EFFICIENT ON-LINE CONSTRUCTION AND
CORRECTION OF POSITION TREES

Mila E. Majster

TR79-393

Department of Computer Science
Cornell University
Ithaca, New York 14853

EFFICIENT ON-LINE CONSTRUCTION AND CORRECTION OF POSITION TREES

Mila E. Majster*

Department of Computer Science
Cornell University

A. Reiser

Technische Universität
München, Germany

Abstract

This paper presents an on-line algorithm for the construction of position trees, i.e. an algorithm which constructs the position tree for a given string while reading the string from left to right. In addition an on-line correction algorithm is presented which--upon a change in the string--can be used to construct the new position tree. Moreover, special attention is paid to computers with small memory. Compactification of the trees and transport costs between main and secondary storage are discussed.

Key Words

Pattern matching, position tree, on-line algorithm.

* Permanent address: Technische Universität, München.

Introduction

Text-editing systems, symbol manipulation problems as well as a number of other computer applications often require a search function which locates instances of a given string within a larger main string (P1). In some applications all positions, in others the leftmost position have to be found. Other pattern matching problems are to search for the occurrences of the elements of a set of pattern strings within a given main string s (P2), to find the longest repeated substring of the main string s (P3), the internal matching problem, i.e. to find for each position i in $s = s_1 \dots s_n$ another position j in s such that the common prefix of $s_i s_{i+1} \dots s_n$ and $s_j s_{j+1} \dots s_n$ is not shorter than the longest common prefix of $s_1 \dots s_n$ and $s_k \dots s_n$, $k \neq i$, $k \neq j$ (P4). For the external matching problem (P5) we consider two strings $s = s_1 \dots s_n$ and $s' = s'_1 \dots s'_m$ and a position i in s and search for a position j in s' such that the longest common prefix of $s_i \dots s_n$ and $s'_j \dots s'_m$ is not shorter than the longest common prefix of $s_1 \dots s_n$ and $s'_k \dots s'_m$, $k \neq j$. Another problem is concerned with finding the longest common substring of two strings s and s' .

A naive algorithm for the solution of problem P1, where all possible alignments are tried successively takes $O(n \cdot m)$ steps, where n is the length of the main string and m the length of the pattern. In 1970 [KMP] showed how to solve P1 in time proportional to $(n+m)$. This algorithm basically involves a preprocessing of the pattern string in order to construct a table which stores information of how far the pattern can be shifted against the main string if a mismatch at position j in the pattern occurs. This table is then used as a data structure in the algorithm. If we consider P2 where the k patterns $p_1, \dots, p_k$ are searched for in the same string s consecutively the above algorithm would take $O(k \cdot |s| + |p_1| + \dots + |p_k|)$. This is one of the reasons which led in [W] to the development of an auxiliary data structure, the prefix tree, storing information about the main string. The prefix tree is called a position tree in [AHU]. Once a compacted version of the tree is constructed in $O(n)$ steps where n is the length

of the main string a single search for a pattern costs time linear in the length of the pattern.

In [McC] a similar data structure, serving the same purposes as the position tree, i.e. to reduce the complexity of pattern matching algorithms is introduced. In addition this data structure is space economical.

At this point it is now important to note that the position tree construction algorithm processes the main string s from right to left. This feature presupposes that the whole text must be known before we can start to build the position tree. Similarly, another solution for the pattern matching problems P1, P2 [CC] presupposes that at least the length of the main string and the set of patterns are known in advance.

If we have to wait with the construction of the position tree until the whole main string is known we must face some considerable drawbacks: D1) It is not possible to answer pattern matching problems and perform corrections, if necessary, for that part of the main string which has been already read in. This is particularly annoying if we consider for example a text editing system where the pattern matching is used to find those positions in the text which have to be corrected. Here, one would like to process that part of the string which is already known. In a typical text editing system with a usually small computer dedicated to the text editing job we must face further considerable drawbacks, namely D2) the processing unit keeps waiting until the input device has scanned the last symbol of the input, D3) as the main store will be usually too small a considerable part of the text has to be transported on to secondary storage until it is going to be processed. The position tree construction algorithm of [W] constructs the position tree for $s_1 \dots s_n$ from the position tree for $s_{i+1} \dots s_n$. In general the position tree T_{i+1} for $s_{i+1} \dots s_n$ will be too large to be kept as a whole in the main store. So we have to decompose the tree and to shift parts of it to secondary store. Unfortunately, as the letter s_i will not be known in advance, one cannot predict

which parts of the tree T_{i+1} will be needed for the construction of the tree T_i for $s_i \dots s_n$. Therefore we have the problem D4) of transferring considerable amounts of data between main and secondary storage. Hence, we are looking for a possibility to construct the position tree in an on-line way. Moreover, we are interested in answering pattern matching questions and in the possibility of "updating" or "correcting" that part of the string which has been already read. And last, we want to get rid of problem D4).

To solve D1) and D2) there is an immediate solution. Instead of treating the string $s = s_1 \dots s_n$ we consider the reversed string $s_n \dots s_1$ for which the position tree can be constructed from right to left as usual. This solution is hardly acceptable for two reasons. First, it does not solve D3 and D4. Second, it can be only used under the assumption that--for the problems P1, P2, P3--the pattern can be reversed before the request. It is then not possible to start a search before a pattern is completely given. In particular, requests, which are looking for a prefix p_1 of the pattern $p = p_1 p_2$, and if this can be located, ask for the rest of the pattern, cannot be treated in this way. Moreover, the pattern matching problems P4, P5 cannot be solved with the position tree for the reversed string.

1. Preliminaries

In this paper we will use the following notations. An alphabet Σ is a finite set of symbols. A string over an alphabet Σ is a finite-length sequence of symbols from Σ . The empty string denoted by ϵ is the string with no symbols. If x and y are strings, then the concatenation of x and y is the string xy . If xyz is a string, $x, y, z \in \Sigma^*$, then x is a prefix, y a substring and z is a suffix of xyz . The length of a string x , denoted by $|x|$, is the number of symbols in x .

A position in a string of length n , $n \geq 1$, is an integer between 1 and n . The symbol

$a \in \Sigma$ occurs in position i of string x if $x = yaz$ with $|y| = i-1$. Let $\$ \notin \Sigma$.¹

A position identifier for position i in $x\$$ ² is the shortest substring u of $x\$$ such that

- i) $x\$ = yuz$ $|y| = i-1$
- ii) if $x\$ = y'uz'$ then $y = y'$, $z = z'$

A Σ -tree is a labeled tree T such that for each node N in T the edges leaving N have distinct labels in Σ . If the edge (N, M) in T is labeled by a , we call M the a -son of N .

A position tree for a string $x\$ = x_1 \dots x_{n+1}$ where $x_i \in \Sigma$, $1 \leq i \leq n$, is a $(\Sigma \cup \{\$\})$ -tree T such that

- i) T has $n+1$ leaves labeled $1, \dots, n+1$. The leaves of T are in one-to-one correspondence with the positions in $x\$$.
- ii) The sequence of labels of edges on the path from the root to the leaf labeled i is the position identifier for position i .

Note that there is exactly one position tree for each string.

Example The position tree for the string $abba\$$ is given by

¹ We use $\$$ as an endmarker for strings over Σ .

² The endmarker is needed to guarantee the existence of a position identifier for each position.

2. On-line Construction of Position Trees

The problem which we are going to solve in the following is: Let a string $x = x_1x_2\ldots$ be read from left to right without knowing the whole string in advance and construct after the reading of each letter the position tree for the actual prefix. The problem with the construction of the position tree when reading the text from left to right is based on the fact that we need an endmarker $\$$ for each string in order to guarantee that there exists a position identifier for each position in the string. This has the consequence that reading from left to right means the transition from

$$x_1x_2\ldots x_i\$ \text{ to } x_1\ldots x_ix_{i+1}\$.$$

The efficiency of Weiner's algorithm stems from the fact that the changes which are caused by updating the tree reading the text from right to left are "local." If we work from left to right changes are no longer "local." In particular we have to solve the following two problems

- i) a position identifier may become invalid by reading a new symbol, as e.g. the position identifier for position 1 in

$$abcb\$ \rightarrow abcba\$$$

- ii) all position identifiers which contain the endmarker have to be changed whenever a new symbol is read in, e.g. the position identifier for position 4 in

$$abcb\$ \rightarrow abcba\$$$

In the following we describe how to construct the position tree for a string $xa\$$ from the position tree for $x\$$, where $x \in \Sigma^*$, $a \in \Sigma$.

Algorithm: Position tree on-line

- 1) For each node N that has a $\$$ -son N' the following steps have to be performed. The order in which the nodes with $\$$ -sons have to be processed is given in Lemma 1 below.

- a) If N does not have an a -son then replace the $\$$ -symbol by a
- b) If N has an a -son N'' that is not a leaf then remove the edge between N and N' and make N' the $\$$ -son of N'' (together with the position number of N')
- c) If N has an a -son N'' that is a leaf then remove the edge between N and N' and make N' the $\$$ -son of N'' together with the position number associated with N' . Moreover, attach a new son to N'' , transfer the position number j of N'' to the new son; label the edge between N'' and its new son by the $(j+1)$ -th letter in $xa\$$, where l is the length of the position identifier for position j in $xa\$$.

2) Attach a $\$$ -son at the root and give it the next position number.

Example Consider the string $xa\$ = abbab\$$. We construct the position tree for $xa\$$ from that for $x\$$. The position tree for $x\$$ is

The father of leaf 4 falls into case a). The father of leaf 5 falls into case c), the father of leaf 6 falls into case b). Performing the algorithm for the father of leaves 4,5,6 (in that order) yields

We now want to make sure that the algorithm works correctly.

Lemma 1. Assume that step 1 of the above algorithm is performed successively for all nodes N with a $\$$ -son in such a way that if a node is processed then all his descendants have been processed previously. Then the algorithm constructs the position tree for $xa\$$ from the position tree for $x\$$.

Proof

1) Let us first make sure that for each node M in the tree constructed by the algorithm and for each $oc \in \Sigma$ there is at most one edge with label o starting in M . This is true by the observation that steps 1a, 1c evidently result in a tree with the requested property. For step 1b we must show that N'' does not have a $\$$ -son before this step is performed. This is guaranteed by the fact that N'' is a descendant of N and is--by assumption--processed before N . Step 1 removes in each case the $\$$ -edge from the considered node. Hence, step 2 also results in a tree with the requested property.

2) The next fact to be verified is that for each position identifier which is affected by the new letter a there is a change in the tree reflecting this change. There are three possibilities for a position identifier to be affected.

- a) The endmarker $\$$ is part of the position identifier. Let $x_1 \dots x_m \$$ be the position identifier for position i in $x_1 \dots x_m \$$; hence there must be positions $j_h, 1 \leq h \leq k$, with position identifier $x_i x_{i+1} \dots x_m r_{j_h}, 1 \leq h \leq k, r_{j_h} \in \Sigma^*$. If $r_{j_h} \notin \{a\}^*$ for all $h, 1 \leq h \leq k$, the new position identifier for position i will be $x_1 \dots x_m a$, which is achieved by step 1a of the algorithm. If there is $h_0, 1 \leq h_0 \leq k$ where $r_{j_{h_0}} = a s_{j_{h_0}}$ the position identifier for i becomes $x_1 \dots x_m a \$$ by step 1b or 1c.

- b) For a position j with identifier $x_j \dots x_{j+r-1}$ ^a there is a position i with identifier $x_i \dots x_m$ such that $x_j \dots x_{j+r-1} = x_i \dots x_m$. The position identifier for position j must be prolonged to $x_j \dots x_{j+r+1}$ which is achieved by step 1c.
- c) The position identifier for a position i in xa starts with the letter a . Then either a occurs only once in x then the position identifier for i is obtained by step 1c, or a occurs more than once in x then the position identifier for position i is obtained by 1b, or a does not occur in x then the position identifier for the new letter a is obtained by step 1a.

Lemma 2. Let $L = N_1, \dots, N_k$ be a list of all nodes that have a $\$$ -son in the position tree for x , but ordered such that $(*) N_i$ is a descendant of $N_j \Rightarrow i < j$. If we perform the algorithm processing the nodes that have a $\$$ -son in the order given by L then we can sequentially update L to get a new list L' that contains all nodes that have a $\$$ -son in the position tree for xa and that fulfills $(*)$. The cost for constructing L' from L is $O(k)$.

Proof

Let $L = N_1, \dots, N_k$ be a list for the position tree for x fulfilling $(*)$. Let us perform step 1 of the algorithm according to this list, i.e. we start with N_1 , continue with N_2 and so on. To manipulate the list we perform the following steps.

- If step 1a is performed with N_1 just remove N_1 from the list.
- If step 1b is applied to N_1 then replace N_1 by its a -son.
- If step 1c is applied to N_1 then replace N_1 by its a -son.
- If step 2 is performed attach the root at the end of the list.

Let L' be the list obtained by the above steps. Let N, M be two nodes with a $\$$ -son in the tree for $xa\$$. Let $M \neq \text{root}$ and N be a descendant of M . Then the father of N is a descendant of the father of M . The only possibility for a node (except the root) to be inserted into the list is to be a substitute for its father. Hence the list L' fulfills (*). The cost for updating L is obviously $O(k)$ as each node has to be processed and the cost for each node is constant.

3. Costs of the Algorithm

In order to be able to analyse the costs of the algorithm we assume that

- 1) we hold the text that has been already read in an array
- 2) we represent the tree in the following form:

Each node is represented by a natural number; in particular the root is represented by 0. We associate with each node three fields. The first contains the position number m , if the node is a leaf corresponding to position m . If the node is not a leaf then this field contains a list of the sons¹, each given by its number and the label of the edge leading to it. The second field contains information about the depth of the node. The depth of the root is 0. The third field serves for linking those nodes that have a $\$$ -son into a list. For example, the tree for $abbabb\$$ is

¹We will assume that the $\$$ -son--if any--is always the first in the list of sons.

which is represented by

NR	son or p.n.	depth	successor in list
0	(12,\$) (1,a), (2,b)	0	-
1	(6,b)	1	-
2	(9,\$), (3,a), (4,b)	1	0
3	3	2	-
4	(7,\$), (11,a)	2	2
5	4	4	-
6	(8,b)	2	-
7	5	3	-
Head of list → 8	(5,\$), (10,a)	3	4
9	6	2	-
10	1	4	-
11	2	3	-
12	7	1	-

Based on this representation the costs for step 1a, 1b, 1c for each node in the list is $O(\text{number of sons})$, as we first check if the node has an a-son. The test whether a node is a leaf and the update can be done in constant time; in particular the search for the letter following the position identifier in the text for step 1c can be performed by selecting the $(m+h)$ -th component of the array where m is the position number of the leaf and h its depth. Hence step 1 takes for each node $O(|S|)$ where S is the set of its sons. Step 2 costs constant time. Hence, the cost for constructing the position tree for $xa\$$ from the position tree for $x\$$ can be bounded above by $O(\sum_{i=1}^k |S_i| + 1)$, where S_i is the set of sons of the i -th node N_i in the list $L = N_1, \dots, N_k$. This cost can be bounded above by $O(|L| \cdot |\Sigma| + 1)$. The cost of the on-line construction of the position tree for $x\$, x \in \Sigma^{n-1}$ can be given by $O(\sum_{i=1}^n |p_i(x)| \cdot |\Sigma|)$, where $p_i(x)$ is the position identifier for the position i in $x\$$. This is based on the fact that for each position i the identifier $p_i(x)$ has to be updated as often as $|p_i(x)|$. Hence,

the father of the leaf with position number i can occur at most as often as $|p_i(x)|$ in the disjoint union of all lists $\cup L_j$, where L_j is the list of nodes with j -sons in the j -th application of the algorithm.

In terms of the average length l_x for a position identifier in the string $x\$$, $x \in \Sigma^{n-1}$, i.e.

$$l_x = \frac{\sum_{i=1}^n |p_i(x)|}{n}$$

the cost of the algorithm for constructing the position tree for $x\$$ can be given by $O(l_x \cdot n \cdot |\Sigma|)$.

4. Compactification

Concerning the space used by the algorithm we note that the position tree for a text of length n may have $O(n^2)$ vertices, as can be seen from the example $a^i b^i a^i b^i \$$. However, one can show that there is a compacted form of the position tree which needs only $O(n)$ space [W]. Here, compacted means that successive edges corresponding to single sons are contracted into one edge named by a string, e.g.

is compacted into

The question how such string-labeled trees can be efficiently represented is discussed in [McC].

The compacted position tree can be constructed in the same way as the non-compacted one. This can be seen as follows: if we start with an already compacted tree for $x\$$ and want to construct the compacted tree for $xa\$$ then at most those nodes which are in the list L are candidates for compactification.

We only sketch the algorithm for the on-line construction of compacted position trees and do not want to go into implementation and analysis details.

The following algorithm takes as input a compacted tree T for $x\$$. Each node in the tree T that has a single son is marked. In steps 1 and 2 the algorithm manipulates the tree basically in the same way as in the non-compacted case. In addition we have to take care of the marks: a mark has to be removed from a node if this node gets a second son and a mark has to be attached to a node, if the node has lost all but one son. After the steps necessary to reflect the new identifiers in the tree there may exist two successive nodes with marks which means that the tree is no longer compact and has to be compactified in step 3.

Algorithm: Compacted position tree on-line

1) For each node N that has a $\$$ -son N' do: (The order in which the nodes with $\$$ -sons are processed is given in Lemma 1).

a) If N does not have an ar-son, $re\bar{L}^*$, then

a1) change the label of the edge (N, N') from $\$$ to a

b) If N has an ar-son N'' , $re\bar{L}^*$, then

b1) remove the edge (N, N') ,

b2) mark N by $*$ if there is exactly one son left,

b3) if $r \neq \epsilon$ (this implies that N'' is not a leaf) then create a new node NN , make the sons of N'' the sons of NN , make NN the r -son of N'' , if N'' was marked by $*$ transfer the mark to NN , change the label of the edge (N, N'') from ar to a

if $r = \epsilon$ and N'' is not a leaf then remove mark at N'' , if any
if $r = \epsilon$ and N'' is a leaf then attach a new son to N'' , transfer the position number j of N'' to the new son, label the edge between N'' and its new son by the $(j+l)$ -th letter in $xa\$$ (where l is the length of the position identifier for position j in $x\$$).

b4) attach N' as $\$$ -son to N'' (together with the position number of N')

- 2) Attach a $\$$ -son to the root, give it the next position number, remove mark at root, if any.
- 3) For all pairs of adjacent nodes marked by $*$ compactify.

Let us briefly consider how we can find two adjacent nodes marked by $*$. New marks are only introduced in step 1.b2) at the node N if it has only one son, namely N'' . If N'' was marked, the mark is either removed or transferred because N'' gets a new son, N' . Therefore, only if the father of N was marked, we get two adjacent marked nodes by the introduction of new marks. Existing marks are only transferred to new created sons which cannot result in two adjacent marked nodes. Thereby it is sufficient to maintain for each node which has a $\$$ -son (members of the list L) the father if it is marked. This information can be easily obtained at the time when a node becomes a member of the list because it gets there as a substitute for its father.

Then we compactify and get

We continue with $xbbs = a^3 b^3 a^3 b^3 \$$. Steps 1 and 2 yield

Then we compactify

5. Main store and transport cost

Let us now consider the problem D4, i.e. given a small computer, we consider the question how we can keep the cost of main store low and how we can reduce cost for transport between main and secondary storage. We shall assume a paging system in the following.

Let us first see which parts of the tree are actually involved if we construct the position tree for $xa\$$ from that for $x\$$. First we need all nodes N with $\$$ -sons together with the information which sons N has. Second, if N has an a -son, this son is needed and third, the $\$$ -son is needed.

A first but, as we will see, not efficient approach to our problem could be to store the tree structure without the links for the list L in secondary store and to hold the information about the nodes with a \$-son in main store. This is done

by maintaining a list of references to the nodes with \$-son. The ordering given by this list should correspond to the one given by L.

If we look closer to this solution we find that the following steps have to be performed

- 1a) if a \$-son of N is transformed into an a-son
 - i) the node N must be removed from the list. Its predecessor in the list must be linked with its successor. This step does not cause any page transfer because L is held in main store.
 - ii) in the list of references to sons of N the \$ must be replaced by an a. Hence the page containing the list of references to sons of N has to be rewritten.
- 1b) if a \$-son has to be transferred from N to its a-son which is not a leaf, we have
 - i) to change the list of references to sons of N
 - ii) to change the list of references to sons of the a-son
 - iii) to change the height of the \$-son of N
 - iv) to change the reference from the predecessor of N to N (in the list of nodes with \$-sons) into a reference from the predecessor of N to the a-son of N
 - v) to erase the reference of N to its successor
 - vi) to write a reference from the a-son of N to the successor.

Hence, the page containing the list of references to sons of N, the page containing the list of references to the sons of the a-son of N and the page containing the \$-son of N have to be rewritten. In addition the list in main store is manipulated.

- 1c) if a \$-son has to be transferred from N to its a-son which is a leaf we have

- i) -vi) to change as above
- vii) to read one of the pages on which the text is situated
- viii) create a new son for the a-son of N

Here, in addition to the changes of 1b) we have to read one text page, to write the new node on some free space and to insert a reference to this new node into the list of references to the sons of the a-son of N.

If we assume that the references to the sons of a node are all stored on one page 1a) causes 1 page transfer, 1b) may cause 3 page transfers and 1c) may cause 4 page transfers disregarding the cost of writing the new node. In order to reduce the number of page transfers we propose to modify slightly the representation of the position tree in the following way:

- I We do not store the \$-sons in the tree representation but keep them separately
- II For each \$-son we store a reference to its father
- III Instead of maintaining the list of references to nodes with \$-sons we maintain a list $L_{\$}$ of \$-sons which is kept in main store.

With this choice we get for our example abbabb\$

Nr.	references to sons or p. n.	depth
0	(1,a)(2,b)	0
1	(5,b)	1
2	(4,b)(3,a)	1
3	3	2
4	(7,a)	3
5	(6,b)	2
6	(8,a)	3
7	2	3
8	1	4

plus the list

Nr.	father	next	pos. nr.
I	6	II	4
II	4	III	5
III	2	IV	6
IV	0	-	7

Assuming this representation we review the page transports for our algorithm.

As we keep the list of $\$$ -sons in main store, we must for each element in the list $L_{\$}$ check if its father has an a-son. This means that we must bring the page p containing this information. Then

- 1a) if the $\$$ -son s of N is changed into an a -son, s is removed from the list L_s in the main store and attached as an a -son to N . As we already brought in the page containing the list of references to sons of N we try to store s on this page, if possible. If this is not possible we must bring a page with free space and write s on it.
- 1b) if the $\$$ -son s of N is transferred to the a -son of N which is a not a leaf we just look on the page p for the reference to this a -son and modify the reference to father of s . This step does not cause any additional page transport.
- 1c) if the $\$$ -son s of N is transferred to the a -son of N which is a leaf, we proceed as in 1b. In addition we must read one of the pages on which the text array is situated.

As one may easily see, 1a) causes 1 page transfer, 1b) causes 1 page transfer, and 1c) causes 2 page transfers, again disregarding the cost of writing a new node and assuming that all son references of a node can be stored on one page. Only in 1a) we have to write on a page. The pages which are transferred for 1b) and 1c) are read only. Compared with the previous solution we have gained by reducing the number of page transfers and by reducing the number of write accesses.

6. Correction Algorithm

The substitution of a substring within a main string is a common operation in string manipulation systems. Hence, we are interested in a simple algorithm that performs this substitution and manipulates the position tree in the corresponding way. A correction algorithm for suffix trees has been given in [McC]. This algorithm is rather complicated and presupposes a link structure which is constructed while the tree is built up by the tree construction algorithm [McC]. In the following we are going to describe an algorithm which constructs the position tree for

xyr\$ from that for x8r\$, x.y,8,rcl*, 8#e. Before we present the algorithm we have to say some words about position numbers. If the lengths of γ and β are different then not only are position identifiers invalidated but also the position numbers for the text that follows β are affected. To circumvent the need to re-number the text that follows β we will assume in the following that our string positions are numbered in a Dewey-Decimal Scheme as follows: let a string

a b b c b
1 2 3 4 5

with annotated position numbers be given. Let us substitute bb by efg. The result is

a e f g c d
1 2 3.1 3.2 4 5

A substitution of f by hl yields

a e h l g c d
1 2 3.1.1 3.1.2 3.2 4 5

Substituting c by the empty word yields

a e h l g d
1 2 3.1.1 3.1.2 3.2 5

The position number scheme corresponds to the following string scheme for the text array.

a b b c d is stored as
1 2 3 4 5

a e f g c d is stored as
1 2 3.1 3.2 4 5

Next a e h l g c d is given in
1 2 3.1.1 3.1.2 3.2 4 5

and finally a e h l g d is given in
1 2 3.1.1 3.1.2 3.2 5

In the following we describe an algorithm which takes the string $x\beta r\$,$ $x, \beta, r \in \Sigma^*, \beta \neq \epsilon,$ ¹ the position tree for $x\beta r\$,$ and the string $y \in \Sigma^*$ as input and constructs the position tree for $x\gamma r\$,$ For simplicity we will first assume that the position numbers for the input string are the natural numbers $1, 2, \dots, |x\beta r\$|.$ The position numbers of the corrected string will be Dewey-Decimal numbers. We denote the sequence of Dewey-Decimal numbers for $x\gamma r\$,$ by $d_1, d_2, \dots, d_{|x\gamma r\$|}.$

The algorithm makes use of a marking operation. This operation will be restricted to nodes that are different from the root. In addition we will use the following notation:

N_i = the node that carries position number i

h_i = height of N_i $h(N)$ = height of N

$|z| = m, |y| = l, |x| = n$

Let us briefly explain how the algorithm works. The algorithm performs seven steps. In step 0 the left-most position $i,$ if any, is determined for which $1 \leq i \leq n$ and for which the position identifier "ends" in $\beta.$ E.g. in the string $abcbab\$,$ with

¹ The restriction is only introduced to avoid a large number of subcases in the algorithm. The restriction is not harmful as the transition from $x_1 \dots x_n r\$,$ to $x_1 \dots x_n \gamma r\$,$ can be always treated by considering the substitution of x_n by $x_n \gamma$ or by taking a right context if $n = 0.$

$x = dab$, $\beta = c$, $r = ab$, the position identifier for position 2 is abc , i.e. it "ends" in β . In the string $dabc\$$ with $x = dab$, $\beta = c$, $r = c$ there is no position i , $1 \leq i \leq n$, for which the position identifier "ends" in β . We let i_0 be the left-most position in x for which the position identifier "ends" in β , if any, else i_0 is set to $n + 1$. i_0 will be used in step III. In step Ia all leaves carrying a position number j , $n + 1 \leq j \leq n + m$, are removed. These position numbers belong to β . After the removal of these leaves it may be that a node N lost all sons. This situation occurs if two or more positions in β have a common prefix that does not occur outside β . The path from such a node N to its youngest ancestor that has more than one son has to be removed. This is performed in Ib.

As a result of Ia and Ib it may be the case that a father is left with one son and this son is a leaf with position number i . This situation occurs if the position i and some positions in β had a common prefix but the prefix did not occur at another position outside β . Then the path to the leaf i has to be shortened. This is done in II. In step III we treat all positions i , $i_0 \leq i \leq n$. The path to the node with position number i is shortened by pushing the position number i stepwise upwards. Whenever the node, from which the position number i is being passed to his father, is a leaf, this leaf is removed. We stop moving the position number i upwards if the current node has height $n - i + 1$. This corresponds to cut off that suffix of the identifier for i that is a prefix of β . At the same time, however, we take care of all positions k , with $k < i_0$ or $k > n + m$. This is achieved by marking a node if it has exactly one son left and this is a leaf with position number l . If $i_0 \leq l \leq n$ the position l will be processed anyway by III. Otherwise, in step IV the path leading to the leaf l is shortened until the point is reached where a prefix common to l and another position is encountered. In step V the position numbers for γ are attached to the root. In step VI the position identifiers for γ are built up; simultaneously a path to a node with position number l , $1 \leq l \leq n$ or $l > n + m$ is prolonged,

if necessary, i.e. the position number is pushed downwards.

Correction Algorithm

0) $i := n$; while $(i + h_1 > n + 1) \wedge i \geq 1$ do $\lceil i := i - 1 \rceil$; $i_0 := i + 1$

1a) For $i = 1, \dots, n$ do

$\lceil$ remove the leaf with position number $n + i$; mark the father if there is exactly one son left and this is a leaf $\rfloor$

1b) Choose a marked node without sons;

remove node;

if father has no more sons left, mark father;

if father has only one son left and this is a leaf, mark father;

if there are marked nodes without sons goto 1b;

II) Choose a marked node R ; co R has exactly one son and this is a leaf co

remove mark;

give position number of the son of R to R ;

remove son;

if R is the only son, mark its father;

if there are marked nodes left goto II.

III) For $i = n, n-1, \dots, i_0$ do

$\lceil$ while $i + h_1 > n + 1$ do

$\lceil$ give position number i to father of $N = N_1$ and remove it at N ;

co now father = N_1 co

if N is a leaf then

$\lceil$ remove N ;

if N_1 has exactly one son left and this is a leaf mark N_1 ;

if N_1 is a leaf and the only son mark father of N_1 ; $\rfloor$

if N_1 is a leaf remove mark---if any---else underline N_1 . $\rfloor$

IV) Choose a marked and not underlined node R;

remove mark;

give R the position number of its son;

remove son;

if R is the only son, mark its father;

if there is a marked and not underlined node left goto IV)

V) If $l \leq m$ give position number $n + 1, \dots, n + l$ to the root else give position number $n + 1, \dots, n + m - 1, (n+m).1, (n+m).2, \dots, (n+m).l - m + 1$ to the root; underline root co Dewey-Decimal numbers used co

VI) For each underlined node N do

┌ For each position number i at N do

┌ remove position number i from N;

follow path beginning at node N as long as the word of labels from the root to the current node coincides with a substring beginning at the position numbered i in $x_{yr}S$.

Let N_{last} be the last visited node.

If N_{last} is not a leaf then

┌ attach a new son to N_{last} ; give the new son the position number i; label the edge by the $h(N_{last})$ -th letter following the position i in $x_{yr}S$ ┘

If N_{last} is a leaf with position number j then

┌ remove position number j from N_{last} ; let a be the $h(N_{last})$ -th letter following position j in $x_{yr}S$; let a' be the $h(N_{last})$ -th letter following position i in $x_{yr}S$; while $a \neq a'$ do

┌ attach a new son NS to N_{last} ; label the edge by $a (= a')$ and set $N_{last} := NS$ ┘

attach two new sons NS_1 and NS_2 to N_{last} ; label the edge to NS_1 by a and give NS_1 the position number j; label the edge to NS_2 by a' and give NS_2 the position number i ┘

remove underlining and marks, if any, at N ┘

Let us illustrate the algorithm at the following example.

Example Consider the string a b c a b c \$ with position numbers 1, 2, ... 7. The position tree is

Let $x = ab$, $\beta = c$, $r = abc$, $\gamma = \epsilon$. Step 0 yields $i_0 = 1$. Let us now perform step I of the algorithm. Hence we remove the leaf with position number 3 and mark its father as there is only one son left. There are no marked nodes without sons hence we can skip Ib and get as result of step I

Performing step II yields

Starting step III with $n = 2$ ($n = |x|$) and step IV yield the tree

Step V is not performed as $t = 0$. Step VI yields

for the string $xRr\$ = a\ b\ a\ b\ c\ \$$ with annotated position numbers which are
 1 2 4 5 6

interpreted as Dewey-Decimal numbers.

In order to prove the correction algorithm correct we need some auxiliary definitions:

A *reduced identifier* for position i in $xRr\$$, $i \notin \{n+1, \dots, n+m\}$ is a prefix p_i of the position identifier such that p_i does not occur at another position j , $j \neq i$, $j \notin \{n+1, \dots, n+m\}$ in $xRr\$$.

Clearly, the position identifier for position i , $i \notin \{n+1, \dots, n+m\}$ is a reduced identifier.

A *partial identifier* for i , $1 \leq i \leq n$, in $xRr\$$ is a word p_i such that

- I) p_i is a prefix of $x_1 \dots x_n$
- II) p_i is a prefix of the position identifier for i in $xRr\$$
- III) p_i does not occur at j , $j \neq i$, $j \notin \{n+1, \dots, n+m\}$ in $xRr\$$

If there is such a p_i then the shortest word fulfilling I, II, III is called the shortest partial identifier else $x_1 \dots x_n$ is called the shortest partial identifier.

A partial identifier for j , $n+m+1 \leq j \leq n+m+|r|$, in $x\&r\&$, is a prefix p_j of the position identifier such that

- I) p_j does not occur at position k , $k \geq n+m+1, k \neq j$
- II) if p_j occurs at $k \leq n$ then

$$|p_j| > | \text{shortest partial identifier for } k |$$

The above declared notions correspond to the different steps of the algorithm, which is shown in the following.

Lemma 3. After step Ia of the algorithm the following holds:

- i) There may be leaves without position number
- ii) A leaf is without position number if and only if it is marked
- iii) If a node is marked then it is either a leaf and has no position number or it has exactly one son. This son is a leaf. If a node N has a single son and this is a non-marked leaf then N is marked.

Proof

- i) If a node has only sons which are leaves with a position number $i \in \{n+1, \dots, n+m\}$ then by step Ia this node becomes a leaf without position number.
- ii) If a leaf L has no position number then it was the father of leaves with a position number $i, n+1 \leq i \leq n+m$. The node L had at least 2 sons before step Ia, as the input of the algorithm is a position tree. After the removal of the next to last son of L , L has been marked by the algorithm. If a leaf is marked after Ia then it must have been an inner node before Ia and did not have any position number.
- iii) Let the node N be marked as a result of Ia then it was at some step the father of a leaf L which was the only son. In the sequel either this leaf is removed and N becomes a leaf without position number and is marked or the leaf remains there. If N has a single son which is a leaf and unmarked

then N must have had another son, as the input of the algorithm is a position tree. After the removal of this son N has been marked.

Lemma 4. After step I of the algorithm the following holds:

- i) Every leaf has a position number
- ii) A node is still marked iff it has exactly one son left which is a leaf.
- iii) Each path from the root to a leaf corresponds to a position identifier. For each position $i \notin \{n+1, \dots, n+m\}$ the position identifier is still in the tree.

Proof.

- i) This is obvious as every leaf without position number has been marked (Lemma 3). The marked leaves are removed by Ib.
- ii) Let a node be marked as a result of I then it must have a son otherwise it would have been removed by Ib. Moreover only nodes are marked which have a single son that is a leaf. Let a node be given that has a single son that is a leaf. Then by i) this leaf has a position number. Hence N must have had at least another son before I has been performed. At the removal of the next to last son N has been marked.
- iii) Obvious, as only leaves with position number $i \in \{n+1, \dots, n+m\}$ are affected by step I.

Remark. In step I of the algorithm all and only those nodes $\neq$ root are removed which 1) lie on a path from the root to a leaf with position number $i, i \in \{n+1, \dots, n+m\}$ and 2) do not lie on a path from the root to a leaf with position number $i \notin \{n+1, \dots, n+m\}$.

Lemma 5. After step II of the algorithm

- i) Each leaf has a position number
- ii) There are no marks left
- iii) Each path from the root to a leaf corresponds to the shortest reduced identifier.

Proof.

- i) Obvious by Lemma 4
- ii) Obvious
- iii) By Lemma 4, the position identifier is still contained in the tree before II. We may distinguish two cases. Either the leaf is a single son then its father is marked and will be processed by step II. The path is shortened as long as the respective father becomes a single son. After this the path reflects the shortest reduced identifier, as any shorter prefix would occur at least at one other position. In the other case the position identifier cannot be reduced any further.

Lemma 6.

- i) If after step III a node is underlined then it has at least one son.
- ii) After step III
 - iii) Each path from the root to a non-leaf node with position number corresponds to a shortest partial identifier.
 - ii2) Each path from the root to a leaf corresponds to a partial identifier.

Proof.

- i) Obvious, since nodes which are leaves are not underlined
- iii) Let p_i be the word of labels from the root to the inner node with position number i then $i \leq n$ and p_i occurs at least at another position; hence p_i is not a partial identifier. But $p_i = x_1 \dots x_n$ as step II is performed until $i + h_i = n + 1$, hence p_i is shortest partial identifier by definition.

ii2) Let the word of labels from the root to a leaf n_i with position number i be to p_i . If $i \leq n$ then $h_i \leq n+1-i$ and p_i is a prefix of the position identifier for position i in $x\$r\$$ and p_i is a prefix of $x_1 \dots x_n$. Moreover, n_i is a leaf and hence p_i does not occur at another position j , $j \notin \{n+1, \dots, n+m\}$. Hence, p_i is partial identifier. If $i > n+m$, then the path from the root to n_i yields the shortest reduced identifier which is a partial identifier by definition.

Remark. After step IV for each $i \notin \{n+1, \dots, n+m\}$ the path from the root to the node with position number i corresponds to the shortest partial identifier. This is clear, as an identifier is shortened in this step until a node is reached which has more than one son.

Finally, in step V the new position numbers are attached at the root and in step VI the position identifiers are built up.

In order to be able to perform the above algorithm we need the following information per each node

- 1) A list of sons
- 2) The position identifiers associated with a node
- 3) The height of the node
- 4) The father of a node

In addition we have to maintain a list of underlined nodes and a list of marked nodes. Moreover, in order to guarantee random access to the leaves of the tree for step Ia, we store--in the array containing the text--for each position number a reference to the leaf with position number i .

Based on this representation the cost of step 0) is $O(n-i_0 + 2)$ where

$$i_0 = \begin{cases} \min \{i: 1 \leq i \leq n \text{ and pos. identifier for } i \text{ is longer than} \\ \quad n - i + 1\} & \text{if this set is not empty} \\ n + 1 & \text{else} \end{cases}$$

Step Ia costs $O(m)$ steps, where m is the length of S .

Steps Ib and II can only be performed at nodes that lie on the path from a leaf with position number $n + i$, $1 \leq i \leq m$, to the root (in the tree for $x\&r\$$). The number of nodes on the path from the leaf with position number j to the root (excluding the root) is given by $|p_j(x\&r\$)|$, where $p_j(x\&r\$)$ is the position identifier for position j in $x\&r\$$. Hence the total number of nodes which will be affected by steps Ib, II can be bounded above by

$$\sum_{i=1}^m |p_{n+i}(x\&r\$)|$$

Each node is processed at most once and the operations for each node cost constant time, hence steps Ib, II cost

$$O\left(\sum_{i=1}^m |p_{n+i}(x\&r\$)|\right)$$

Let T^{II} be the tree resulting from step II. Step III can affect only such nodes in the tree T^{II} that lie on a path from a leaf that has a position number i , $i_0 \leq i \leq n$, and fulfills $h_i + i > n + 1$ to the ancestor A_i of that leaf for which

$$h(A_i) + i = n + 1$$

holds. The path in T^{II} from the root to the leaf i contains (without root) at most $|p_i(x\&r\$)|$ nodes, hence the path from the ancestor A_i to the leaf i contains at most

$$\begin{aligned} & |p_i(x\&r\$)| - h(A_i) \\ &= |p_i(x\&r\$)| - (n + 1 - i) \end{aligned}$$

nodes. As the loop in step III manipulates the nodes with position numbers $i_0, i_0 + 1, \dots, n$ the total number of nodes affected by step III can be bounded above by

$$\sum_{i=i_0}^n (|p_i(x\&r\$)| - (n + 1 - i)).$$

A lesser upper bound for the number of nodes affected by step III can be given in terms of the shortest reduced identifier. Let shr_i be the shortest reduced identifier for position i in $x\text{Er}$. As a result of step III every path from the root to a leaf i corresponds to the shr_i . Hence, the total number of nodes affected by step III is

$$\sum_{i=i_0}^n \max(0, |\text{shr}_i| - (n + 1 - i))$$

Here, we have to use $\max(0, |\text{shr}_i| - (n + 1 - i))$ because $|\text{shr}_i|$ may be less than $n + 1 - i$. If this is the case for leaf i then step III will not be performed for the position number i .

The cost for manipulating a node is constant, hence the total cost of III can be given by

$$\begin{aligned} & O\left(\sum_{i=i_0}^n \max(0, |\text{shr}_i| - (n + 1 - i))\right) \leq \\ & \leq O\left(\sum_{i=i_0}^n (|p_i(x\text{Er})| - (n + 1 - i))\right) \end{aligned}$$

As a result of III for each position number $i, i_0 \leq i \leq n$, there is at most one marked node left the marking of which was caused by i ; if in T^{III} the node N_i carrying the position number i is a leaf (T^{III} is the result of III) then the father of N_i may be marked. If N_i is not a leaf and hence underlined then it may be marked. If N_i is not a leaf and hence underlined then it may be that there is a successor of N_i which was marked as the leaf with position number i was removed. If the position number i was not processed by III because $i + h_i \leq n + 1$ then i did not cause any marking. If N_i is a leaf as a result of III, at most the nodes on the path between the root and N_i can be affected by step IV. If N_i is not a leaf then at most the nodes between N_i and the possibly existing marked successor can be affected by step IV. Hence we can bound the total number of nodes affected

by step IV

$$\sum_{i \in I_1} (n - i + 1) + \sum_{i \in I_2} (|shp_i| - (n + 1 - i))$$

where shp_i is the shortest partial identifier for i in $x\bar{s}r\$$ and

$$I_1 = \{i: i_0 \leq i \leq n \text{ and } i + h_i > n + 1 \text{ in } T^{II} \text{ and } N_i \text{ is leaf in } T^{III}\}$$

$$I_2 = \{i: i_0 \leq i \leq n \text{ and } i + h_i > n + 1 \text{ in } T^{II} \text{ and } N_i \text{ is not a leaf in } T^{III}\}$$

where T^{II} is the tree resulting from step II and T^{III} is the result of step III.

Step (V) costs $O(i)$ time.

For step VI we note that an underlined node is either the root and carries i position numbers or it is an inner node and carries exactly one position number.

The total cost of step VI can be bounded above by

$$O\left(\sum_{d \in \{d_{n+1}, \dots, d_{n+l}\}} |p_d(xyr)| + \sum_{i=i_0}^n (|p_i(xyr)| - |shp_i|)\right)$$

where shp_i is the shortest partial identifier for i in $x\bar{s}r\$$, $p_i(xyr)$ is the position identifier for position i in $xyr\$$ and $d_{n+1}, \dots, d_{n+l}$ are the (Dewey-Decimal) position numbers for the $(n+1), \dots, (n+l)$ -th letters in $xyr\$$. This can be seen by the fact that the inner loop of VI executes for $d \in d_{n+1}, \dots, d_{n+l}$ as many steps as the length of $p_d(xyr)$. In the case of $i \in \{i_0, \dots, n\}$ the shortest partial identifier for position i is prolonged to become $p_i(xyr)$.

The above algorithm and its cost considerations work under the condition that the position numbers of the input string $x\bar{s}r\$$ are the natural numbers $1, 2, \dots, |x\bar{s}r\$|$. However, as we may want to apply the correction algorithm repeatedly we have to give some consideration to the general case where the input may be numbered by Dewey-Decimal numbers. There is one obvious approach: after an application of the correction algorithm all positions that are situated to the right of the correction position n are renumbered resulting in position numbers $1, 2, \dots, |x\bar{s}r\$|$. This solu-

tion corresponds to shifting text in the text array.

If there is no restriction as to when corrections can be made this solution will not be feasible in general. Hence, we have to find out which changes have to be made to our correction algorithm in order to allow for input with Dewey-Decimal position numbers. Let

$$d_1, d_2, \dots, d_{|xsr\$|}$$

be the Dewey-Decimal position numbers for $xsr\$$. Let $|x| = n$, $|s| = m > 0$ as before. We call the text array with pointers to substrings (as explained in the beginning of chapter VI) an extended array. Let the position number of the first letter of s be d^s , i.e. $d^s = d_{n+m}$.

In general we will have to deal with the case that the input of the algorithm consists of the position number d^s , the strings s , γ , the position tree for $xsr\$$ and the extended text array for $xsr\$$. The size of n , i.e. the length of the text to the left of s will not be given as input in general (even though it could be determined by traversing the extended text array).

We will use the following notations and operations. Let

$$d_1, \dots, d_{|w|}$$

be the Dewey-Decimal position numbers for the text w . We define

$$\begin{aligned} \text{pred}(d_i) &= d_{i-1} & \text{for } i > 1 \\ \text{succ}(d_i) &= d_{i+1} & \text{for } i < |w| \end{aligned}$$

d_i is the position number of the i -th letter in w , $\text{pred}(d)$ determines the position number of the position left of d , $\text{succ}(d)$ determines the position number of the position right of d .

Moreover we define an addition between Dewey-Decimal numbers and natural numbers by

$$d_i + h := d_{i+h} \quad \text{if } i + h \leq |w|$$

The order between Dewey-Decimal numbers is the lexicographical order.

Let us now consider the position tree for $x_2r\$$ with position numbers $d_1, \dots, d_{|x_2r\$|}$. As before, let N_d be the node carrying the position number d and let $h(d)$ be the height of N_d . We consider the modifications which have to be made to our original correction algorithm in order to allow for Dewey-Decimal position numbers for the input.

One can immediately see that steps Ib, II, IV, VI of the correction algorithm can remain unchanged if we want to use Dewey-Decimal numbers for the input.

Steps O, Ia, III and V have to be substituted by

O') If $d^B > d_1$ then

$\lceil d := \text{pred}(d^B);$

 while $d + h(d) > d^B \wedge d > d_1$ do

$\lceil d := \text{pred}(d);$

 If $d = d_1 \wedge d + h(d) > d^B$ then $d^O := d$

 else $d^O := \text{succ}(d)$ $\rfloor$

Ia') $d := d^B;$

For $i = 1, \dots, m$ do

$\lceil$ remove leaf with position number d ; mark father if there is exactly one son left and this is a leaf; $d := \text{succ}(d)$ $\rfloor$

III') If $d^2 > d_1$ then

$\lceil d := \text{pred}(d^2);$

 while $d \geq d^0$ do

$\lceil$ while $d + h(d) > d^2$ do

$\lceil$ give position number d to father of $N = N_d$ and remove it at N ;

co now father = N_d co

 if N is a leaf then

$\lceil$ remove N ;

 if N_d has exactly one son left and this is a leaf then mark N_d ;

 if N_d is a leaf and the only son then mark father of N_d ;

 if N_d is a leaf then remove mark--if any--else underline N_d ;

 if $d = d_1$ then exist else $d := \text{pred}(d)$;

VI') If $1 \leq m$ give the position numbers $d_{n+1}, \dots, d_{n+m}$ to the root else give the position numbers $d_{n+1}, \dots, d_{n+m-1}, d_{n+m-1}, d_{n+m-2}, \dots, d_{n+m} \cdot (k-m+1)$ to the root.

As one can see, the correction algorithm using Dewey-Decimal numbers for the input can be obtained by minor modifications of the original algorithm. It should be noted, however, that this new algorithm is more expensive. This can be seen from the following observations: i) given a position number d , there is no immediate access to the next smaller or next greater position number. We have to find these position numbers in the extended array. This costs at most as many steps as there are levels of references to substrings in the extended array. ii) in order to calculate $d + h(d)$ for steps 0') and III') we have to find the $h(d)$ -th position number following d . iii) to determine the $h(N_{\text{last}})$ -th letter following position d in step VI) we have to traverse parts of the extended array.

The costs for ii) and iii) can be reduced if we maintain information about the length of the substrings that are pointed at from the main array. Work is in progress that investigates some time-space tradeoffs of the above sketched solution and investigates garbage collection problems caused by replacing β by γ with $\gamma_1 \in \{1\}$.

Let us finally consider the correction problem in compacted trees. For compacted trees there is a simpler solution to the correction task which will be explained in the following. We assume for simplicity that the position numbers of the input $x\hat{s}r\hat{s}$ are the natural numbers $1, 2, \dots, |x\hat{s}r\hat{s}|$. The algorithm takes as input the compacted position tree for $x\hat{s}r\hat{s}$. If a node N in this tree has a single son, then N is marked.

For each position number i , let dh_i be the length of the word of labels on the path from the root to the node carrying the position number i . As before $|x| = n$, $|s| = m > 0$ and $|y| = l$. In the algorithm we make use of an auxiliary symbol $\perp \notin E \cup \{\hat{s}\}$. The position numbers for $x\hat{s}r\hat{s}$ are $d_1, \dots, d_{|x\hat{s}r\hat{s}|}$. We write $x\hat{s}r\hat{s} = a_1 a_2 \dots a_n a_{n+1} \dots a_{n+l} \dots a_{|x\hat{s}r\hat{s}|}$.

Correction in compacted position trees

```

0)  $i := n$ ;
   while  $i + dh_i > n + 1 \wedge i \geq 1$  do  $i := i - 1$ ;
    $i_0 := i + 1$ 
1)  $i := i_0$ ;
   while  $i \leq n + m$  do
     [remove the leaf with position number  $i$ ;
      if the father  $F$  has exactly one son  $L$  left and this is not a leaf then
        if there are two adjacent marked nodes, then compactify]
      if the father  $F$  has exactly one son  $L$  left and this is a leaf then
        [give  $F$  the position number  $j$  of  $L$ ;
         remove  $L$ ;
         if  $F$  is the single son of  $E$  then
           [give  $E$  the position number  $j$ ;
            remove  $F$ ; remove mark at  $E$ ]
         else replace the word of labels  $c_1 \dots c_n$  on the edge from  $E$  to  $R$  by  $c_1$ ];
      $i := i + 1$ 

```

- II) attach a 1-son to the root with position number d_{i_0} ; remove mark at root, if any
- IIIa) For $i = i_0, i_0 + 1, \dots, n + 1$ do
- [let a be the i -th letter in $xyr\$$;
 - for each node N that has a 1-son N' do;
 - (The order in which the nodes are processed is that given in Lemma 1)
 - [if N does not have an ar-son, reE^* , then change the label of the edge (N, N') from 1 to a ;
 - if N has an ar-son N'' , reE^* , then
 - [remove the edge (N, N') ;
 - mark N , if there is exactly one son left;
 - if $r \neq \epsilon$ then
 - [create a new NN ;
 - make the sons of N'' the sons of NN ;
 - make NN the r -son of N'' ;
 - if N'' was marked transfer the mark to NN ;
 - change the label of (N, N'') from ar to a];
 - if $r = \epsilon$ and N'' is not a leaf then remove mark at N'' , if any;
 - if $r = \epsilon$ and N'' is a leaf then
 - [attach a new son to N'' ; transfer the position number j of N'' to the new son; label the edge between N'' and the new son by the ℓ -th letter following position j in $a_1 \dots a_{i-1} r \$$ (where ℓ is the length of the position identifier for position j in $a_1 \dots a_{i-1} r \$$)];
 - attach N' as a 1-son to N'' together with the position number];
 - attach a 1-son at the root; give it position number d_{i+1} ; remove mark at root, if any;
 - for all pairs of adjacent marked nodes: compactify]

IIIb) remove 1-son of root

IV) $i := n + l + 1$:

while there are nodes with 1-son do

┌ let a be the i -th letter in $x\gamma r$;

for each node that has a 1-son do: (order given in Lemma 1)

┌ as in step III a);

for all pairs of adjacent marked nodes;

compactify;

$i := i + 1$ ┘

The algorithm basically works as follows: In step 0 the left most position i_0 is determined by the position identifier of which "ends" in β . In step I the position identifiers for all positions i , $i_0 \leq i \leq n + m$ are removed from the tree. The tree is now a compacted position tree for $x_1 \dots x_{i_0-1} r$. In step II we attach an auxiliary 1 son with position number d_{i_0} to the root; this corresponds to work with the string $x_1 \dots x_{i_0-1} r$. In step III we use the on-line construction algorithm for compacted position trees for $a_1 \dots a_{i_0} r$, ..., $a_1 \dots a_{n+l} r$. In step IV we continue to use the on-line algorithm until the 1 does not occur any more as part of any position identifier. In contrast to step III we do not introduce any new position numbers in step IV, as the position number for the positions right of γ are already in the tree.

The cost of the algorithm are as follows: let $x^* = x_1 \dots x_n$ and let $r^* = r_1 \dots r_j$ where $r_1 \dots r_j$ is the longest prefix of r that occurs twice in $x\gamma r$. Step 0) costs $O(|x^*|)$. Step 1) costs $O(|x^*| + |\beta|)$, because the number of iterations is $|x^*| + |\beta|$ and each iteration costs constant time. Step IIIa) basically applies the on-line construction algorithm¹ for compacted trees $(|x^*| + |\beta|)$ -times, step IV applies the on-line construction algorithm¹ $|r^*|$ -times.

¹One application of the on-line construction algorithm costs $O(|S_i| + 1)$, where the list of nodes with a 1-son is $(N_i)_{i \in I}$ and S_i is the set of sons of N_i .

REFERENCES

- [AHU] A. Aho, J. Hopcroft, J. Ullmann: The design and analysis of computer algorithms.
- [CC] R. Cohen, M. Cimet: A scheme for constructing on-line linear time recognition algorithms. Conference on theoretical computer science, Waterloo, p. 70-80, (1977)
- [KMP] D.E. Knuth, J.H. Morris, V.R. Pratt: Fast pattern matching in strings. SIAM Journal of Computing, vol. 6, no. 2, June, 1977.
- [McC] E.M. McCreight: A space-economical suffix tree construction algorithm. Journal of the ACM, Vol. 23, No. 2, April 1976.
- [W] P. Weiner: Linear pattern matching algorithms. Conf. Record, IEEE 14th Annual Symposium on Switching and Automata Theory, p. 1-11, (1973).