6 Suffix Trees (Knut Reinert / Clemens Gröpl)

This exposition is based on:

1. Dan Gusfield: Algorithms on Strings, Trees, and Sequences. Computer Science and Computational Biology. Cambridge University Press, Cambridge, 1997, pages 94ff. ISBN 0-521-58519-8
2. An earlier version of this lecture by Knut Reinert.

We will present the online suffix tree construction proposed by Ukkonen.

6.1 Introduction

Exact string matching is used in many algorithms in Computational Biology as a first step:

Given a pattern $P = P[1 .. m]$, find a all occurrences of P in a text $S = S[1 .. n]$.

This can readily be done with string matching algorithms in time $O(m + n)$. If however, the text is very long, we would prefer not to scan it completely for every query, but rather spend time $O(m)$.

To do that we have to preprocess the text, such that $O(m)$ queries are possible. In this lectures we introduce such a preprocessing, namely the construction of a *suffix tree*, which is a central data structure in computational molecular biology.

Suffix trees were introduced 1973 by Weiner and three years later optimized by McCreight. The analysis is a bit involved and it was not until 1995, before Esko Ukkonen proposed an alternative algorithm with the same space and time complexity which has also the advantage of being online. A comparison of the three methods can be found in a paper by Giegerich et. al.

6.2 Definition

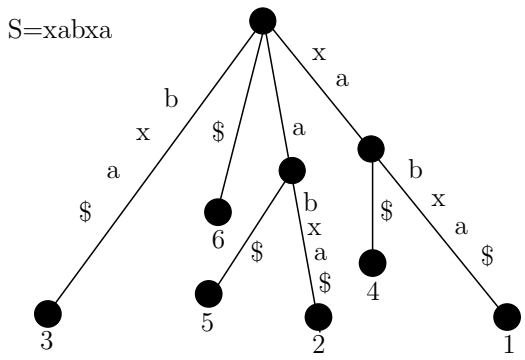
Definition 1 *Let $S = S[1 .. n]$ be a string of length n over a fixed alphabet Σ. A* suffix tree *for S is a tree with n leaves and the following properties:*

1. *Every internal node other than the root has at least 2 children.*
2. *Every edge is labeled with an nonempty substring of S.*
3. *The edges leaving a given node have labels starting with different letters.*
4. *The concatenation of the labels of the path from the root to leaf i spells out the i-th suffix $S[i .. n]$ of S. We denote $S[i .. n]$ by S_i.*

6.3 Marking the end of S

Note that according to the above definition there is no suffix tree if a suffix of S is a prefix of another suffix of S. For example for $S = xabxa$ there is no leaf for the fourth suffix S_4.

This problem is easily overcome by adding a special letter \$ to the alphabet which does not occur in S, and putting it to the end of S. This way no suffix can be a prefix of another suffix.



The above figure shows a suffix tree for the string $xabxa\$$.

6.4 Storing the edge labels efficiently

What about the space consumption? The total length of all edge labels in a suffix tree can easily be $\Omega(n^2)$, e.g. for $S = abc \cdots xyz$.

Therefore we do not store the substrings $S[i \,..\, j]$ of S in the edges, but only their start and end indices (i, j). Nevertheless we keep thinking of the edge labels as substrings of S.

Definition 2

- *The* label of a path *from the root to an inner node v is the concatenation of the labels of all the edges on the path.*
- *The* label of a node *is the label of the path from the root to the node.*
- *The* node depth of a node *is the number of nodes from the root to the node.*
- *The* string depth of a node *is the number of characters in its label.*

6.5 A naive algorithm for suffix tree construction

The *naive algorithm* for constructing a suffix tree is as follows:

We insert the suffixes $S_1, S_2, \ldots, S_n$ (in this order) and modify the tree according to the definition.

1. Say we want to insert S_i into the current tree. We read the letters of S_i and walk down the path from the root accordingly, until a mismatch occurs. At this point we have to branch off a new edge.
2. If the mismatch occurs in the midst of an edge, then we split it into two edges and branch off from the inserted vertex.
3. Otherwise we can branch off from a vertex which is already present.

We need time $O(n - i)$ for the i-th suffix and hence the total running time is $\sum_1^n O(i) = O(n^2)$.

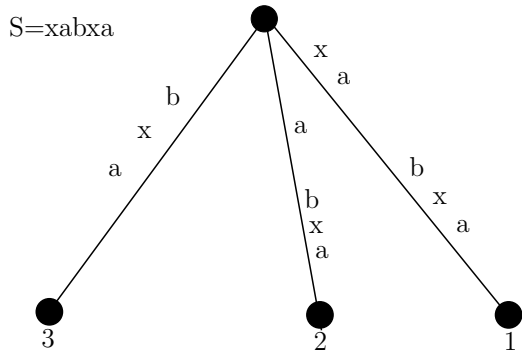
6.6 Implicit suffix trees

A faster construction will use the notion of an *implicit* suffix tree.

Definition 3 *A* implicit *suffix tree for S is constructed from a suffix tree T for $S\$$ by the following steps:*

1. *Delete every occurrence of the end symbol* \$ *from* T.
2. *Delete all edges without labels.*
3. *Remove all nodes of degree* 1.

The implicit suffix tree for the i-th *prefix* $S[1 \,..\, i]$ is denoted by T_i.



The above figure shows an implicit suffix tree for the string $xabxa$.

6.7 Ukkonen's algorithm

Contrary to the naive algorithm, Ukkonen's algorithm begins with an implicit suffix tree T_1 for $S[1]$.

The algorithm then proceeds in *phases* $i = 2, \ldots, n$. In phase $i+1$, the tree T_{i+1} for $S[1 \,..\, i+1]$ is constructed from the tree T_i for $S[1 \,..\, i]$.

Each phase i consists of several *extension steps* $j = 1, \ldots, i$. After extension step j of phase i, the string $S[j \,..\, i]$ is a prefix of some path label in the tree.

Basically, we will obtain a suffix tree for $S\$$ from an implicit suffix tree for $S\$$ and some final "cleanup". (So far they are identical, but we will add some extra features to the data structure.)

ConstructSuffixTree(S[1 .. n])

Construct T_1;
for $i = 1;\ i < n;\ i++$ do
    /* begin of phase $i + 1$ */
    for $j = 1;\ j \leq i + 1;\ j++$ do
        /* begin of j-th extension */
        Find the end of the path with label $S[j \,..\, i]$;
        If needed, extend the path with $S[i + 1]$;
    od
    /* T_{i+1} has been constructed */
od

An invariant of the algorithm is that after phase $i + 1$ *all* suffixes of $S[1 \,..\, i + 1]$ are contained in the implicit suffix tree T_{i+1}.

The extension j in phase $i + 1$ follows three rules. Let $\beta := S[j \,..\, i]$, i.e. $\beta S[i + 1]$ has to be inserted.

Then the following three cases apply:

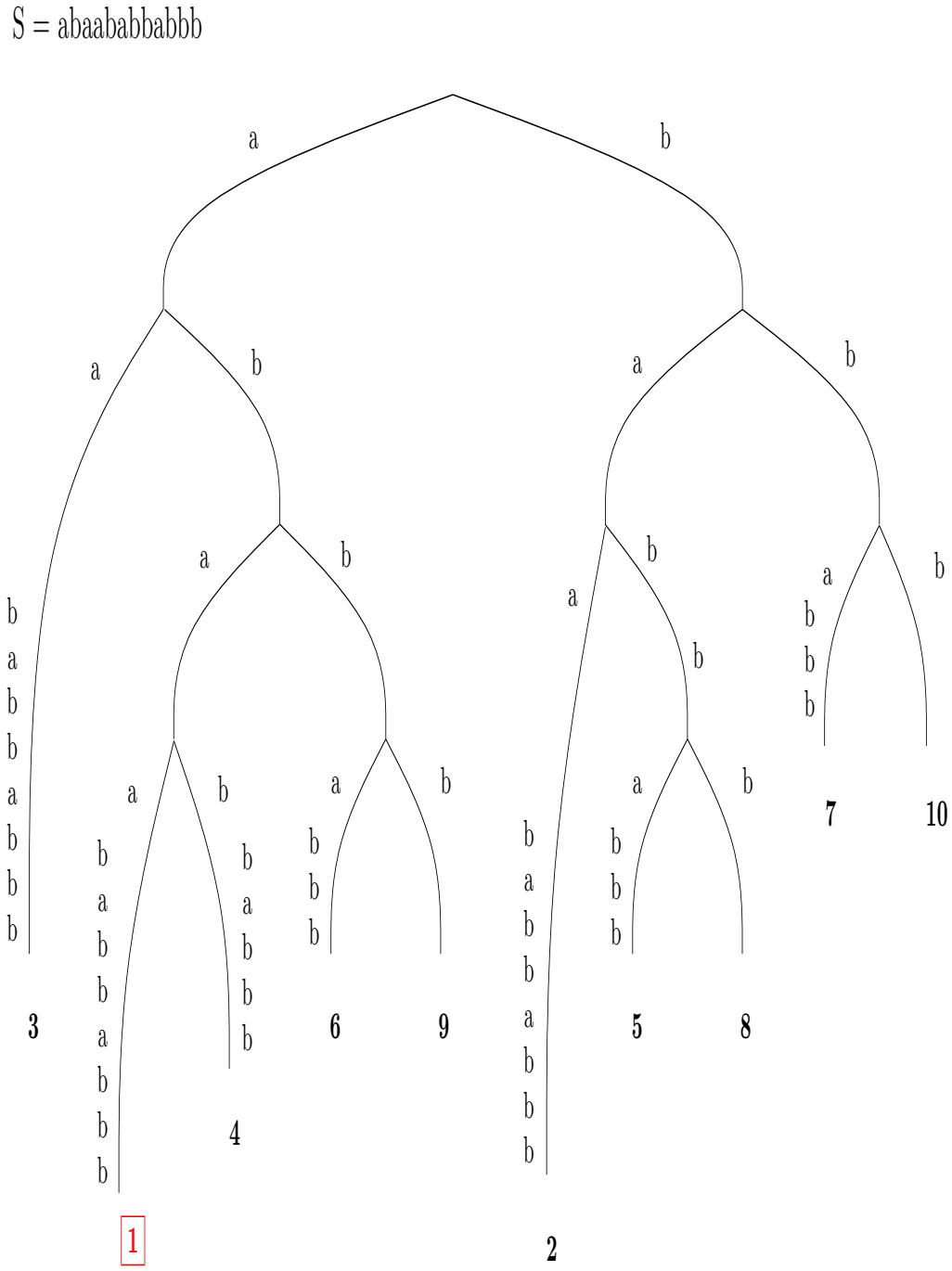
1. *The path for* β *ends in a leaf.*
   $\Rightarrow$ Then we append $S[i + 1]$ to the last edge label.

2. *No path from the end of β starts with $S[i+1]$, but at least one other path continues from the end of β.*
   $\Rightarrow$ Then we add a new edge with label $S[i+1]$ at the end of β. In case that β ends in the middle of an edge, a new node has to be created in order to do this. The new leaf node is given the number j.

3. *There is a path at the end of β, which starts with $S[i+1]$.*
   $\Rightarrow$ Then $\beta S[i+1]$ is already in the tree and we do nothing.

6.8 Example

Here is an implicit suffix tree for the string $abaababbabbb$.



6.9 Analysis

A naive implementation of Ukkonen's algorithm needs time $O(i+1-j)$ for extension j in phase $i+1$, i.e. for all extensions time $O(i^2)$ and hence for all n phases time $O(n^3)$. This is rather stupid, since we already know an algorithm to construct a tree in time $O(n^2)$.

We show how to improve the naive implementation of Ukkonen's algorithm in order to bring the build time down to $O(n)$. The first improvement is to add so called *suffix links* to the suffix tree data structure.

6.10 Suffix links

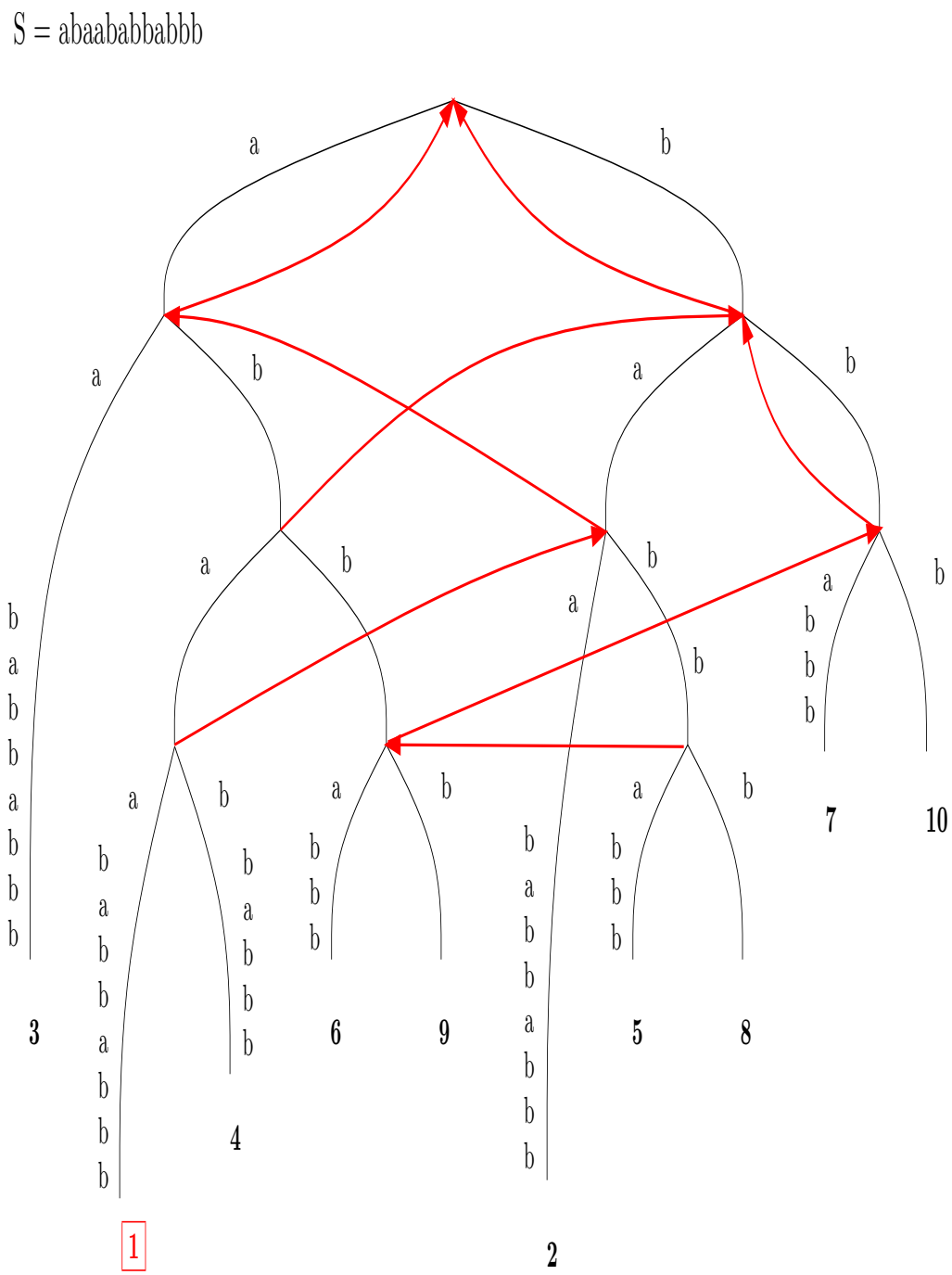
Definition 4 *Let $x\alpha$ be any string with $x \in \Sigma$ and $\alpha \in \Sigma^*$. If for an internal node v with label $x\alpha$ there is another node $s(v)$ with label α, then the pointer $v \to s(v)$ is called a* suffix link.

A suffix link can point to the root (which represents the empty string), but the root does not have a suffix link from it.

It turns out that all internal vertices of an implicit suffix tree have a suffix link leaving it.

6.11 Example

Here is an implicit suffix tree with suffix links for the string $abaababbabbb$.

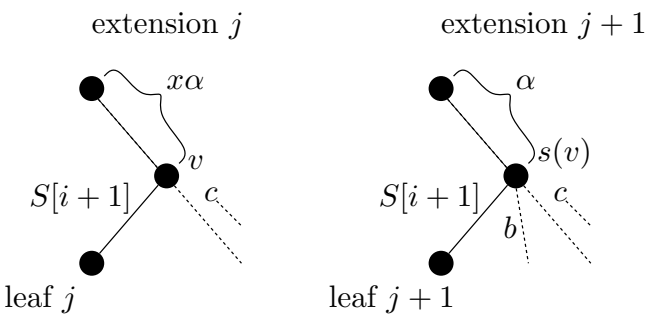


Lemma 5 *Assume that v is a node with label $x\alpha$ which was added in extension j of phase $i+1$, i.e. $x = S[j]$, $\alpha = S[j+1\,..\,i+1]$. Then one of the following cases is true for the next extension $j+1$, when α is to be added:*

1. *The path with label α already ends in an internal node.*
2. *At the end of string α, a new internal node is created.*

Thus after extension $j+1$ of phase $i+1$, there exists a node $s(v)$ with label α, and we have a suffix link $v \rightarrow s(v)$.

Proof: Since v is a new node, it was created by rule 2, i.e., we forked off a new leaf with number j, and the edge to the leaf has label $S[i+1]$. Moreover there is another edge leaving v, the label of which starts with say c.



Now in extension $j+1$ we search the prefix α in the tree, and we know that it will be followed by c. Then there are two cases:

1. α is followed *only* by c. Then rule 2 is applied again, creating a new leaf with number $j+1$ and a new internal node $s(v)$.
2. α is also followed by a letter different from c. Then there is already a node $s(v)$.

In both cases the lemma is true. ■

Corollary 6 *In Ukkonen's algorithm every newly created, internal node has an outgoing suffix link at the end of the next extension.*

Proof: Exercise.

Or put differently:

Corollary 7 *In any implicit suffix tree T_i, if an internal node has label $x\alpha$, then there is in T_i a node $s(v)$ with label α.*

6.12 Using suffix links

How do we make use of the suffix links during the algorithm?

1. The first extension $j = 1$ of phase $i+1$ always follows rule 1. We can perform it in constant time if we can find the end of $S[1\,..\,i]$ in constant time. Thus we handle the case $j = 1$ separately before we enter the "extension loop" for $j = 2, \ldots, i+1$ and maintain a pointer to the end of $S[1\,..\,i]$ during the "phase loop" for $i = 1, \ldots, n-1$. [This is indicated by the box around 1 in the preceding figure.]
2. For $j \geq 1$ we can find $S[j+1\,..\,i]$ much faster using suffix links:
   - Let $\beta := S[j\,..\,i]$, $x = S[j]$ and $\alpha = S[j+1\,..\,i]$. Then $\beta = x\alpha$ and we want to find the end of α using the end of β and suffix links.
   - Let v be the first node *above* or *at* the end of β that has a suffix link or is the root. Let γ (possibly empty) be the substring of S which is read between both.
     By the preceding lemma, such a node v must exist and its node rank is at most one less than that of the end of β.
     (a) If v is the root we let $w := v$.
     (b) If v has a suffix link $v \to s(v)$ then we let $w := s(v)$.
   - Now the crucial observation is that *the end of α is in the subtree below w.*
     (a) If $w = v$ is the root, we search for α starting at the root as before.
     (b) But if $w = s(v)$, then we only have to follow the path with label γ starting from w in order to find the end of α. This is easy to see: Let δ be the path label of $s(v)$, then $\beta = x\delta\gamma$ and $\alpha = \delta\gamma$.
   - When have found the end of α, we apply the extension rules for the $(j+1)$-th extension.
   - Finally, if v was created in extension j, then after extension $j+1$ there is a vertex $s(v)$ at the end of α, see the preceding lemma, and we can create a suffix link $v \to s(v)$ accordingly.

6.13 Running time

Can we show that the algorithm will take just a constant number of steps for each extension? Not quite! But we can bound the number of steps during an entire phase.

The algorithm never goes up more than one node in extension j when it searches for a suffix link, according to Corollary 6.

We need one more observation:

Lemma 8 *Let $v \to s(v)$ be a suffix link which is traversed during Ukkonen's algorithm. Then the node depth of v is at most one higher than the node depth of $s(v)$.*

Proof: Exercise.

Theorem 9 *Each phase in Ukkonen's algorithm needs time $O(n)$.*

Proof: In each of the $i + 1 \leq n$ extensions in phase $i + 1$, the algorithm goes at most one edge up, traverses at most one suffix link and then follows some edges downwards.

Looking at the node depth of the traversed nodes, it decreases by at most two. (Go to v, then to $s(v)$). Since the number of nodes is bounded by n, we can have at most $2n$ steps that decrease the node depth.

Then the node depth increases somewhat during the walk down. Since no node has a node depth $> n$ and since (summed over all extensions) the node depth can decrease by at most $2n$, the node depth can increase overall by at most $3n$.

Hence at most $5n$ nodes can be visited. Since each visit costs constant time, the theorem follows. ■

We are now down to a running time of $O(n^2)$.

6.14 Two observations

We need only two more observations to conclude that the algorithm takes time $O(n)$.

Lemma 10 *As soon as in phase $i + 1$ rule 3 is applied during an extension j, it will also apply for extensions $j + 1, \ldots, i + 1$ during that phase.*

Proof: Rule 3 applies in extension j iff $S[j \,..\, i + 1]$ is a prefix of some suffix of $S[1 \,..\, i]$. But then the same is true for extensions $j + 1, \ldots, i$. ■

Hence we can stop a phase once rule 3 has been applied for the first time, say in extension j^*.

Later extensions would not alter the tree. In particular, there is no need to create new suffix links.

We say that the extensions from $j^* + 1$ on are *implicit*, as opposed to the *explicit* extensions performed before.

The second observation concerns rule 1.

Lemma 11 *A leaf stays a leaf. It is always extended according to rule 1.*

Proof: Rule 2 can only generate a new branch somewhere in the midst of an edge. ■

6.15 Implicit extensions

At the beginning of any phase i we have a number of extensions according to rules 1 and 2, followed by implicit extensions.

Let j_i be the last explicit extension in phase i. Since rule 2 generates leaves, in phase $i+1$ rule 1 applies for all extensions up to j_i. This implies $j_i \leq j_{i+1}$.

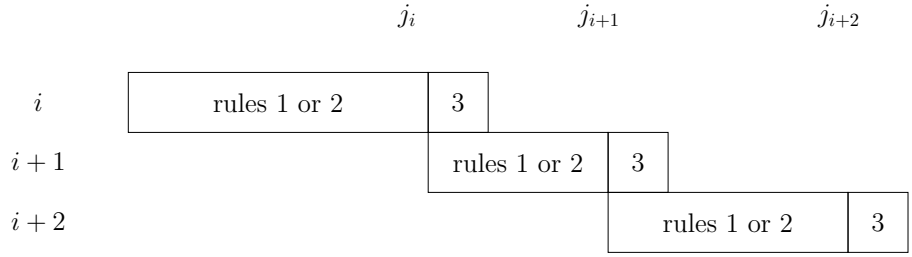
Recall that the edges are actually labeled with the start and end positions of the corresponding substrings in S.

We can perform the extensions according to rule 1 implicitly and all at once if we label the leaf edges using a "global end index" e. That is, leaf edges are labeled (p, e) instead of using $(p, i+1)$. Then we only need to increase the value of e from i to $i+1$.

We can summarize the procedure in phase $i+1$ as follows:

1. Set e to $i+1$, thus performing extensions $1, \ldots, j_i$ in one step.
2. Perform extensions $j_i + 1, j_i + 2, \ldots$ explicitly (using suffix links) until in some extension j^* rule 3 is applied, or $j^* = i + 2$.
3. Set $j_{i+1} := j^* - 1$ and proceed to the next phase. (Extensions $j^* + 1, \ldots, i + 1$ are implicit.)

It follows that two successive phases of the algorithm have at most one extension index in common for which we have to compute explicit extensions.



Concluding we have have the following theorem:

Theorem 12 *Ukkonens algorithm constructs the implicit suffix tree T_n in time $O(n)$.*

Proof: The time for all implicit extensions is constant in each phase and hence at most $O(n)$ for the n phases. Since the indices j_i are nondecreasing and any two phases share at most one position where an explicit extension is due, there are at most $2n$ explicit extensions in total.

Moreover, as phase $i+1$ begins where phase i ended, we can store the position of $S[j_i \,..\, i]$ in between phases. The node depth does not change and at most $O(n)$ nodes are traversed, by the same argument as for Theorem 9.

6.16 Example

Here is a table for the calculation of the implicit suffix tree for the string $abaababbabbb$ we have seen before. The numbers indicate the rules. Only explicit extensions are shown.

a 2	b	a	a	b	a	b	b	a	b	b	b
	b 2	a	a	b	a	b	b	a	b	b	b
		a 3	**a 2**	b	a	b	b	a	b	b	b
			a 3	**b 3**	**a 3**	**b 2**	b	a	b	b	b
				b	a	**b 2**	**b 2**	a	b	b	b
					a	**b 3**	**b 2**	a	b	b	b
						b	**b 2**	**a 3**	**b 3**	**b 3**	**b 2**
							b 3	a	b	b	**b 2**
									b	b	**b 2**
										b	**b 3**
											b

6.17 Final clean-up

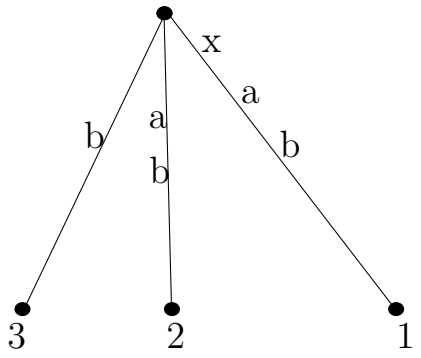
The final implicit suffix tree T_n can be converted into a suffix tree for $S\$$ in $O(n)$ time:

- Add a terminal symbol \$ in one more "phase".
- Replace the global end index e with $n+1$ in an $O(n)$ traversal of the tree.
- The result is a suffix tree for $S\$$.

6.18 Example

In the following we show another example for the string $xabxacbxd$.

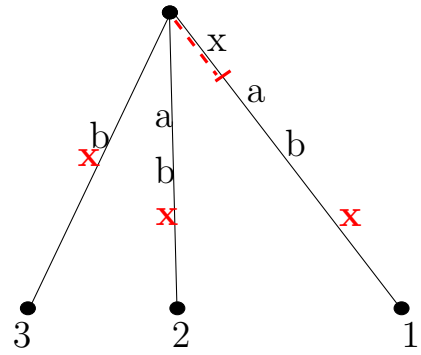
$\mathcal{T}_3=$



phase 4: $S[1..4] = \text{xabx}$

E	$S[j..i]$	$S[i+1]$	R
1	xab	x	1
2	ab	x	1
3	b	x	1
4		x	3

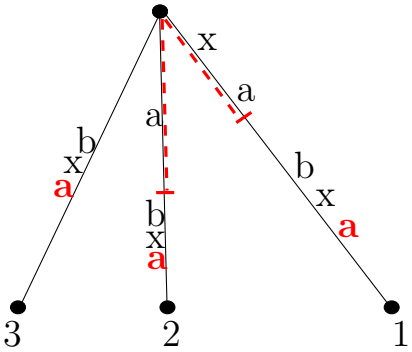
$\mathcal{T}_4=$



phase 5: $S[1..5] = \text{xabxa}$

E	$S[j..i]$	$S[i+1]$	R
1	xabx	a	1
2	abx	a	1
3	bx	a	1
4	xa	a	3
5		a	3

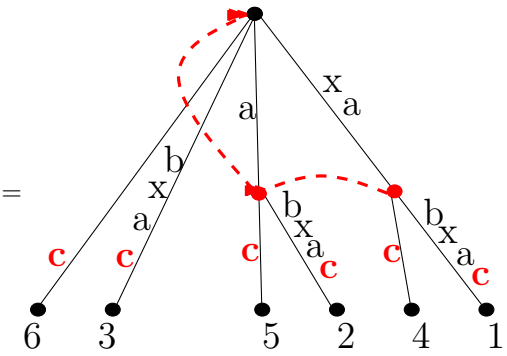
$\mathcal{T}_5=$



phase 6: $S[1..6] = \text{xabxac}$

E	$S[j..i]$	$S[i+1]$	R
1	xabxa	c	1
2	abxa	c	1
3	bxa	c	1
4	xa	c	2
5	a	c	2
6		c	2

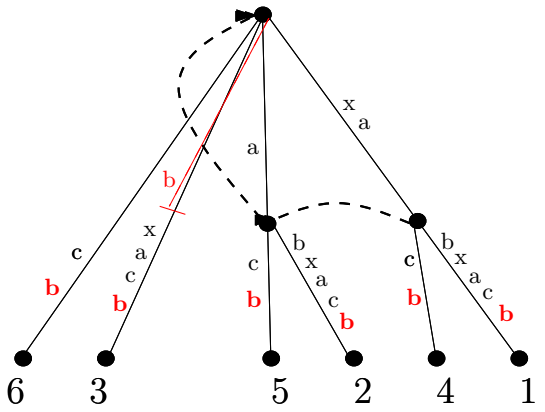
$\mathcal{T}_6=$



phase 7: $S[1..7] = \text{xabxacb}$

E	$S[j..i]$	$S[i+1]$	R
1	xabxac	b	1
2	abxac	b	1
3	bxac	b	1
4	xac	b	1
5	ac	b	1
6	c	b	1
7		b	3

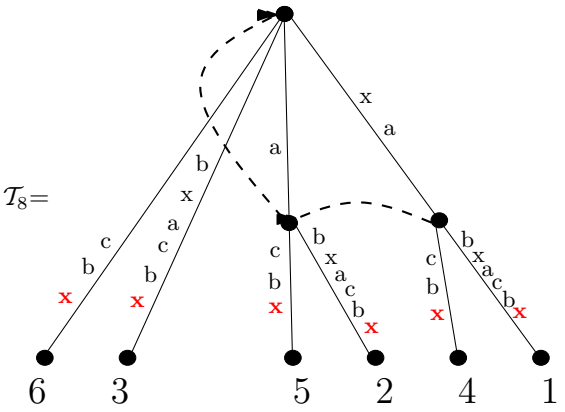
$\mathcal{T}_7=$



phase 8: $S[1..8] = \text{xabxacbx}$

E	$S[j..i]$	$S[i+1]$	R
1	xabxacb	x	1
2	abxacb	x	1
3	bxacb	x	1
4	xacb	x	1
5	acb	x	1
6	cb	x	1
7	b	x	3
8		x	3

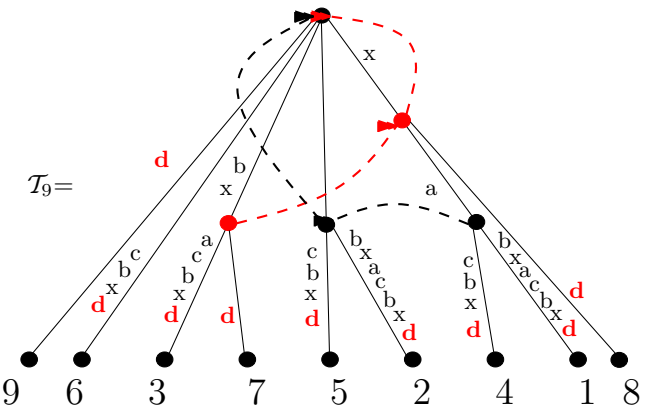
$\mathcal{T}_8=$



phase 9: $S[1..9] = \text{xabxacbxd}$

E	$S[j..i]$	$S[i+1]$	R
1	xabxacbx	d	1
2	abxacbx	d	1
3	bxacbx	d	1
4	xacbx	d	1
5	acbx	d	1
6	cbx	d	1
7	bx	d	2
8	x	d	2
9		d	2

$\mathcal{T}_9=$



6.19 Applications of suffix trees

In the following we discuss how suffix trees can be used to efficiently solve common problems in sequence analysis.

6.20 Longest common substring of two strings

Assume we are given two strings $S_1[1 \,..\, n_1]$ and $S_2[1 \,..\, n_2]$. Then we would like to know i, j and k such that $S_1[i \,..\, i+k-1] = S_2[j \,..\, j+k-1]$ and k is as large as possible. This ist the *longest common substring* problem (for two strings).

Here is the outline of a solution using suffix trees:

1. Build a suffix tree T for $S := S_1\$_1S_2\$_2$ in $O(n_1 + n_2)$ time using Ukkonen's algorithm. Here $\$_1$ and $\$_2$ are different new symbols not occuring in S_1 and S_2.
2. Mark every internal node of T with $\{1\}$, $\{2\}$, or $\{1, 2\}$, depending on whether its path label is a substring of S_1 and/or S_2.
3. Find a node v which is labeled by both 1 and 2 and has the largest string depth among all those.
4. Output the path label of v and where it occurs in S_1 and S_2.

Now we go into more details:

2. Mark every internal node of T with $\{1\}$, $\{2\}$, or $\{1, 2\}$, depending on whether its path label is a substring of S_1 and/or S_2.

This can be done by a traversal of T. We know from the label of a leaf whether its suffix starts in S_1 or S_2. As we return from the leaves, we can record at the internal nodes whether there is a leaf for a suffix starting in S_1 and/or S_2 in the subtree below.

3. Find a node v which is labeled $\{1, 2\}$ and has the largest string depth among all those.

Clearly we can compute the string depths of all internal nodes by a tree traversal and keep track of the maximum among all nodes labeled $\{1, 2\}$.

4. Output the path label of v and where it occurs in S_1 and S_2.

The path itself is straightforward. By traversing the subtree below v, we can also output all starting positions of the longest common substring in S_1 and S_2.

Note: The generalization to k strings is straightforward. We can even search for the longest substring occuring in at least ℓ of the k strings. (Idea: use $S_1\$_1S_2\$_2 \ldots S_k\$_k$ and mark the internal vertices with subsets of $\{1, 2, \ldots, k\}$.)

6.21 Maximal repeats

Assume we are given a string $S[1 \,..\, n]$. A *repeat* is a substring of S that occurs at different starting positions. That is, we have $S[i \,..\, i + k - 1] = S[j \,..\, j + k - 1]$, $i \neq j$, $k > 0$. Such a repeat (i, j, k) is *left maximal* iff $S[i-1] \neq S[j-1]$ and *right maximal* iff $S[i+k] \neq S[j+k]$. It is *maximal* if it is both left and right maximal.

Maximal repeats can be found using suffix trees. Let T be a suffix tree for $S\$$.

Observation 13 *Every repeat is right maximal iff it is a path label of some internal node of T.*

Proof: Let (i, j, k) be a right maximal repeat. Then $S[i \,..\, n]$ and $S[j \,..\, n]$ are suffixes of S such that the first k positions are identical, but $S[i+k] \neq S[j+k]$. Hence T has an internal vertex with path label $S[i \,..\, i+k-1] = S[j \,..\, j + k - 1]$. This vertex has two outgoing edges whose labels begin with $S[i + k]$ and $S[j + k]$.

The other direction is obvious. ■

Corollary 14 *A string of length n can have at most n maximal repeats.*

Definition 15

- *Let* $2 \leq i \leq n$. *Then* $S[i-1]$ *is called the* left character *of position* i.
- *The* left character *of a leaf of* T *is the left character of its label (i. e., of its starting position in* S*).*
- *An node* v *of* T *is called* left diverse *if there are at least two leaves below* v *which have different left characters.*

Lemma 16 *Let* α *be the path label of an internal vertex* v *of* T. *Then* α *is a maximal repeat if and only if* v *is left diverse.*

Proof: First assume that v is left diverse. Then there are substrings $x\alpha$ and $y\alpha$, $x \neq y$, of S. Let p and q be the letters following these substrings, so that $x\alpha p$ and $y\alpha q$ are substrings of S. If $p \neq q$ then α is a maximal repeat. If $p = q$ then there must be another suffix beginning with αr, $r \neq p$, because v is a branching node. Its left character is either x or y, but not both. So again α is a maximal repeat.

The other direction is obvious. ■

Thus we only need to determine the set of left diverse internal nodes. Observe that if an internal node is left diverse, then all its predecessors are left diverse, too. The set T' of left diverse internal nodes represents the maximal repeats of S in linear space. We can compute T' in $O(n)$ time by a simple depth first search traversal of the tree as follows:

1. Every leaf is labeled by its left character. The leaf labeled 1 receives a special, unique marker, e.g. 'ε'.
2. As the DFS returns from the leaves, when it visits an internal node v for the last time, it encounters one of these two cases:
   (a) All leaves below v have the same left character x.
   $\Rightarrow$ Then v is not left diverse. The character x is propagated to the predecessor of v.
   (b) There are leaves below v with different left characters.
   $\Rightarrow$ Then v is left diverse, and so are all its predecessors.

6.22 Maximal unique matches, and the MUMmer tool

A *maximal unique match* (MUM) betwen two sequences $A[1 \,..\, m]$ and $B[1 \,..\, n]$ is a maximal substring of A and B that does not occur at other places in A and B. Thus a MUM is given by (i, j, k) such that:

- $A[i \,..\, i+k-1] = B[j \,..\, j+k-1]$
- $A[i-1] \neq B[j-1]$
- $A[i+k] \neq B[j+k]$
- $\forall\, p \neq i: \quad A[p \,..\, p+k-1] \neq A[i \,..\, i+k-1]$
- $\forall\, q \neq j: \quad B[q \,..\, q+k-1] \neq B[j \,..\, j+k-1]$

Here are some examples:

This is a MUM:
```
ababababerndbababab
abcdcdabcdcberndcdc
```

This is not a MUM:
```
ababababerndabababab
abcdberndcdabcdcberndcdcdc
```
(not unique)

This is not a MUM:
```
abababawaberndernebelabababab
abcdcdabcdcwaberndernebelcdcdc
```
(not maximal)

The program MUMmer uses MUMs in much the same way as FastA does with diagonal runs. The basic steps are:

1. Find the MUMs of the two sequences for which an alignment shall be found.
2. Find a consistent (and optimal) subset of all MUMs, i.e., solve a “chaining problem”.
3. Try to close the remaining unaligned gaps, using a variety of methods – including DP alignment similar to the Smith-Waterman algorithm, and recursive calls of the MUMmer algorithm.
4. Output the final alignment.

The details of the MUMmer algorithm, and the improvements implemented in MUMmer2, can be found in the following papers:

- AL Delcher, S Kasif, RD Fleischmann, J Peterson, O White, SL Salzberg: *Alignment of whole genomes*. Nucleic Acids Research, 1999, Vol. 27, No. 11, p. 2269-2376.
- AL Delcher, A Phillippy, J Carlton, SL Salzberg: *Fast algorithms for large-scale genome alignment and comparison*. Nucleic Acids Research, 2002, Vol. 30, No. 11, p. 2478-2483.

Both papers are recommended reading (especially the algorithmic sections).

7 Suffix arrays (Knut Reinert)

This exposition is based on the following sources, which are all recommended reading:

1. Dan Gusfield: Algorithms in strings, trees and sequences, Cambridge, pages 94ff.
2. Udi Manber, Gene Myers: Suffix arrays: A new method for online string searching, Siam Journal on Computing 22:935-48,1993
3. Kasai, Lee, Arimura, Arikawa, Park: Linear-Time Longest-Common-Prefrix Computation in Suffix Arrays and Its Applications, CPM 2001
4. Abououelhoda, Kurtz, Ohlebusch: The enhanced suffix array and its application to genome analysis, WABI 2002, Rome, Italy

7.1 Introduction

Exact string matching is used in many algorithms in Computational Biology as a first step. Given a pattern $P = p_1, \ldots, p_m$, we want to find a all occurrences in a text $S = s_1, \ldots, s_n$. This can readily be done with string matching algorithms in time $O(m + n)$. If however, the text is very long, we would prefer not to scan it completely for every query, but rather spend time $O(m)$. To do that we have to preprocess the text, such that linear time queries are possible.

In this lectures we introduce such a preprocessing, namely the construction of a *suffix array*, which is, together with the suffix tree, a central data structure in computational molecular biology. Suffix arrays were introduced by Manber and Myers in 1993.

They are a more space efficient way to build an index structure than suffix trees. This is offset by a moderate increase in search time from $O(m)$ to $O(m+\log n)$. In practice this increase is counterbalanced by better cache behaviour.

7.2 Definition

Definition 17 *Given a text $S = s_1, \ldots, s_n$, a* suffix array *for S, called* `suftab`*, is an array of integers of range 0 to n specifying the lexicographic ordering of the $n + 1$ suffixes of the string S\$.*

7.3 Example

Assume we have the text $acaaacatat$\$, then the basic suffix array is:

0	aaacatat$
1	aacatat$
2	acaaacatat$
3	acatat$
4	atat$
5	at$
6	caaacatat$
7	catat$
8	tat$
9	t$
10	$

Obvously this basic form needs only $4n$ bytes. It can be constructed in time $O(n \cdot \log n)$ in this space, or, alternatively in linear time by first building a suffix tree in linear time, then transforming the suffix tree into a suffix array (exercise), and finally discarding the suffix tree. Of course, sufficient memory has to be available to construct the suffix tree.

We will discuss the original construction proposed by Manber and Myers which needs less memory.

(Exercise: How much would a simple quicksort cost?)

7.4 Construction

In order to construct the suffix array we have to cleverly sort the n suffixes $S_1, \ldots, S_n$. We proceed in $\lceil \log_2(n+1) \rceil$ stages. In the first stage the suffixes are put into buckets according to their first symbol.

Now, inductively, we partition the buckets at each stage further by sorting according to *twice* the number of symbols. Lets number the stages $1, 2, 4, 8, \ldots$ to indicate the number of affected symbols. Thus, in the h-th stage, the suffixes are sorted according to $\leq_h$ order, i.e. sorted lexicographically according to the first h letters ($=_h$ and $<_h$ are defined accordingly).

Store the result in the table $suftab$ and, in addition, store in another array Bh a boolean value that demarcates the partitioning of the suffix array into buckets. Each bucket initially holds the suffixes with the same first symbol.

If we look at the example $acbaacatat\$$, we have the following after stage 1 (we demarcate the h-buckets with a double line):

ind	Bh	suftab
0	1	0=acbaacatat$
1	0	3=aacatat$
2	0	4=acatat$
3	0	6=atat$
4	0	8=at$
5	1	2=baacatat$
6	1	1=cbaacatat$
7	0	5=catat$
8	1	7=tat$
9	0	9=t$
10	1	10=$

Why would it help us to sort in those stages?

Assume at the h-th stage the suffixes are partitioned into m_h buckets, each holding suffixes with the same h first symbols. Then these buckets are sorted according to $\leq_h$. We now want to sort the elements in each h-bucket to produce the $\leq_{2h}$ order.

To do this we can make use of the following observation:

Observation 18 *Let S_i and S_j be two suffixes belonging to the same bucket after the h-th step, that is $S_i =_h S_j$. We need to compare the next h symbols. But the next h symbols of S_i (S_j) are exactly the first h symbols of S_{i+h} (S_{j+h}). By assumption we already know the relative order of S_{i+h} and S_{j+h} according to $\leq_h$.*

The idea is now the following: Let S_i be the first suffix in the first bucket (i.e. $suftab[0] = i$), and consider S_{i-h}.

Since S_i starts with the smallest h-symbol string, S_{i-h} should be the first in its $2h$ bucket. Hence we move S_{i-h} to the beginning of its bucket and mark this fact.

The algorithm scans the suffixes as they appear in $\leq_h$ order and for each S_i moves S_{i-h} to the next available place in its h-bucket.

We maintain three integer arrays $suftab$, $sufinv$ and $count$, and two boolean arrays Bh and $B2h$, all with $n+1$ elements.

At the start of stage h, $suftab[i]$ contains the start position of the i-th smallest suffix (according to the first h symbols).

$sufinv[i]$ is the inverse of suftab, i.e. $sufinv[suftab[i]] = i$, and Bh is 1 if $suftab[i]$ contains the leftmost suffix of an h-bucket.

If we look at the example $acbaacatat$\$, we have the following:

ind	Bh	sufinv	suftab
0	1	0	0=acbaacatat$
1	0	6	3=aacatat$
2	0	5	4=acatat$
3	0	1	6=atat$
4	0	2	8=at$
5	1	7	2=baacatat$
6	1	3	1=cbaacatat$
7	0	8	5=catat$
8	1	4	7=tat$
9	0	9	9=t$
10	1	10	10=$

In stage $2h$ we reset $sufinv[i]$ to point to the leftmost cell of the h-bucket containing the i-th suffix, rather than to the suffix's precise place in the bucket. The help array count is initialized to 0 for all i. In our example we have for stage 1:

ind	Bh	sufinv	suftab
0	1	0	0=acbaacatat$
1	0	6	3=aacatat$
2	0	5	4=acatat$
3	0	0	6=atat$
4	0	0	8=at$
5	1	6	2=baacatat$
6	1	0	1=cbaacatat$
7	0	8	5=catat$
8	1	0	7=tat$
9	0	8	9=t$
10	1	10	10=$

Now scan $suftab$ in increasing order, one bucket at a time. Let l and r mark the left and right boundary of the h-bucket currently being scanned.

Let T_i denote $suftab[i] - h$ (if T_i is negative we do nothing).

Then for every $l \leq i \leq r$, we

1. increment $count[sufinv[T_i]]$
2. set $sufinv[T_i] = sufinv[T_i] + count[sufinv[T_i]] - 1$
3. mark this by setting $B2h[sufinv[T_i]]$ to 1.

We denote in the following the position we are at with a $*$ and show a field *new pos* for clarification, which is not used in the algorithm. This is the initialization:

ind	Bh	B2h	count	sufinv	new pos	suftab
* 0	1	0	0	0		0=acbaacatat$
1	0	0	0	6		3=aacatat$
2	0	0	0	5		4=acatat$
3	0	0	0	0		6=atat$
4	0	0	0	0		8=at$
5	1	0	0	6		2=baacatat$
6	1	0	0	0		1=cbaacatat$
7	0	0	0	8		5=catat$
8	1	0	0	0		7=tat$
9	0	0	0	8		9=t$
10	1	0	0	10		10=$

Nothing happens since $0 - 1 < 0$.

ind	Bh	B2h	count	sufinv	new pos	suftab
0	1	0	0	0		0=acbaacatat$
* 1	0	0	0	6		3=aacatat$
2	0	0	0	**5**		4=acatat$
3	0	0	0	0		6=atat$
4	0	0	0	0		8=at$
5	1	1	1	6	5	2=baacatat$
6	1	0	0	0		1=cbaacatat$
7	0	0	0	8		5=catat$
8	1	0	0	0		7=tat$
9	0	0	0	8		9=t$
10	1	0	0	10		10=$

S_2 is moved to the front of its bucket, i.e. stays were it is. $T_1 = 2$ and $sufinv[2] = 5$, hence increment $count[5]$, set $sufinv[2] = 5 + count[5] - 1 = 5$, and $B2h[5] = 1$.

ind	Bh	B2h	count	sufinv	new pos	suftab
0	1	1	1	0		0=acbaacatat$
1	0	0	0	6	0	3=aacatat$
* 2	0	0	0	**5**		4=acatat$
3	0	0	0	**0**		6=atat$
4	0	0	0	0		8=at$
5	1	1	1	6	5	2=baacatat$
6	1	0	0	0		1=cbaacatat$
7	0	0	0	8		5=catat$
8	1	0	0	0		7=tat$
9	0	0	0	8		9=t$
10	1	0	0	10		10=$

S_3 is moved to the front of its bucket, i.e. to position 0. $T_2 = 3$ and $sufinv[3] = 0$, hence increment $count[0]$, set $sufinv[3] = 0 + count[0] - 1 = 0$, and $B2h[0] = 1$.

ind	Bh	B2h	count	sufinv	new pos	suftab
0	1	1	1	0		0=acbaacatat$
1	0	0	0	6	0	3=aacatat$
2	0	0	0	**5**		4=acatat$
* 3	0	0	0	**0**		6=atat$
4	0	0	0	0		8=at$
5	1	1	1	**6**	5	2=baacatat$
6	1	1	1	0		1=cbaacatat$
7	0	0	0	8	6	5=catat$
8	1	0	0	0		7=tat$
9	0	0	0	8		9=t$
10	1	0	0	10		10=$

S_5 is moved to the front of its bucket, i.e. to position 6. $T_3 = 5$ and $sufinv[5] = 6$, hence increment $count[6]$, set $sufinv[5] = 6 + count[6] - 1 = 6$, and $B2h[6] = 1$.

ind	Bh	B2h	count	sufinv	new pos	suftab
0	1	1	1	0		0=acbaacatat$
1	0	0	0	6	0	3=aacatat$
2	0	0	0	**5**		4=acatat$
3	0	0	0	**0**		6=atat$
* 4	0	0	0	0		8=at$
5	1	1	1	**6**	5	2=baacatat$
6	1	1	1	0		1=cbaacatat$
7	0	0	0	**8**	6	5=catat$
8	1	1	1	0	8	7=tat$
9	0	0	0	8		9=t$
10	1	0	0	10		10=$

S_7 is moved to the front of its bucket, i.e. to position 8. $T_4 = 7$ and $sufinv[7] = 8$, hence increment $count[8]$, set $sufinv[7] = 8 + count[8] - 1 = 8$, and $B2h[8] = 1$.

Now we have scanned the first bucket. Then we scan the bucket again, find all moved suffixes in all buckets and update the $B2h$ bitvector so that it points only to the leftmost positions of the $2h$-buckets, i.e. $B2h$ is set to false in the interval $[sufinv[a] + 1, b - 1]$ where a is every position marked in $B2h$ and $b = \min\{j : j > sufinv[a]$ and $(Bh[j]$ or not $B2h[j])\}$.

In our example nothing happens, since all moved suffixes are put at the beginning of a new bucket. This scan updates the $sufinv$ and $B2H$ tables and makes them consistent with the $2h$ order. At the end of each stage after all buckets are scanned, we update the $suftab$ array using the $sufinv$ array.

The next step shows that indeed the order of S_1 and S_5 is changed. S_5 was investigated during the scan of the first bucket and put to the beginning of its $2h$ bucket. Also, the $B2h$ vector changes now in the second scanning step.

ind	Bh	B2h	count	sufinv	new pos	suftab
0	1	1	1	0		0=acbaacatat$
1	0	0	0	**7**	0	3=aacatat$
2	0	0	0	**5**		4=acatat$
3	0	0	0	**0**		6=atat$
4	0	0	0	0		8=at$
* 5	1	1	1	**6**	5	2=baacatat$
6	1	1	2	0	7	1=cbaacatat$
7	0	1	0	**8**	6	5=catat$
8	1	1	1	0	8	7=tat$
9	0	0	0	8		9=t$
10	1	0	0	10		10=$

Now S_1 is put at the second position of its bucket. $T_5 = 1$ and $sufinv[1] = 6$, hence increment $count[6]$, set $sufinv[1] = 6 + count[6] - 1 = 7$, and $B2h[7] = 1$.

ind	Bh	B2h	count	sufinv	new pos	suftab
0	1	1	2	**1**	1	0=acbaacatat$
1	0	1	0	**7**	0	3=aacatat$
2	0	0	0	**5**		4=acatat$
3	0	0	0	**0**		6=atat$
4	0	0	0	0		8=at$
5	1	1	1	**6**	5	2=baacatat$
* 6	1	1	2	0	7	1=cbaacatat$
7	0	1	0	**8**	6	5=catat$
8	1	1	1	0	8	7=tat$
9	0	0	0	8		9=t$
10	1	0	0	10		10=$

Now S_0 is put at the second position of the first bucket. $T_6 = 0$ and $sufinv[0] = 0$, hence increment $count[0]$, set $sufinv[0] = 0 + count[0] - 1 = 1$, and $B2h[1] = 1$.

ind	Bh	B2h	count	sufinv	new pos	suftab
0	1	1	3	**1**	1	0=acbaacatat$
1	0	1	0	**7**	0	3=aacatat$
2	0	1	0	**5**	2	4=acatat$
3	0	0	0	**0**		6=atat$
4	0	0	0	**2**		8=at$
5	1	1	1	**6**	5	2=baacatat$
6	1	1	2	0	7	1=cbaacatat$
* 7	0	1	0	**8**	6	5=catat$
8	1	1	1	0	8	7=tat$
9	0	0	0	8		9=t$
10	1	0	0	10		10=$

Now S_4 is put at the third position of the first bucket. $T_7 = 4$ and $sufinv[4] = 0$, hence increment $count[0]$, set $sufinv[4] = 0 + count[0] - 1 = 2$, and $B2h[2] = 1$. We finished scanning this bucket and scan it again to update $B2h$, which in this case sets $B2h[2]$ to 0, since two suffixes with second character c were moved but $B2h$ should only mark the beginning of the $2h$-bucket.

In the original paper Myers describes a small modification of the construction phase which leads to an $O(n)$ expected time algorithm at the expense of another n bytes.

The idea is to store for all suffixes S_i their prefixes of length $T = \lfloor \log_{|\Sigma|} n \rfloor$ as $T-$digit radix-$\mid \Sigma \mid$ numbers.

Then instead of performing the radix sort on the first symbol of the suffixes, we perform it on this array, which can be done in time $O(n)$ since our choice of T guarantees that all integers are less than n. Hence the base case of the sort has been extended from 1 to T.

It can be shown that in the expected case there is only a constant number of additional rounds that need to be performed.

After constructing our suffix array we have the table $suftab$ which gives us in sorted order the suffixes of S. Suppose now we want to find all instances of a string $P = p_1, \ldots, p_m$ of length $m < n$ in S.

The let

$$L_P = \min\{k : P \leq_m S_{suftab[k]} \text{ or } k = n\}$$

and

$$R_P = \max\{k : S_{suftab[k]} \leq_m P \text{ or } k = -1\}.$$

Since $suftab$ is in $\leq_m$ order, it follows that P matches a suffix S_i if and only if $i = suftab[k]$ for some $k \in [L_P, R_P]$. Hence a simple binary search can find L_P and R_P. Each comparison in the search needs $O(m)$ character comparisons, and hence we can find all instances in the string in time $O(m \log n)$.

This is the simple code piece to search fo L_P.

Algorithm 19

```
1  if P ≤_m S_suftab[0]
2    then L_P = 0;
3     else if P >_m S_suftab[n−1]
4            then L_P = n;
5             else
6                  (L, R) = (0, n − 1);
7                  while R − L > 1 do
8                        M = (L + R)/2;
9                        if P ≤_m S_suftab[M]
10                         then R = M;
11                          else L = M;
12                       fi
13                 od
14                 L_P = R;
15         fi
16 fi
```

For example if we search for $P = aca$ in the text $S = acaaacatat\$$ $L_P = 2$ and $R_P = 3$. We find the value L_P and R_P respectively, by setting (L, R) to $(0, n-1)$ and changing the borders of this interval based on the comparison with the suffix at position $\lceil (L_P + R_P)/2 \rceil$ e.g. we find L_P with the sequence: $(0, 9) \Rightarrow (0, 5) \Rightarrow (0, 3) \Rightarrow (0, 2) \Rightarrow (1, 2)$. Hence $L_P = 2$.

0	aaacatat$
1	aacatat$
2	acaaacatat$
3	acatat$
4	atat$
5	at$
6	caaacatat$
7	catat$
8	tat$
9	t$
10	$

The binary searches each need $O(\log n)$ steps. In each step we need to compare m characters of the text and the pattern in the $\geq_m$ operations. This leads to a running time of $O(m \log n)$.

Can we do better?

As the binary search continues let L and R denote the left and right boundaries of the current search interval. At the start, L equals 0 and R equals $n-1$. Then in each iteration of the binary search a query is made at location $M = \lceil (R+L)/2 \rceil$ of $suftab$. We keep track of the longest prefixes of $suftab(L)$ and $suftab(R)$ that match a prefix of P. Let l and r denote the prefix lengths respectively and let $mlr = \min(l, r)$.

Then we can use the value mlr to accelerate the lexicographical comparison of P and the suffix $suftab[M]$. Since $suftab$ is ordered, it is clear that all suffixes between L and R share the same prefix. Hence we can start the first comparison at position $mlr + 1$. In practice this trick already brings the running time to $O(m + \log n)$, however one can construct an example that still needs time $O(m \cdot \log n)$ (exercise).

We call an examination of a character of P *redundant* if that character has been examined before. The goal is to limit the number of redundant character comparisons to one per iteration of the binary search.

The use of mlr alone does not suffice, since in the case that $l \neq r$ all characters in P from $mlr + 1$ to $\max(l, r)$ will have already been examined. Thus all comparisons to these characters are redundant. We need a way to begin the comparisons at the *maximum* of l and r.

To do this we introduce the following definition.

Definition 20 *$lcp(i, j)$ is the length of the longest common prefix of the suffixes specified in positions i and j of $suftab$.*

For example for $S = aabaacatat$ the $lcp(0, 1)$ is the length of the longest common prefix of $aabaacata$ and $aacata$ which is 2.

With the help of the lcp information, we can achieve our goal of one redundant character comparison per iteration of the search. For now assume that we know $lcp(i, j), \forall i, j$.

How do we use the lcp information? In the case of $l = r$ we compare P to $suftab[M]$ as before starting from position $mlr + 1$, since in this case the minimum of l and r is also the maximum of the two and now redundant character comparisons are made.

If $l \neq r$, there are three cases in which we assume w.l.o.g. $l > r$:

Case 1: $lcp(L, M) > l$:
Then the common prefix of the suffixes $suftab[L]$ and $suftab[M]$ is longer than the common prefix of P and $suftab[L]$.

Therefore, P agrees with the suffix $suftab[M]$ up through character l. Or put it differently, characters $l + 1$ of $suftab[L]$ and $suftab[M]$ are identical and lexically less then character $l + 1$ of P.

Hence any possible starting position must start to the right of M in $suftab$. So in this case no examination of P is needed. L is set to M and l and r remain unchanged.

Case 2: $lcp(L, M) < l$:
Then the common prefix of suffix $suftab[L]$ and $suftab[M]$ is smaller than the common prefix if $suftab[L]$ and P.

Therefore P agrees with $suftab[M]$ up through character $lcp(L, M)$. The $lcp(L, M) + 1$ characters of P and and $suftab[L]$ are identical and lexically less than the character $lcp(L, M) + 1$ of $suftab[M]$.

Hence any possible starting position must start left of M in $suftab$. So in this case again no examination of P is needed. R is set to M, r is changed to $lcp(L, M)$, and l remains unchanged.

Case 3: $lcp(L, M) = l$:
Then P agrees with $suftab[M]$ up to character l. The algorithm then lexically compares P to $suftab[M]$ starting from position $l + 1$. In the usual manner the outcome of the compare determines which of L and R change along with the corresponding change of l and r.

```
Illustration of the three cases:
case 1)                 case 2)               case 3)
P =  a b c d e m n      P =  a b c d e m n     P =  a b c d e m n
                lcp(L,M)          lcp(L,M)                lcp(L,M)
            l                       l                     l
L -> a b c d e f g....  L -> a b c d e f g...  L -> a b c d e f g....

M -> a b c d e f g....  M -> a b c d g g....   M -> a b c d e g....

R -> a b c w x y z....  R -> a b c w x y z...  R -> a b c w x y z....
         r                       r                     r
```

Then the following theorem holds:

Theorem 21 *Using the lcp values, the search algorithm does at most $O(m + \log n)$ comparisons and runs in that time.*

Proof: exercise. Use the fact that neither l nor r decrease in the binary search, and find a bound for the number of redundant comparisons per iteration of the binary search.

7.5 Computing the lcp values

We now know how to search fast in a suffix array under the assumption, that we know the lcp values for all pairs i, j.

But how do we compute the lcp values? And which ones? Computing them all would require too much time and, worse, quadratic space!

We will now first dicuss, which lcp values we really need, and then how to compute them. For the computation we sketch Myers' proposal for computing the lcp values during the construction of the suffix array and then give in more detail a newer, simple $O(n)$ algorithm to compute the lcp values given the suffix array $suftab$.

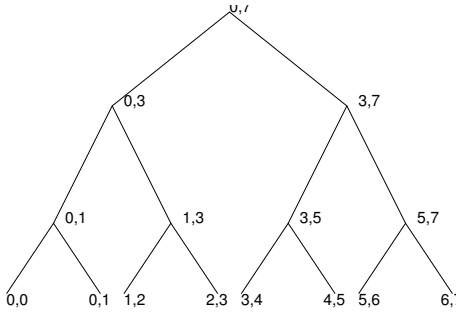
7.6 Computing the lcp values

We first observe, that indeed we only need the lcp values of the values of L and R we encounter in our binary search for L_P and R_P. However, these values are constant for each search, and there are only linearly many.

Observation 22 *Only $O(n)$ many lcp values are needed for the lcp based search in a suffix array.*

We will see that we get those values in a two step procedure:

1. Compute the lcp values for pairs of suffixes *adjacent* in $suftab$ suing an array $height$ of size n.
2. For the fixed binary search tree used in the search for L_P and R_P compute the lcp values for its internal nodes using the array $height$. (exercise)



Hence the essential thing to do is to compute the array $height$, i.e. the lcp values of adjacent suffixes in suftab.

This can be done during the construction of the suffix array, without additional overhead, or alternatively in in linear time with a scan over the suffix array.

In Myers' algorithm the computation of the lcp values can be done during the construction of the suffix array without additional time overhead and with an additional $n + 1$ integers. The key idea is the following. Assume that after stage h of the sort we know the lcps between suffixes in adjacent buckets (after the first stage, the $lcps$ between suffixes in adjacent buckets are 0).

At stage $2h$ the buckets are partitioned according to $2h$ symbols. Thus, the $lcps$ between suffixes in newly adjacent buckets must be at least h and at most $2h - 1$. Furthermore if S_p and S_q are in the same h-bucket but are in distinct $2h$-buckets, then

$$lcp(S_p, S_q) = h + lcp(S_{p+h}, S_{q+h}) \text{ and } lcp(S_{p+h}, S_{q+h}) < h.$$

The problem is that we only have the $lcps$ between suffixes in adjacent buckets, and S_{p+h} and S_{q+h} may not be in adjacent buckets. However, if $S_{suftab[i]}$ and $S_{suftab[j]}$ with $i < j$ have an lcp less than h and $suftab$ is in $\leq_h$ order, then their lcp is the minimum of the lcps of every adjacent pair of suffixes between $suftab[i]$ and $suftab[j]$. That is

$$lcp(S_{suftab[i]}, S_{suftab[j]}) = \min_{k \in [i,j-1]} \{lcp(S_{suftab[k]}, S_{suftab[k+1]}\}$$

Using the above formula to compute the lcp values directly would require too much time and maintaining the lcp for every par of suffixes too much space.

By using a balanced tree that records the minimum pairwise lcps over a collection of intervals of the suffix array, we are able to determine the lcp between any two suffixes in $O(\log n)$ time which is sufficient for Myer's online construction.

Since there are only n internal leaves in the tree, for which the lcp has to be computed, we spend a total of $O(n \log n)$ time to precompute the lcp values.

7.7 Computing the $height$ array

We now give the linear time construction of Kasai et al.

As noted above, it is sufficient to compute an array $height$ with $height(k) = lcp(S_{suftab[k-1]}, S_{suftab[k]})$, that means we compute in additional linear space the lcp values of suffixes that are adjacent in the suffix array.

We need again the inverse of the suffix array, the array $sufinv$ with the property $sufinv[suftab[i]] = i$. This can be computed in one linear scan over $suftab$.

Lets observe some properties of the lcp values.

Recall that

$$lcp(S_{suftab[i]}, S_{suftab[j]}) = \min_{k \in [i,j-1]} \{lcp(S_{suftab[k]}, S_{suftab[k+1]}\}.$$

This implies that the lcp of a pair of adjacent suffixes in the suffix array is greater than or equal to the lcp of a pair of suffixes that surrounds them.

Observation 23

$$lcp(S_{suftab[y-1]}, S_{suftab[y]}) \geq lcp(S_{suftab[x]}, S_{suftab[z]}), x < y \leq z$$

The next observation states that, whenever the lcp of adjacent suffixes is greater than 1, then the lexicographical order of the suffixes is preserved when the first character of each suffix is deleted.

Observation 24 *If* $lcp(S_{suftab[x-1]}, S_{suftab[x]}) > 1$, *then*

$$sufinv[suftab[x-1]+1] < sufinv[suftab[x]+1].$$

In this case the lcp between $S_{suftab[x-1]+1}$ and $S_{suftab[x]+1}$ is one less than the lcp between $S_{suftab[x-1]}$ and $S_{suftab[x]}$.

Observation 25 *If* $lcp(S_{suftab[x-1]}, S_{suftab[x]}) > 1$, *then*

$$lcp(S_{suftab[x-1]+1}, S_{suftab[x]+1}) = lcp(S_{suftab[x-1]}, S_{suftab[x]}) - 1.$$

Now consider the following problem: compute the lcp between a suffix S_i and its adjacent suffix in $suftab$ when the lcp between S_{i-1} and its adjacent suffix is known. For ease of exposition use the following notations:

```
sufinv       S          suftab
p-1          abcbdabe   j-1 = suftab[p-1]
p            abe        i-1 = suftab[p]
...
...          bcbdabe    j   = suftab[p-1]+1
...
...
q-1          bdabe      k   = suftab[q-1]
q            be         i   = suftab[q]
```

That means we want to compute $height[q]$ if $height[p]$ is given.

Then the following holds:

Lemma 26 *If* $lcp(S_{j-1}, S_{i-1}) > 1$ *then* $lcp(S_k, S_i) \geq lcp(S_j, S_i)$.

Proof: Since $lcp(S_{j-1}, S_{i-1}) > 1$ we have $sufinv[j] < sufinv[i]$ by Observation 24. Since $sufinv[j] \leq sufinv[k] = sufinv[i] - 1$ we get $lcp(S_k, S_i) \geq lcp(S_j, S_i)$ by Observation 23.

Now we can prove the following theorem:

Theorem 27 *If* $height[p] = lcp(S_{j-1}, S_{i-1}) > 1$ *then*

$$height[q] = lcp(S_k, S_i) \geq height[p] - 1.$$

Proof: By Lemma 26 $lcp(S_k, S_i) \geq lcp(S_j, S_i)$ and by Observation 25 $lcp(S_j, S_i) = lcp(S_{j-1}, S_{i-1}) - 1$.

That means whenever the lcp between suffix S_{i-1} and its adjacent suffix is h, suffix S_i and its adjacent suffix in $suftab$ have a common prefix of length at least $h-1$. Hence for computing the lcp between S_i and its adjacent suffix, it suffices to compare from the h-th character on.

7.8 The algorithm

The following algorithm computes the array $height$ following the above discussion in time $O(n)$:

Algorithm 28

```
1  GetHeight(S, suftab)
2  for i = 1 to n do
3      sufinv[suftab[i]] = i;
4  od
5  h = 0;
6  for i = 1 to n do
7      if sufinv[i] > 1
8        then
9              k = suftab[sufinv[i] − 1];
10             while S[i + h] = S[k + h] do
11                   h + +;
12             od
13             height[sufinv[i]] = h;
14             if h > 0 then h = h − 1; fi
15     fi
16 od
```

The above algorithm uses only linear time. In the loop in line 6 we iterate from 1 to n. In the loop is a while loop in line 10 that increases the height (i.e. the lcp value of adjacent suffixes). Since the height is maximally n and since in line 14 we decrease h by at most 1 per iteration of the main loop, it follows that the while loop can increase h at most $2n$ times in total.

Our overall strategy for constructing and searching a suffix array could then be as follows:

- Construct the suffix array for S in time $O(n \log n)$. (Linear time constructions are possible)
- Compute the $height$ array (for adjacent positions) in linear time.
- Precompute the search tree for the binary search and annotate its internal nodes with lcp values in time $O(n)$. (exercise)
- support $O(\log n + m)$ queries by adapting the searches for L_P and R_P.

8 Factor oracle (Knut Reinert)

This exposition is based on the following sources, which are all recommended reading:

1. Cyril Allauzen, Maxime Crochemore, Mathieu Raffinot: Factor oracle: a new structure for pattern matching, Conference on current trends in theory and practice in informatics, 1999
2. Cyril Allauzen, Maxime Crochemore, Mathieu Raffinto: Factor oracle, Suffix oracle: Technical report 99-08, Institut Gaspard-Monge, University de Marne-la-Vall'ee, 1999

8.1 Definition

Efficient pattern matching on fixed texts is often based on indices built on the text. Simple techniques are based on q-gram decomposition, while more advanced methods use more elaborate data structures.

The classical data structures in this field are: suffix trees, suffix arrays, suffix automata, and factor automata, which accept the set of all factors of the text. All these structures lead to very time-efficient pattern matching algorithms but require a fairly large amount of memory space.

The suffix array for example need only five bytes per characters, but does not allow linear search, while other structures need 12 or more bytes per character.

We introduce some notation. A word $x \in \Sigma^*$ is called a *factor* if it can be written as $x = uxv$ for with $u, v \in \Sigma^*$. We denote with $Fact(p)$ the set of all factors of p. We define the sets $Pref(p)$ and $Suff(p)$ analogously.

We denote with $pref_p(i)$ the i-th prefix of p. For $u \in Fact(p)$ we denote with $poccur(u, p) = \min\{\mid z \mid, z = wu \text{ and } p = wuv\}$ the ending position of the first occurrence of u in p.

For example for $p = abbbaab$, $poccur(bba, p) =\mid w \cdot u \mid=\mid ab \cdot bba \mid= 5$.

Finally for $u \in Fact(p)$ we define

$$endpos_p(u) = \{i \mid p = wup_{i+1} \dots p_m\}.$$

If two factors u and v of p have the same endpos set we denote $u \approx_p v$.

The goal of the factor oracle is to built for a string p an automaton with the following properties:

1. it is acyclic.
2. it recognizes at least the factors of p.
3. has as few states as possible.
4. has a linear number of transitions.

Such an automaton could then for example be used in multiple string matching algorithms such as SBOM.

The factor oracle for a word $p = p_1 \dots p_m$ has a constructive definition

Algorithm 29

```
1 BuildOracle(p_1 p_2 ... p_m)
2 for i = 0 to m do
3     i = new state;
4 od
5 for i = 0 to m - 1 do
6     i ->^{p_{i+1}} i + 1 = new transition;
```

```
 7 od
 8 for i = 0 to m − 1 do
 9     u = minimal length word in state i;
10     for all σ ∈ Σ, σ ≠ p_{i+1} do
11         if uσ ∈ Fact(p_{i−|u|+1} ... p_m)
12           then
13                 i →^σ i + poccur(uσ, p_{i−|u|+1} ... p_m);
14         fi
15     od
16 od
```

Lets look at an example, the factor oracle for $p = abbbaab$ a string over the alphabet $\Sigma = \{a, b\}$. The main loop consists of the following steps:

i	u	σ	$i- \mid u \mid$	poccur
0	ϵ	b	0	2
1	a	a	0	6
2	b	a	1	5
3	bb	a	1	5
4	bbb	-	-	-
5	bbba	-	-	-
6	aa	-	-	-

Note that all the transitions that reach state i of the oracle are labeled by p_i.

We now prove some results about the factor oracle to see whether it satisfies the desired properties.

8.2 Properties of the factor oracle

Lemma 30 *Let $u \in \Sigma^*$ be a minimal length word among the words recognized in state i of $oracle(p)$. Then, $u \in Fact(p)$ and $i = poccur(u, p)$.*

Proof: By induction on i. The lemma holds for $i = 0, 1$. Let u be a minimal length word recognized in state i. Consider the last transition on the path labeled by u leading from 0 to i. then there are two cases:

1. The transition was built in line 6. Then $u = zp_i$ with $z \in \Sigma^*$. z is a minimal length word for state $i - 1$. By the induction hypothesis $z \in Fact(p)$ and $i - 1 = poccur(z, p)$. Hence the claim follows.
2. The transition was built in line 13. Then it uses the transition $j \rightarrow^{p_i} i$. Hence $u = zp_i$ and z is a minimal length word for state j. By the induction hypothesis and the construction of the transition in line 13, $u \in Fact(p)$ and $i = poccur(u, p)$.

From the above lemma follows directly (because of the the fact that $i = poccur(u, p)$):

Corollary 31 *Let $u \in \Sigma^*$ be a minimal length word among the words recognized in state i of $oracle(p)$, then u is unique.*

We now denote with $min(i)$ the minimal length word of state i.

Corollary 32 *Let i and j be two states of $oracle(p)$ with $j < i$. Let $u = min(i)$ and $v = min(j)$, then u can not be a suffix of v.*

Proof: exercise (hint: prove the claim indirectly).

Lemma 33 *Let i be a state of $oracle(p)$ and $u = min(i)$. Then u is a suffix of any word $c \in \Sigma^*$ which is the label of a path leading from state 0 to state i.*

Proof: By induction on i. The claim holds for state 0 and 1. Consider state i and let $u = min(i)$. Let $c = c_1 a$ be a path leading to state i, where c_1 leads to a state $j < i$. Let $v = min(j)$. Then it holds:

1. $\mid va \mid \geq \mid u \mid$ because u is the minimum of i and va leads to i.
2. according to line 13, $i \in endpos_p(va)$ because $v = min(j)$ and there is a transition $j \rightarrow^a i$.
3. according to Lemma 30 $i \in endpos_p(u)$.

Hence, u is a suffix of va. Since $j < i$, by the hypothesis, v is a suffix of c_1 and therefore u is a suffix of c.

Lemma 34 *Let $w \in Fact(p)$, then w is recognized by $oracle(p)$ in a state $j \leq poccur(w, p)$.*

Proof: By induction on the length f of $w = w_0 w_1 \dots w_f$. We denote $i = poccur(w, p)$.

1. For $f = 0$ there is a transition by w_0 leading to a state $k_0 \leq i - f$.
2. Assume there is a path labeled by $w_0 \dots w_j$ leading to a state $k_j \leq i - f + j$. Let $u = min(k_j)$. According to Lemma 33, $u = w_{j-|u|+1} \dots w_j$. As uw_{j+1} is a factor of p and since $i - f + j + 1 \in endpos_p(uw_{j+1})$, there is a transition labeled w_{j+1} leading to a state $k_{j+1} \leq i - f + j + 1$.

Note that the above $\leq$ can indeed be a $<$ (exercise: find an example).

Corollary 35 *Let $w \in Fact(p)$. Then every word $v \in Suff(w)$ is recognized by $oracle(p)$ in a state $j \leq poccur(w, p)$.*

Lemma 36 *Let i be a state of $oracle(p)$ and $u = min(i)$. Any path ending by u leads to a state $j \geq i$.*

Proof: exercise (hint: proof by induction on i).

Lemma 37 *Let $w \in \Sigma^*$ be a word recognized by $oracle(p)$ in i, then any suffix of w is recognized in a state $j \leq i$.*

Proof: exercise (hint: proof by induction on $\mid w \mid$).

Finally we can now consider the number of transitions:

Lemma 38 *The number $T_{or}(p)$ of transitions in oracle($p = p_1 \dots p_m$) satisfies $m \leq T_{or}(p) \leq 2m - 1$.*

Proof: There are always m transitions $i \rightarrow^{p_{i+1}} i + 1$. As the words a^m only have these transitions, m is a lower bound. Consider the transitions $i \rightarrow j$ with $j > i + 1$. We now build an injective function that maps each of these transitions to a proper suffix of p.

Each transition $i \rightarrow^\sigma j$ with $j > i + 1$ is mapped to $min(i)\sigma p_{j+1} \dots p_m$. The set of proper suffixes of $p = p_1 p_2 \dots p_m$ is of size $m - 1$. Hence $T_{or} \leq 2m - 1$. This bound is achieved for the words $a^{m-1}b$.

Lemma 39 *The above mapping is injective.*

Proof: Assume it is not and there are two distinct transitions $i_1 \rightarrow^{\sigma_1} j_1$ and $i_2 \rightarrow^{\sigma_2} j_2$ such that

$$min(i_1)\sigma_1 p_{j_1+1} \dots p_m = min(i_2)\sigma_2 p_{j_2+1} \dots p_m$$

Assume w.l.o.g $i_1 \geq i_2$. Consider the following cases:

1. If $j_1 = j_2$, then $\sigma_1 = \sigma_2$ and the above equality implies $min(i_1) = min(i_2)$ and hence $i_1 = i_2$.
2. If $j_1 > j_2$ then $min(i_2)\sigma_2$ is a propoer prefix of $min(i_1)$. Let $\delta = \mid min(i_1) \mid - \mid min(i_2)\sigma_2 \mid$, then we have $j_2 = j_1 - \delta - 1$. Since an occurrence of $min(i_1)$ ends in i_1 (according to Lemma 30), an occurrence of $min(i_2)\sigma_2$ ends in $i_1 - \delta < j_1 - \delta - 1 = j_2$. This is a contradiction to the construction of $oracle(p)$.
3. If $j_1 < j_2$ then $min(i_1)$ is a proper prefix of $min(i_2)$, so there is an occurrence of $min(i_1)$ ending before $i_2 \leq i_1$. This is a contradiction to Lemma 30.

We now know, that the factor oracle represents indeed at least the factors of p. In addition it contains only linearly many transitions.

The only thing left to do is to show that we can indeed contruct it in linear time. We can indeed do this by adding one letter at a time going from left to right.

8.3 The linear time algorithm

Denote with $repeat_p(i)$ the longest suffix of $pref_p(i)$ that appears at least twice in $pref_p(i)$. Define a function S_p defined on the states of the automaton, called *supply function* that maps each state $i > 0$ of $oracle(p)$ to a state j in which the reading of $repeat_p(i)$ ends. We arbitrarily set $S_p(0) = -1$.

Note that $S_p(i)$ is well defined for every state i of $oracle(p)$ (see Corollary 35) and that for any state i of $oracle(p)$, $i > S_p(i)$ (Lemma 34).

We denote $k_0 = m, k_i = S_p(k_{i-1})$ for $i \geq 1$. The sequence $CS_p = \{k_0 = m, k_1, \ldots, k_t = 0\}$ is called the suffix path of p in $oracle(p)$.

Lemma 40 *Let $k > 0$ be a state of $oracle(p)$ such that $s = S_p(k)$ is strictly positive. We denote $w_k = repeat_p(k)$ and $w_s = repeat_p(s)$. Then w_s is a suffix of w_k.*

We now consider for a word $p = p_1 p_2 \ldots p_m$ and a letter $\sigma \in \Sigma$ the construction of $oracle(p\sigma)$ from $oracle(p)$. Denote with $oracle(p) + \sigma$ the automaton $oracle(p)$ on which the transition $m \rightarrow^\sigma m + 1$ is added. Hence the only difference between $oracle(p\sigma)$ and $oracle(p) + \sigma$ is some transitions to $m + 1$ which are labeled by σ.

Lemma 41 *Let k be a state of $oracle(p) + \sigma$ such that there is a transition $k \rightarrow^\sigma m + 1$ in $oracle(p\sigma)$. Then k has to be on the suffix path CS_p in $oracle(p) + \sigma$.*

Among the states on the suffix path of p, every state that has no transition by σ in $oracle(p) + \sigma$ must have one in $oracle(p\sigma)$. More formally:

Lemma 42 *Let $k_l < m$ be a state on the suffix path CS_p of state m in $oracle(p) + \sigma$. If k_l does not have a transition by σ in $oracle(p)$, then there is a transition by σ from k_l to $m + 1$ in $oracle(p\sigma)$.*

Lemma 43 *Let $k_l < m$ be a state on the suffix path CS_p of state m in $oracle(p) + \sigma$. If k_l has a transition by σ in $oracle(p) + \sigma$, then all the states $k_i, 0 \leq i \leq t$ also have a transition by σ in $oracle(p) + \sigma$.*

Te above Lemmata 41,42, and 43 we have to do the following to transform $oracle(p) + \sigma$ into $oracle(p\sigma)$:

We have to go down the suffix path $CS_p = \{k_0 = m, k_1, \ldots, k_t = 0\}$ of state m and while the current state k_l does not have an existing transition by σ, a we add a transition $k_l \rightarrow^\sigma m + 1$ (Lemma 42). If k_l has already one, the process ends because, according to Lemma 43, all the states k_j after k_l on the suffix path already have a transition by σ.

We only need to know how to compute the supply function value $S_{p\sigma}(m + 1)$. This is done with the following lemma.

Lemma 44 *If there is a state k_d which is the greatest element of $CS_p = \{k_0 = m, k_1, \ldots, k_t = 0\}$ in $oracle(p)$ such there is a transition by σ from k_d to a state s in $oracle(p)$, then $S_{p\sigma}(m+1) = s$ in $oracle(p\sigma)$, else $S_{p\sigma} = 0$.*

From the above lemmas we can deduce the algorithm AddLetter.

Algorithm 45

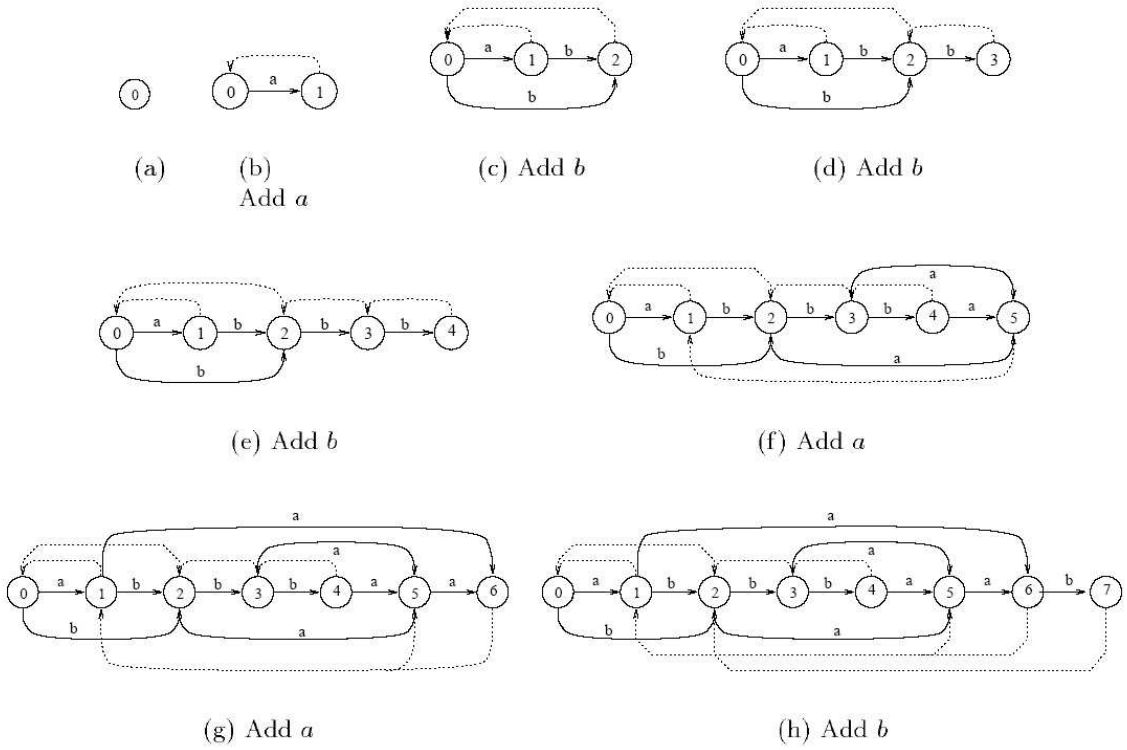
```
1  AddLetter(oracle(p_1 p_2 ... p_m, σ))
2  m + 1 = new state;
3  m →^σ m + 1 = new transition;
4  k = S_p(m);
5  while k > −1 ∧ ∄ transition from k by σ do
6        k →^σ m + 1 = new transition;
7        k = S_p(k);
8  od
9  if k = −1 then
10             s = 0;
11       else
12             s = sink of transition from k by σ;
13 fi
14 S_pσ(m + 1) = s;
15 return oracle(p = p_1 ..., p_m σ);
```

Lemma 46 *The algorithm AddLetter builds $oracle(p = p_1p_2 \ldots p_m\sigma)$ from $oracle(p = p_1p_2 \ldots p_m)$ and updates the supply value of state $m + 1$ of $oracle(p\sigma)$.*

Proof: We go down the suffix path of p in accordance with Lemma 42. We stop in accordance with Lemma 43 and update the supply value of state $m + 1$ according to Lemma 44.

Algorithm 47

```
1  OracleOnline(p_1 p_2 ... p_m)
2  Create oracle(ε) with state 0;
3  S_ε = −1;
4  for i = 1 to m do
5      oracle(p = p_1 ... p_i) = AddLetter(oracle(p_1 ... p_{i−1}, p_i);
6  od
```



Theorem 48 *The algorithm OracleOnline(p) builds* $oracle(p)$.

Proof: By induction on word p using lemma 46.

Theorem 49 *The complexity of OracleOnline(p) is* $O(m)$ *time and space.*

Proof: The algorithm works in space $O(m)$ since all transitions that are created are of $oracle(p)$. $m+1$ states are created and additional supply links are a constant overhead.

It also needs time $O(m)$. Since it only creates a linear number of transitions, we only need to verify that the number of backward jumps is linear in total. But this is also straightforward. For each jump backwards we introduce a new transition. Hence the total number of those jumps is linear.