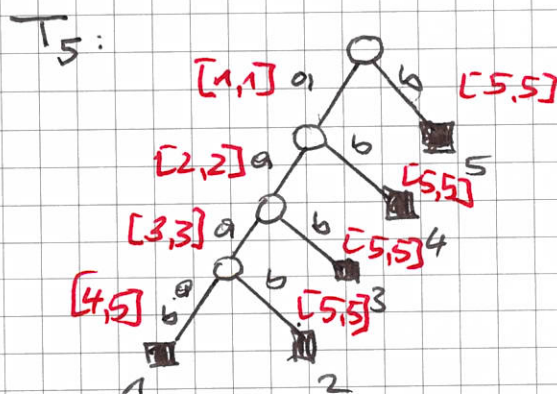
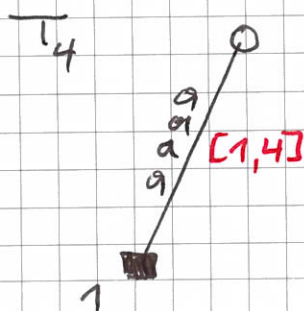
- 1) If the current suffix tree T_i contains a path which starts in the root with marking $c_1 c_2 \dots c_k$ then there exists in T_i for $1 \leq e \leq k$ a path which starts in the root with marking $c_e c_{e+1} \dots c_k$

$\Rightarrow$

If during the construction of T_{i+1} during the extension step j the first action "do nothing" is performed then "do nothing" is also performed for each subsequent extension step l , $j+1 \leq l \leq i+1$.

- 2) After the creation of a leaf with number k , this node remains to be the leaf with number k also in the subsequent suffix trees.

For the illustration of the next observation, we consider for $x = aaaaab$ the construction of T_5 from T_4 .



3) The structure of the sequence of extension steps during the construction of T_{i+1} from T_i is the following:

- Sequence of actions of Type 2 "Extend leaf marking"

followed by a

- sequence of actions of Type 3 "Add eventually an inner node and a leaf."

followed by a

- sequence of actions of Type 1 "Do nothing."

One or two of the sequences described above could be empty.

$\Rightarrow$

For $1 \leq k \leq n$, the following is fulfilled:

At the moment when a leaf with number k is created, a leaf with number ℓ , $1 \leq \ell < k$ already exists.

Observations above $\Rightarrow$

- a) The extensions of leaf markings have not to be performed explicitly. For the

7
incoming edge of a leaf it suffices to specify the beginning of the substring which corresponds to the marking of the edge. The end of this substring is always the current end in x .

$\Rightarrow$

The sequence of actions of Type 2 is not performed explicitly.

b) Assumption:

In T_i the leaves with numbers $1, 2, \dots, k$ exist. The leaf with number $k+1$ does not exist. The goal is the construction of T_{i+1} from T_i .

To start for $k < i$ the explicit construction of T_{i+1} we need

- in T_i the end of the path P which starts in the root with marking $a_{k+1} a_{k+2} \dots a_i$.

As observed above, P ends in an inner node or on an edge.

Because of observation 1) and the choice k , the needed path P is exactly that path considered at the end of the construction of T_i .

$\Rightarrow$

The end of P is known such that the explicit construction of T_{i+1} can be started.

Two cases can arise:

Case 1: An action of Type 1 is performed.

Because of observation 3), the construction of T_{i+1} is ready. Moreover, we know the end of the path starting in the root of T_{i+1} with marking $a_{k+1} a_{k+2} \dots a_{i+1}$ such that the explicit construction of T_{i+2} can be started.

Case 2: An action of Type 3 is performed.

In dependence where the path P ends two sub-cases are possible:

2.1 P ends in an inner node v .

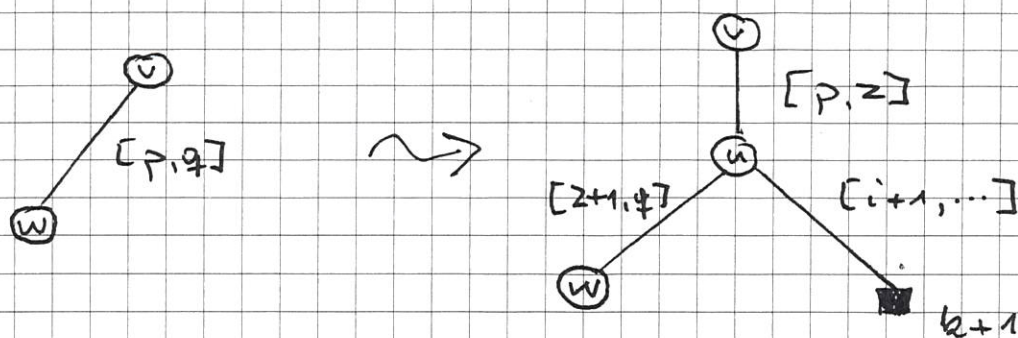
By construction all markings of the outgoing edges of the node v have first symbol $\neq a_{i+1}$.

The node v obtains a new leave b with number $k+1$ as a new son. The edge (v, b) is marked by $[i+1, \dots]$

2.2 P ends on an edge $e = (v, w)$.

Let $[p, q]$ be the marking of the edge e .

Situation and modification:



For the continuation of the construction the end of that path P' is needed which starts in the root and has the marking $a_{z+2} a_{z+3} \dots a_i$.

Observation 1 $\Rightarrow P'$ exists.

Goal:

The development of a data structure which enables the access to the end of P' in constant time.

First we need some notations. Let v be a node in the current suffix tree T . $m(v)$ denotes the marking corresponding to the path from the root of T to the node v .

Let $m(v) := a\alpha$ with $a \in \Sigma$ and $\alpha \in \Sigma^*$.

Then we define

$$sc(v) := \begin{cases} \text{node } w \text{ in } T \text{ with } m(w) = \alpha & \text{if } w \text{ exists} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Observation:

- 1) Let v be a leaf with number p and u be a leaf with number $p+1$ in T . Then $scv = u$.

Assume that each node v in the current suffix tree T contains a pointer to the node scv if this node exists.

Then we can find the end of the path P' in the following way:

Independence if with respect to the path P
Case 2.1 or Case 2.2 above arises, we distinguish two cases.

2.1 P ends in an inner node.

Since each inner node has at least two sons, there are paths with markings

$$a_{z+1} a_{z+2} \dots a_i c \dots \quad \text{and also} \quad a_{z+1} a_{z+2} \dots a_i d$$

with $c \neq d$ from the root to a leaf.

Observation 1 $\Rightarrow$

There are paths with markings

$$a_{z+2} a_{z+3} \dots a_i c \dots \quad \text{and} \quad a_{z+2} a_{z+3} \dots a_i d \dots$$

from the root to a leaf.

⇒

The path P starting in the root with marking $a_{i+2} a_{i+3} \dots a_i$ ends in an inner node

⇒

scv is a node in the current suffix tree T .

Assumption ⇒

We have in T a so-called suffix pointer from v to scv .

Exercise:

Construct an example such that scv has an outgoing edge with the first symbol of its marking is $g \in \Sigma$ but v has not such an edge.

The exercise above shows that two cases can arise:

2.1.1 scv has an outgoing edge with marking $a_{i+1} \dots$

~)

Apply action of Type 1 "do nothing".

Observation 3 ⇒ T_{i+1} is constructed.

We know the end of the path P which starts in the root of T_{i+1} and has the marking

$a_{i+2} a_{i+3} \dots a_{i+1}$

$\Rightarrow$

We can start with the construction of T_{i+2} .

2.1.2 The markings of all outgoing edges of scv have a first symbol $\neq a_{i+1}$.

$\leadsto$

scv obtains a new leaf b with number $i+2$ as new son. The edge (scv, b) obtains the marking $[i+1, \dots]$.

We follow the pointer to $sc(scv)$. If scv is not the root of the current suffix tree then $sc(scv)$ exists.

If scv is the root of the current suffix tree then in the definition of scv the string α would be the empty string ϵ .

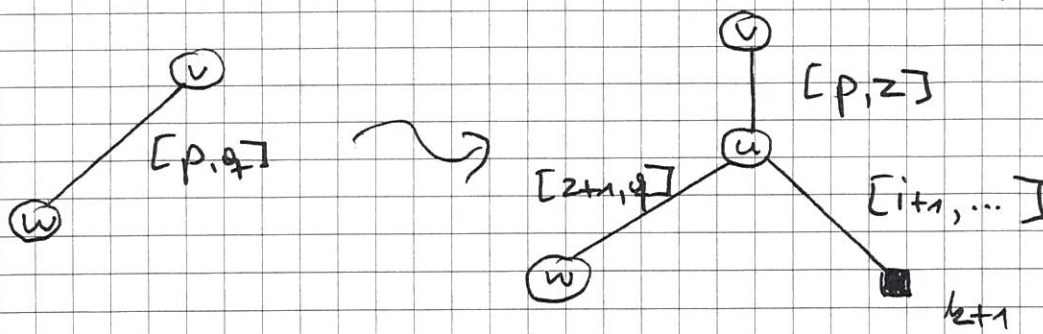
$\Rightarrow$

Extension step $i+1$ is the last such a step, which can be performed in constant time.

2.2 P ends on an edge $e = (v, w)$.

Then v is an inner node. As shown above, scv exists in the current suffix tree.

Situation and modification:



Let

$$\alpha_{z+1} \times \beta = \alpha_{z+1} \alpha_{z+2} \dots \alpha_i,$$

where $\alpha_{z+1} \times$ is the marking of the path $\bar{p}$ from the root to the node v and β is the marking of the edge (v, u) .

$\Rightarrow$

$$|\beta| = z - p + 1.$$

Assumption $\Rightarrow$

The node v contains a pointer to scv .

- We follow the pointer to scv and then from scv the path with marking β until the end of this path is found.

Analysis:

Let

$$\beta = \alpha_e \alpha_{e+1} \dots \alpha_i.$$

Always when an inner node is inserted, this new inner node obtains a new leaf as son.

During the whole construction, this leaf remains to be a leaf in the current suffix tree. The number of the leaf is never changed.

By construction, scv has at most one outgoing edge with marking $[a_e, \dots]$.

Since P ends on the edge (v, w) , scv has an outgoing edge

$$e_1 := (scv, w_1)$$

with marking $[a_{e_1}, \dots]$

Let $[p_1, q_1]$ be the marking of the edge e_1 .

In the subsequence, L denotes the length of the suffix of β which has not been considered.

At the beginning there holds

$$L = z - p + 1.$$

If $L = q_1 - p_1 + 1$ then the path with marking β ends in the node w_1 .

If $L < q_1 - p_1 + 1$ then this path ends on the edge e_1 .

If $L > q_1 - p_1 + 1$ then the marking of the edge e_1 is a proper prefix of length $q_1 - p_1 + 1$ of β . We modify L by

$$L := L - (q_1 - p_1 + 1).$$

1) and follow the rest of the path starting in w_1 .

Knowing the end of the path which starts in the root and has markings $a_{z+2} a_{z+3} \dots a_i$, we proceed analogously to the treatment of the path with marking $a_{z+1} a_{z+2} \dots a_i$.

Two things remain to be explained. We have to develop a method for the construction of the suffix pointers for the inner nodes. We have to estimate the time needed to follow the paths.

• Computation of the suffix pointers

The following lemma shows that after the insertion of a new inner node u during an extension step of a phase, the node $sc(u)$ will be visited during the next extension step of the same phase. At the moment when $sc(u)$ is visited during the next extension step, a pointer to $sc(u)$ could be attached to u in constant time. By construction, the pointer to $sc(u)$ is needed for the first time after its attachment to u . Note that

$$m(u) = a_j a_{j+1} \dots a_i$$

if u is added during the construction of T_{i+1} during the extension step j .

Lemma 2.2

- 1) Assume that a new inner node u is added as son of the node v to the current suffix tree during the construction of T_{i+1} during the extension step j . Then during the extension step $j+1$, the path starting in $s(v)$ which has the same marking β as the edge (v, u) either
- ends in an inner node w with $m(w) = a_{j+1} \dots a_i$
- or
- adds to the current suffix tree a new node w with $w = a_{j+1} a_{j+2} \dots a_i$.

Proof:

exercise



It remains the analysis of the total time used for following the suffix pointer and the subpaths in the tree. If we can show that the total time is $O(n)$ then this implies that the implicit suffix tree T_n can be constructed in $O(n)$ time.

Let u be a node in a tree T . The depth $D(u, T)$ of u with respect to T is defined by

$$D(u, T) := \begin{cases} 0 & \text{if } u = \text{root}(T) \\ 1 + D(u, T_i) & \text{otherwise,} \end{cases}$$

where T_i is that direct subtree of T which contains u .

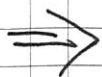
Note that $D(u, T)$ is the number of nodes on the path from $\text{root}(T)$ to u where the node u itself is not counted.

Idea:

We consider the algorithm as an agent which starts in the root of the initial suffix tree and follows the paths as done by the algorithm running from node to node. We wish to estimate the total number of nodes visited by the agent.

Observation:

- 1) The depth of the start node is zero.
- 2) Running over a tree edge increases the depth of the current node of the agent by one.
- 3) The depth of the current node of the agent can only be decreased by running over a suffix pointer.
- 4) After the termination of the algorithm, the depth of the current node of the agent is at most $n-1$.



$$TN \leq 2n + td$$

where td denotes the total decrease of the

depth of the current node of the agent.

It remains to get an estimate of td .

Lemma 2.3:

Let (v, scv) be a suffix pointer used by the agent. Then at the moment when the agent uses (v, scv) in the current suffix tree T there holds

$$D(v, T) \leq D(scv, T) + 1.$$

Proof:

Let

P be the path from $\text{root}(T)$ to v in T
and

P' be the path from $\text{root}(T)$ to scv in T .

Let w be any predecessor of v on P with

$$m(w) = a\beta, \quad a \in \Sigma, \beta \in \Sigma^+$$

Lemma 2.2 $\Rightarrow$ scw exists

By construction, scw is a node on P' .

Let w and w' be two distinct predecessors of v on P with $|m(w)| \geq 2$ and $|m(w')| \geq 2$

$$\Rightarrow scw \neq scw'.$$

Only the direct Successor of $\text{root}(T)$ on P eventually has not a suffix pointer

Altogether, we have shown

(8)

$$|P| \leq |P'| + 1$$

$\Rightarrow$

$$D(v, T) \leq D(scv, T) + 1$$

■

Each suffix pointer is used by the agent at most once.

Exercise

Show that each suffix pointer is used by the agent at most once.

Lemma 2.3 $\Rightarrow$

$$td \leq n$$

Altogether, we have proved the following theorem.

Theorem 2.2

The length of the path followed by the agent during the construction of the implicit suffix tree is at most $3n$.

In each of the n phases, the other work uses only constant time. Hence, the construction of the implicit suffix tree can be performed in $O(n)$ time.

It remains to explain how to construct a suffix tree using the implicit suffix tree.

We add at the end of the text string x the new symbol $\$ \notin \Sigma$. Note that no suffix of $x\$$ is prefix of another suffix. Therefore, the implicit suffix tree for $x\$$ is also the suffix tree for $x\$$.

Exercise

Develop an algorithm which obtains a suffix tree for $x\$$ and constructs a suffix tree for x in linear time.

Altogether, we have proved the following theorem

Theorem 2.3

Let Σ be a finite alphabet and let $x \in \Sigma^+$ be a string of length n . A suffix tree for x can be constructed in $O(n)$ time.

2.2.3 Applications of suffix trees

Note that each substring of a text string is a prefix of a suffix of the text string. Hence, suffix trees can be used for the implementation of an index for a text.

Let $x := a_1 a_2 \dots a_n$ be a text string; e.g. a book.

An identifier of the position i , $1 \leq i \leq n$ is the shortest prefix of $a_i a_{i+1} \dots a_n$ which occurs nowhere else in x . If x ends with a special symbol then the identifier is defined for each position in x . A position tree of a text string $x = a_1 a_2 \dots a_n \in \Sigma^n$ is a compact trie with respect to the alphabet Σ which contains $\leq n$ marked leaves. The leaves are numbered with pairwise distinct numbers from $\{1, 2, \dots, n\}$. The path from the root to the leaf with number i corresponds to the identifier of the position i in the text string x .

Exercise

Develop an algorithm which obtains a suffix tree for x as input and constructs a position tree for x in linear time.

The index of a text string x should support the following four operations for a given string $y \in \Sigma^+$

- 1) Find the longest prefix of y contained in x .
- 2) Find the first occurrence of y in x .
- 3) Determine the number of occurrences of y in x .
- 4) Compute a list of all occurrences of y in x .

Exercise:

Develop algorithms for the four operations which should be supported by the index of a text string x . Show that the suffix tree for a text string x can be prepared in linear time such that the number of occurrences of any string $y \in \Sigma^+$ in x can be determined in $O(|y|)$ time.

In 1970, Don Knuth has conjectured that the following problem cannot be solved in linear time.

Longest common substring problem

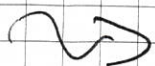
Input: Strings $x_1, x_2 \in \Sigma^+$ where Σ is a finite alphabet

Output: A longest common substring of x_1 and x_2

Using suffix trees this problem can be solved in linear time.

Observation:

A longest common suffix of x_1 and of x_2 is prefix of a suffix of x_1 and also prefix of a suffix of x_2 .



Idea:

Construct a common suffix tree for the strings x_1 and x_2 .

Goal:

Construction of a common suffix tree for a set $\{x_1, x_2, \dots, x_t\}$ of strings.

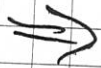
Let $\$1, \$2, \dots, \$t$ be pairwise distinct special symbols not occurring in any of the strings $x_1, \dots, x_t$.

- Construct a suffix tree for the string

$$x_1 \$1 x_2 \$2 \dots \$t x_t \$t.$$

Properties:

The choice of the special symbols $\$1, \$2, \dots, \$t$ and construction algorithm



The constructed tree has the needed structure. Since the markings of the leaves are not extended explicitly, we consider these to end with the corresponding special symbol $\$i$.

We get a solution of the longest common substring problem by the construction of a common suffix tree for both input strings x_1 and x_2 . During the construction, we store for each inner node v the number of symbols on the path from the root to v .

Then we traverse the suffix tree bottom up and decide for each inner node if its subtree contains a leaf with respect to both strings x_1 and x_2 . Such a node with maximal number of symbols on the path from the root to the node v defines a longest common substring of x_1 and x_2 .

Exercise:

- a) Work out the algorithm for the solution of the longest common substring problem.
- b) Generalize the algorithm such that a longest common substring of k strings $x_1, x_2, \dots, x_k$ can be computed in $O(k \cdot n)$ time where $n := |x_1| + |x_2| + \dots + |x_k|$.

Remark:

A multitude of further applications of suffix trees can be found in

Dan Gusfield, Algorithms on Strings, Trees, and Sequences, Cambridge University Press, 1997, Chapters 7 and 9.