Algorithmica (2003) 37: 43–74
DOI: 10.1007/s00453-003-1029-2

Algorithmica
© 2003 Springer-Verlag New York Inc.

Linear Bidirectional On-Line Construction of Affix Trees

Moritz G. Maaß[1]

Abstract. Affix trees are a generalization of suffix trees that are based on the inherent duality of suffix trees induced by the suffix links. An algorithm is presented that constructs affix trees on-line by expanding the underlying string in both directions and that is the first known algorithm to do this with linear time complexity.

Key Words. Algorithms and data structures, Pattern matching, Affix tree, Suffix tree, Amortized analysis.

1. Introduction. Suffix trees are a widely studied and well known data structure. They can be applied to a variety of text and pattern matching problems and have become more popular with their application in problems related to molecular biology. A suffix tree is in essence an index structure for a string (or text) t. Besides the ability to answer the question whether a string w occurs in t in time $\mathcal{O}(|w|)$, it reveals many properties of the string t like the longest repeated substring, the frequency of substrings, and others. There are at least three widely known algorithms that construct suffix trees in linear time with respect to the length of the underlying string. The first algorithm was due to Weiner [8], the second algorithm was developed by McCreight [4], and the last algorithm was developed and published by Ukkonen [7]. All these algorithms make use of auxiliary edges usually called suffix links. Weiner uses these edges in a reverse direction, which is one of the reasons his algorithm is slower and more space consuming than those of McCreight and Ukkonen.

Each node in a suffix tree represents a string. Passing along an edge to another node usually lengthens the represented string. Suffix links are used to move from one node to another so that the represented string is shortened at the front. This is extremely useful since two suffixes of a string always differ by the characters at the beginning of the longer suffix. Thus, successive traversal of suffixes can be sped up greatly by the use of suffix links.

Giegerich and Kurtz have shown in [3] that there is a strong relationship between the suffix tree built from the reverse string (often called the reverse prefix tree) and the suffix links of the suffix tree built from the original string. The suffix links of a suffix tree form a tree by themselves. Giegerich and Kurtz have shown that the suffix links of the atomic suffix tree (AST—sometimes also referred to as a suffix trie) for a string t are exactly the edges of the AST for the reverse string t^{-1}. This property is partially lost when turning to the construction of the compact suffix tree (CST—most often referred to simply as a suffix tree). The suffix links of the CST for a string t represent a subset of the strings of the CST for the reverse string, which we call the compact prefix tree (CPT).

[1] Fakultät für Informatik, TU München, Boltzmannstrasse 3, D-85748 Garching, Germany. maass@informatik.tu-muenchen.de.

Received October 16, 2002; revised February 27, 2003. Communicated by S. Albers.
Online publication May 28, 2003.

A similar relationship was already shown by Blumer et al. in [1] for compact directed acyclic word graphs (c-DAWG). As shown there, the DAWG contains all nodes of the CPT (where nodes for nested prefixes are also inserted, although they are not branching) and uses the CPT's edges as suffix links. The c-DAWG is the "edge compressed" DAWG. Because the nodes of the c-DAWG are invariant under reversal one only needs to append the suffix links as reverse edges to gain the symmetric version. Although not stated explicitly, Blumer et al. in essence already observed the dual structure of suffix trees as stated above because the c-DAWG is the same as a suffix tree where isomorphic subtrees are merged.

In this paper we describe how a tree that incorporates both the CST and the CPT can be built on-line in linear time by appending characters in any order at either side of the underlying string. This tree is called a compact affix tree (CAT). We assume a unit cost model and a constant size alphabet, the same assumptions under which the common suffix tree algorithms are linear. Our algorithm is then the first linear algorithm known. The CAT data structure has been introduced by Stoye in his Master's thesis [5] (an English translation is available [6]). Our algorithm uses additional data collected in paths to assure a suffix-tree-like view. The additional data shows that the resulting algorithm has linear amortized running time. The basic algorithmic idea is to merge an algorithm for building CSTs on-line (Ukkonen's algorithm) with an algorithm for building CSTs anti-on-line, appending to the front of the string in a fashion similar to Weiner's algorithm [8].

The core problem for achieving linear complexity is that Ukkonen's or any other suffix tree algorithm expects suffix links of inner nodes to be atomic, i.e., the links represent a single character. As can be easily seen in Figure 1 this is the case for suffix trees, but not for affix trees (nodes $\overline{\texttt{aba}}$ and $\overline{\texttt{abab}}$ in Figure 1(d) and nodes $\overline{\texttt{ba}}$ and $\overline{\texttt{cbab}}$ in Figure 1(e)). It seems that the problem cannot be solved by traversing the tree to find the appropriate atomic suffix links. Stoye gave an example where this approach might even lead to quadratic behavior (similar to Figure 2). We use additional data collected in paths to establish a view of the CAT as if it were the corresponding CST (the paths correspond to the boxed nodes in Figure 1).

Affix trees have applications where a bidirectional search is needed, i.e., the search pattern needs to be extended in both directions depending on previous extensions. An example of this is the search for inverted tandem repeats, which play a role in DNA analysis.

We continue by briefly describing the data structures in Section 2. Then the algorithms for constructing CSTs in linear time on-line by adding characters either always to the front or to the end of the string are given in Section 3. These techniques are merged to form the linear bidirectional on-line algorithm for CATs in Section 4. The linearity for bidirectional construction is proven in Section 5.

2. Definitions and Data Structures

2.1. *Strings.* Suffix and affix tree are structures defined over strings. In the following, let Σ be an arbitrary finite alphabet. Let Σ^* denote the set of all finite strings over Σ (including the empty string ε), let $\Sigma^+ = \Sigma^* \backslash \{\varepsilon\}$ denote the set of all nonempty strings over Σ. Throughout this paper we regard the size of the alphabet as constant.

Let $t = x_1 x_2 x_3 \cdots x_n$ be a string with characters $x_i \in \Sigma$, we define $|t| = n$ to be its length and $t^{-1} = x_n x_{n-1} x_{n-2} \cdots x_1$ to be the reverse string. For the strings $u, v \in \Sigma^*$ we define uv as their concatenation. For $u, v, w \in \Sigma^*$ and $t = uvw$, u is a *prefix*, v is a *substring*, and w is a *suffix* of t. A suffix w (or a prefix v) is called *proper* if $w \neq t$ ($v \neq t$, respectively). We use the letters u, v, w, x, y for strings and the letters a, b, c, d, e for characters.

We will need to work with substrings appearing at different positions in a string t. To reduce the notational overhead we will use the notation introduced in the following example (see also [5]). Let $t = uvaw$ and $t = u'vbw'$ for a substring $v \in \Sigma^+$ of t, characters $a, b \in \Sigma$ and arbitrary strings $u, w, u', w' \in \Sigma^*$. If we are only interested in the substring v followed by an a in the first occurrence and a b in the second occurrence, we will write $t = \{\ \_\_\_\_va\_\_, \_\_vb\_\_\_\_\ \}$, which emphasizes that there are at least two different occurrences of v in t, one followed by an a another by a b.

A suffix w of t is called *nested* if there exists a $v \in \Sigma^+$ such that $t = uwv$, i.e., the suffix w occurs at another position in t, $t = \{\ \_\_w\_\_, \_\_\_\_w\ \}$. Similarly, a prefix w of t is called *nested* if there exists a $u \in \Sigma^+$ such that $t = uwv$.

A substring w of t is called *right branching* if $a, b \in \Sigma$ are two distinct characters ($a \neq b$) and there are two occurrences of w, one followed by a and one by b, i.e., $t = uwav$ and $t = u'wbv'$ for any $u, u', v, v' \in \Sigma^*$ or—in the notation from above—$t = \{\ \_\_\_\_wa\_\_, \_\_wb\_\_\_\_\ \}$. A substring w of t is called *left branching* if $a, b \in \Sigma$ are two distinct characters ($a \neq b$) and there are two occurrences of w, one preceded by a and one by b ($t = uawv$ and $t = u'bwv'$ for any $u, u', v, v' \in \Sigma^*$: $t = \{\ \_\_\_\_aw\_\_, \_\_bw\_\_\_\_\ \}$).

The following observation is easily adapted to left branching and prefixes.

OBSERVATION 1 (The Right Branching Property Is Inherited by Suffixes). *If w is right branching in t, then all suffixes of w are also right branching in t.*

2.2. *Trees*. We define the affix tree data structure in terms of [3] as trees over strings, beginning with Σ^+-trees.

DEFINITION 2 (Σ^+-Tree). A Σ^+-tree T is a rooted, directed tree with edge labels from Σ^+. For each $a \in \Sigma$, every node in T has at most one outgoing edge whose label starts with a.

A Σ^+-tree is called *atomic* if all edges are labeled with single characters. A Σ^+-tree is called *compact* if all nodes other than the root are either leaves or branching nodes.

We assume a uniform cost model where operations on pointers and integers have constant cost and the size of pointers and integers are regarded as constant. These assumptions are common in pattern matching and reflect the situation on modern computers where all numbers fit into the computer's word size.

The edges can be implemented in constant space (assuming the uniform cost model) if the edge labels are represented by means of pointers into an underlying string. For example, for the string $t = x_0 \cdots x_n$, the edge label $x_i \cdots x_{i+k}$ will be represented by two indices i and $i+k+1$ (the last index is exclusive, so that the length of the string can be calculated by the difference of the indices). An edge is said to be *atomic* if its label has length 1.

For a node n let $w \in \Sigma^*$ be the string that is constructed by concatenating all edge labels on the path from the root to the node n. We define $\texttt{path}(n) = w$. Each such path in T is uniquely represented by a string because the outgoing edges of every node start with different characters (Definition 2). If $u = \texttt{path}(n)$, we can identify the node n by the string $u \in \Sigma^*$ and we will write $\bar{u} = n$.

Let $w = \texttt{path}(n)$, then the *depth* of n is $\texttt{depth}(n) = |\texttt{path}(n)|$ (sometimes also called the string depth).

Each Σ^+-tree T defines a *word set* that is represented by the tree. A string $u \in \Sigma^*$ is in $\texttt{words}(T)$ iff there is a node n in tree T with $\texttt{path}(n) = uv$ for $v \in \Sigma^*$ (i.e., u is a prefix of the string represented by a node in T).

If a string u is represented by a node n in T (i.e., $u = \texttt{path}(n)$), we call the location of the string *explicit* (the node $\bar{u}$ exists). Otherwise, if the string u is in $\texttt{words}(T)$, but there is no node n such that $\texttt{path}(n) = u$, then the location of the string is *implicit*. If a string u has an implicit location, there is a node m and a string $v \in \Sigma^+$ such that $u = \texttt{path}(m)v$ and there is an outgoing edge at m labeled with vw with $w \in \Sigma^+$ (i.e., there is no empty part w of the edge label which is not v). Node m is closest below the implicit location of u and we can represent the location of u by m and v (we will use this later in the algorithm). In other words, there are two nodes m and n such that $\texttt{path}(m)$ is a proper prefix of u and u is a proper prefix of $\texttt{path}(n)$. An explicit location is represented by a node, while an implicit location is "in" an edge.

We will add auxiliary edges to Σ^+-trees. Each such edge allows us to find quickly a node that represents a suffix of the string associated with the starting node of the edge. A *suffix link* is an auxiliary edge from node n to node m, with the property that $\texttt{path}(m)$ is a proper suffix of $\texttt{path}(n)$ and there exists no other node m' such that $\texttt{path}(m')$ is a proper suffix of $\texttt{path}(n)$ and $|\texttt{path}(m)| < |\texttt{path}(m')|$. For a suffix link from n to m let $\texttt{path}(n) = u\texttt{path}(m)$ with $u \in \Sigma^+$. The suffix link can be labeled similarly to regular edges with the string u.

We can augment all Σ^+-trees such that every node but the root has a suffix link (the empty string, represented by the root, is always a proper suffix). Since every node but the root has a suffix link, the suffix links form a tree by themselves.

DEFINITION 3 (Reverse Tree T^{-1}). The reverse tree T^{-1} of a Σ^+-tree T augmented with suffix links is defined as the tree that is formed by the suffix links of T, where the direction of each link and its label is reversed.

The intuition behind reversing the labels is that the labels are read along the direction of the edge character by character. Moving along the opposite direction will thus result in a reversed label. Figure 1(a) shows the CST for the string $\texttt{ababc}$ with the suffix links from $\overline{\texttt{ab}}$ to $\overline{\texttt{b}}$ to the root as it would have been constructed, e.g., by Ukkonen's algorithm. These links are labeled with $\texttt{a}$ and $\texttt{b}$. We added suffix links for the leaves, too, from $\overline{\texttt{ababc}}$ to $\overline{\texttt{babc}}$ to $\overline{\texttt{abc}}$ to $\overline{\texttt{bc}}$ to $\overline{\texttt{c}}$ to the root. These links are labeled with $\texttt{a}$, $\texttt{b}$, $\texttt{a}$, $\texttt{b}$, and $\texttt{c}$. Figure 1(b) shows the resulting reverse tree.

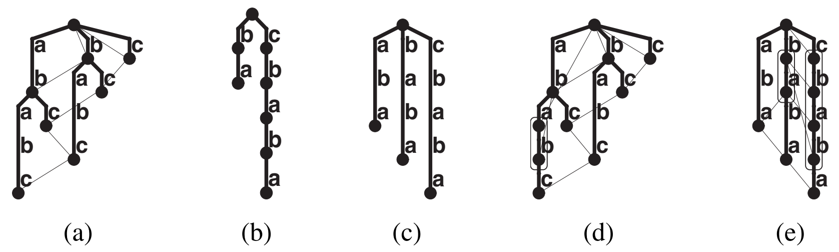


Fig. 1. Example trees based on the string `ababc`: (a) the CST for t = `ababc`, (b) the reverse tree of the CST for t = `ababc` (see Definition 3), (c) the CST for t = `cbaba`, (d) a view of the CAT for t = `ababc` focused on the suffix structure, and (e) the CAT focused on the prefix structure.

A suffix tree now can be easily defined as the Σ^+-tree that represents all suffixes of a string t.

DEFINITION 4 (Suffix Tree). A suffix tree of string t is a Σ^+-tree with

$$\texttt{words}(T) = \{u \mid u \text{ is a suffix of } t\}.$$

The definition $\texttt{words}(T) = \{u \mid u \text{ is a substring of } t\}$ is equivalent because for a given substring v of t there is a suffix vu of t represented in T by a node n with $\texttt{path}(n) = vuw = vw$. If the suffix tree is atomic, it is often referred to as a trie. Because of its importance the CST is often referred to only as a suffix tree.

Giegerich and Kurtz have proven in [3] a duality of reversing the string and taking the reverse tree. The AST T for the string t is the same as the reverse AST T^{-1} for the reverse string t^{-1}. This can be easily seen as follows: Let $t = x_1 \cdots x_n$. Let w be a substring of t. The string w is explicitly represented by node n in T with $w = \texttt{path}(n) = x_i x_{i+1} x_{i+2} \cdots x_j$. Because T is atomic, all edges have length 1 and all substrings of t have explicit locations. Therefore, there is also a node m with $\texttt{path}(m) = x_{i+1} x_{i+2} \cdots x_j$ and a suffix link labeled x_i links n to m. By induction we will thus find a chain of suffix links with labels $x_i, x_{i+1}, x_{i+2}, \ldots, x_j$. Thus the string $x_j x_{j-1} x_{j-2} \cdots x_i = w^{-1}$ is represented in T^{-1}. Therefore all reverse substrings are represented in T^{-1}.

For CSTs there is only a weak duality where the reverse T^{-1} of the CST for t represents a subset of the words of the CST for t^{-1} [3]. Since not all substrings are represented explicitly the above construction does not work. The reverse of all strings that are represented explicitly is also represented in the reverse tree, but some strings might not be represented.

The following lemmas give an intuition of the represented parts. These results are common knowledge and variations appear throughout the literature. We formalize the results suitable to our needs as follows.

LEMMA 5 (Explicit Locations in CSTs). *In the CST T for the string t, a substring u is represented by an inner node (its location is explicit) iff it is a right branching substring. It is represented by a leaf iff it is a non-nested suffix of t. Finally, the substring is represented by the root iff it is the empty string. There are no other nodes in T.*

PROOF. Clearly, the empty string is always represented by the root.

If $u \in \texttt{words}(T)$ is a non-nested suffix of t, there exists no $v \in \Sigma^+$, $w \in \Sigma^*$ such that $t = wuv$. If u were not represented by a leaf, there must be a node n such that $\texttt{path}(n) = uv'$ with $v' \in \Sigma^+$. This is a contradiction to u being non-nested. Therefore, u is represented by a leaf.

If $w \in \texttt{words}(T)$ is right branching in t, there exist $a, b \in \Sigma$, $a \neq b$, and $t = \{\ \_\_\_\_wa\_\_, \_\_wb\_\_\_\_\ \}$, where wa and wb are in $\texttt{words}(T)$. Hence, there are nodes n, n' and shortest strings $v, v' \in \Sigma^*$ such that $\texttt{path}(n) = wav$ and $\texttt{path}(n') = wbv'$. From the uniqueness of string locations in Σ^+-trees and from $wav \neq wbv'$ it follows that $n \neq n'$. Therefore there must be a branching node m such that m is an ancestor of n and n', and that $\texttt{path}(m)$ is a prefix of wbv' and wav. Therefore $\texttt{path}(m)$ must be a prefix of w. If $\texttt{path}(m) \neq w$, there would be two edges leading out of m with labels starting with the same character, which contradicts the T being a Σ^+-tree. As a result, $\texttt{path}(m) = w$ and w is represented by a node.

Let n be a node in T. If m is not the root (representing the empty string) by the compactness of T it might either be a leaf or a branching node.

If m is a leaf, it represents a suffix $u = \texttt{path}(m)$ of t. Suppose the suffix is nested, then the string uw with $w \in \Sigma^+$ is in $\texttt{words}(T)$ and there is a node n such that $\texttt{path}(n) = uwv$ with $v \in \Sigma^+$. Hence, there is an edge from m to n which is a contradiction to m being a leaf.

If m is a branching inner node, there must be two nodes n, n', children of m. Let $w = \texttt{path}(m)$. Since all outgoing edges at m have labels with different start characters, $\texttt{path}(n) = wav$ and $\texttt{path}(n') = wbv'$ for $v, v' \in \Sigma^*$. Therefore, wa, wb are substrings of t and w is right branching in t. □

LEMMA 6 (Chain Property of Suffix Links). *Let n be a node in the CST T for the string t. The node n is either the root or the leaf representing the shortest non-nested suffix, or n has an atomic suffix link.*

PROOF. Let n be a node that is not the root and not the node representing the shortest non-nested suffix of t. Let $w = \texttt{path}(n)$. If n is a leaf, it represents a non-nested suffix. Because n does not represent the shortest non-nested suffix there is a shorter non-nested suffix w' such that $aw' = w$ for $a \in \Sigma$. Since w' is a non-nested suffix there is a node n' representing w' and there is an atomic suffix link from n to n'. If n is not a leaf it must be branching. Therefore w is right branching in t. Let $w' \in \Sigma^*$ be a proper suffix of w such that $aw' = w$ for $a \in \Sigma$. By Observation 1, w' is also right branching and, by Lemma 5, represented by a node n'. Therefore, there is an atomic suffix link from n to n'. □

From the above lemmas we know that the reverse tree T^{-1} of a CST T for the string t represents a subset of all substrings of t^{-1}, namely, the left branching substrings and t^{-1} itself (see Figure 1(b) and (c)). The idea of an affix tree is to augment a suffix tree with nodes such that it represents all suffixes of t and its reverse represents all suffixes of t^{-1} (respectively the prefixes of t).

Affix trees can be defined as “dual” suffix trees.

DEFINITION 7 (Affix Tree). An affix tree T of a string t is a Σ^+-tree such that

$$\begin{aligned}\texttt{words}(T) &= \{u \mid u \text{ is a suffix of } t\} \quad \text{and}\\ \texttt{words}(T^{-1}) &= \{u \mid u \text{ is a suffix of } t^{-1}\}.\end{aligned}$$

Figure 1(d) and (e) shows the affix tree for `ababc`. Figure 1(d) focuses on the suffix tree like structure and Figure 1(e) on the reverse suffix tree (prefix tree) like structure. The differences to the CST and to the CPT in each structure are the additional nodes that are boxed in the figure (compare (a), (c), (d), and (e) in Figure 1).

Note that a substring w of t might have two different locations in the CAT T for t, the location of w in the CST part and the location of w^{-1} in the reverse part. If these locations are implicit, then they are not represented by the same node but lie "in" two different edges. The location of `bab` appears in the suffix edge between the nodes $\overline{\texttt{b}}$ and $\overline{\texttt{babc}}$ (Figure 1(d)) and in the prefix edge between the nodes $\overline{\texttt{ba}}$ and $\overline{\texttt{baba}}$ (Figure 1(e)). The location of `ba` appears in the suffix edge between the nodes $\overline{\texttt{b}}$ and $\overline{\texttt{babc}}$ (Figure 1(d)) and in the prefix edge between the root and $\overline{\texttt{aba}}$ (Figure 1(e)).

Having two different implicit locations complicates the process of making a location explicit (i.e., inserting a node). Both implicit locations must be found for that. To insert a node $\overline{\texttt{ba}}$ (respectively $\overline{\texttt{ab}}$ in the prefix view) into the affix tree shown in Figure 1(d) and (e), the two edges ($\overline{\texttt{b}}$, $\overline{\texttt{babc}}$) and (root, $\overline{\texttt{aba}}$) would have to be split.

The CAT for t represents two trees in one. It represents the CST for t and the CST for t^{-1}. The latter is called the CPT for the string t. Because of Lemma 5 the CAT contains all nodes that would be part of the CST and all nodes part of the CPT. As an application of the lemma, one can classify each node, whether it belongs to the suffix part, the prefix part, or both parts. We can thus define a node type. A node is a *suffix node* if it is a suffix leaf (it has no outgoing suffix edges), if it is the root, or if it is a branching node (it has at least two outgoing suffix edges). A node is a *prefix node* if it is a prefix leaf (it has no outgoing prefix edges), if it is the root, or if it is a branching node (it has at least two outgoing prefix edges). The attribute need not be saved explicitly but can be reconstructed on-line. Edges are either suffix or prefix edges. The suffix links are the (reversed) prefix edges and vice versa. Therefore, all edges need to be traversable in both directions. We can thus distinguish between the suffix structure and the prefix structure of the CAT.

2.3. *Additional Data for the Construction of CSTs and CATs.*

For the algorithmic part we need to be able to keep track of locations in the CST or CAT and be able to reach these quickly. Ukkonen [7] defined *reference pairs* for that reason for use in the construction of CSTs. A reference pair is a pair (n, u) where n is a node (called the *base* of the reference pair), $u \in \Sigma^*$ is a string, and $\texttt{path}(n)u$ is the represented location. This way implicit and explicit locations can be represented. To reach the location we start at the base and move along edges, whose labels represent the same string as u. The string part u can be easily represented in constant space with an index and a length characterizing a substring of the underlying string t (similar to edge labels). The reference pair is then a pair $(n, (o, l))$ where (o, l) represents the substring u of t, $u = t_o \cdots t_{o+l-1}$.

As mentioned above, there may be two locations of a substring of t in CAT(t), one in the suffix structure and one in the prefix structure. We therefore need to distinguish between a suffix reference pair, where the represented location is reached by moving along suffix edges, and a prefix reference pair, where the represented location is reached by moving along prefix edges.

For the construction of suffix trees, a reference pair (n, u) is *canonical* if there is no node m and a string v such that $\texttt{path}(n)$ is a proper prefix of $\texttt{path}(m)$ and $\texttt{path}(n)u = \texttt{path}(m)v$. For the construction of affix trees, we use suffix and prefix reference pairs. For suffix reference pairs (n, u) we require n to be a suffix node and for prefix reference pairs (n, u) we require n to be a prefix node. A *suffix* reference pair is canonical if there is no *suffix* node m and a string v such that $\texttt{path}(n)$ is a proper prefix of $\texttt{path}(m)$ and $\texttt{path}(n)u = \texttt{path}(m)v$. In other words, a canonical reference pair is a reference pair where the base node has maximal depth.

For example, in Figure 1 with underlying string $t = t_0 \cdots t_4 = \texttt{ababc}$ ($t^{-1} = t_{-1} \cdots t_{-5} = \texttt{cbaba}$) the canonical prefix reference pair for $\texttt{ab}$ is (root, $(-3, 2)$), the canonical suffix reference pair for $\texttt{ba} = (\texttt{ab})^{-1}$ is ($\overline{\texttt{ab}}$, $(3, 0)$). A non-canonical suffix reference pair for $\texttt{ba}$ is (root, $(1, 2)$).

We will also use *open edges* as introduced by Ukkonen [7]. The edges leading to leaves will be labeled with the starting point and an infinite end point (i.e., (i, ∞)) that represents the end of the underlying string. Such an edge will always extend to the end of the string. An edge is thus a quintuple $e = (n_{start}, n_{end}, i_{start}, i_{end}, isOpenEdge)$, such that the string label is $t_{i_{start}} \cdots t_{i_{end}-1}$ if *isOpenEdge* is false, and the edge $t_{i_{start}} \cdots t_n$ otherwise (for a shorthand we write (i_{start}, i_{end}) if *isOpenEdge* is false and (i_{start}, ∞) otherwise).

There are some important locations in the CAT. Let $\alpha(t)$ be the *longest nested suffix* of string t and let $\hat{\alpha}(t)$ be the *longest nested prefix* of t ($\alpha(t) = (\hat{\alpha}(t^{-1}))^{-1}$). The longest nested suffix is called the *active suffix* and the longest nested prefix is called the *active prefix*. By Lemma 5, all longer suffixes (respectively prefixes) are represented by leaves. As we will see later all updating is done "around" the positions of the active suffix and the active prefix. We therefore always keep the following data for the construction of CATs:

- A canonical suffix reference pair for the active suffix point $asp_{suffix}(t)$, the suffix location of the active suffix.
- A canonical prefix reference pair for the active suffix point $app_{prefix}(t)$, the prefix location of the active suffix, which is the active prefix of the reverse string.
- A canonical prefix reference pair for the active prefix point $asp_{prefix}(t)$, the prefix location of the active prefix. The active prefix is the active suffix of the reverse string, so it is the counterpart to $asp_{suffix}(t)$ only in the prefix view.
- A canonical suffix reference pair for the active prefix point $app_{suffix}(t)$, the suffix location of the active prefix.
- The active suffix leaf $asl_{suffix}(t)$, which represents the shortest non-nested suffix of t.
- The active prefix leaf $asl_{prefix}(t)$, which represents the shortest non-nested prefix of t.

For the four reference pairs and the two leaves we also keep the "depth" (the length of the represented string). This is needed in order to calculate the correct indices of edge labels later in the algorithm.

A non-nested suffix, represented by a suffix leaf in the CST, is not represented by a node in the CPT because it occurs only once as the prefix of t^{-1}. Hence, all suffix leaves must be hidden in a CPT-like view of the CAT.

By Lemma 6, the asl_{suffix} is the only suffix node other than the root that may have a non-atomic suffix link. Therefore, all suffix leaves form a chain with atomic suffix links because if a leaf is not the asl_{suffix}, there is a leaf representing a suffix that is shorter by one character. The prefix parent of asl_{suffix} will always be a prefix node, too, and hence represented by a node in the CPT:

THEOREM 8 (Parent of the Active Suffix Leaf). *The prefix* (*respectively suffix*) *parent of the active suffix* (*respectively prefix*) *leaf is always a prefix* (*respectively suffix*) *node.*

PROOF. The $asl_{suffix}(t)$ is the leaf representing the shortest non-nested suffix. Let s be the prefix parent of $asl_{suffix}(t)$. Suppose s is not a prefix node. Then s must be a suffix node and $w = \texttt{path}(s)$ is right branching in t. By the definition of suffix links, u is a suffix of $\texttt{path}(asl_{suffix}(t))$ and hence a suffix of t. For w to be right branching there must be at least two occurrences of w in t, one followed by a, the other by b. Therefore, $t = \{\ \_\_\_\_\_\_xw, \_\_\_\_ywa\_\_, \_\_zwb\_\_\_\_\ \}$ with $x, y, z \in \Sigma \cup \{\varepsilon\}$. Either y and z are two different characters ($y \neq z$, in which case w is left branching and by Lemma 5 a prefix node) or they are equal, or one of the additional occurrences of w is as a prefix of t ($y = \varepsilon$ or $z = \varepsilon$). If $y = z$, then either $x = y = z$ and xw would be a suffix node, a contradiction to s being the parent of $asl_{suffix}(t)$, or $x \neq y = z$ and w is left branching, making s a prefix node. Hence, w occurs as a prefix of t. Without loss of generality $z = \varepsilon$: $t = \{\ \_\_\_\_\_\_xw, \_\_ywa\_\_\_\_, wb\_\_\_\_\_\_\ \}$. If $x \neq y$, then w is left branching and by Lemma 5 a prefix node. Otherwise let $c = x = y$. Taking a look at the next position to the left of c we find that $t = \{\ \_\_\_\_\_\_xcw, \_\_ycwa\_\_\_\_, wb\_\_\_\_\_\_\ \}$. Again, if $x \neq y$ we have a prefix node representing a larger suffix than s and hence a contradiction to s being the prefix parent of $asl_{suffix}(t)$.

Iterating this argument leads to $t = \{\ \_\_\_\_xuw, uwa\_\_\_\_, wb\_\_\_\_\ \}$. Therefore wb is a prefix of uw. (Clearly, $|u| > 0$ because otherwise $a = b$ and s would not be a suffix node. This would contradict s not being a prefix node.) It follows that $t = \{\ \_\_\_xwb\_, uwa\_\_\_\_, wb\_\_\_\_\ \}$. Let $u = vd$ for $d \in \Sigma$: $t = \{\ \_\_\_xwb\_, vdwa\_\_\_\_, wb\_\_\_\_\ \}$. If $x = d$, then dw is right branching and represented by a node. This contradicts s being the prefix parent of $asl_{suffix}(t)$. Therefore $x \neq d$ and w is left branching. Thus, s is a prefix node. ☐

From Lemma 6 and the above we conclude that all nodes that occur in the CPT for t but not in the CST for t (so called "prefix only nodes") occur in chains with atomic edges limited by suffix nodes. As a consequence, we can cluster the prefix nodes in the CAT that would not be part of the CST. Let p be a prefix node that is not a suffix node. Hence, p has exactly one suffix child and a suffix parent. If the suffix parent or the suffix child of p is also a non-suffix node, the edge length must be 1. For the suffix structure the prefix nodes can thus be clustered to *path*s, where the nodes are connected by edges of length 1. It suffices to know the number of nodes in the path to know the string length of the path. What would be an edge in the CST is then an edge leading to a non-suffix node, a path of prefix nodes, and a final edge leading again to a suffix node in the CAT. See Figure 1(d) and (e), where the nodes are already in boxes corresponding to their paths.

We will introduce a path structure with the properties that

- all non-suffix (or non-prefix) nodes in the suffix (or prefix) structure are clustered in paths,
- access to the ith (in particular the first and last) node takes constant time,
- the first and the last node of the path have a reference to the path and know the path length,
- a path can be split into two in constant time similar to an edge (when a node in the path becomes a suffix node),
- nodes can be dynamically appended and prepended.

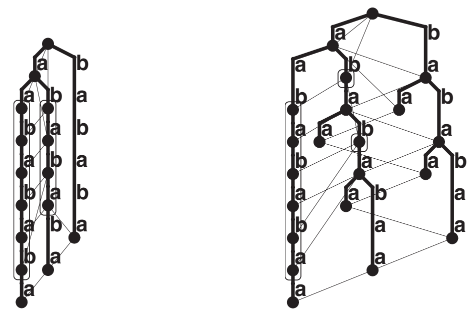
With the first three properties the CAT can be traversed in the same asymptotic time as the corresponding CST and with the last two properties the updating will also work in the same asymptotic time.

The CAT can be traversed ignoring nodes of the reverse kind: if a non-suffix node is hit while traversing the suffix part, it must have a pointer to a path. The last node of that path is again a prefix node with a single suffix node as suffix child. This is the key to achieving a linear time bound in the construction of CATs.

We briefly take a closer look at the details of the data structures. The edges need to be traversable in both directions. Therefore, we keep a starting and ending point and the indices (including a flag to mark open edges) into the underlying string with each edge. To traverse the edges in both directions, we need to store the outgoing suffix and prefix edges indexed by their starting character and the suffix and prefix parent edges at the nodes. The first and last nodes of a path need to have a reference to the path. A node cannot be part of two paths, so each node carries a single path pointer. The type of a path can be derived from the node type. The adjacency list can be stored in various ways (trees, list, arrays, hash tables). The optimal storage depends on the application, for the algorithm the alphabet is assumed fixed and the only thing needed is a procedure getEdge(*node*, *type*, *starting-character*) that returns the edge of the given type and with the given character.

The paths are stored as two dynamic arrays which can grow dynamically with constant amortized cost per operation (see Section 18.4, “Dynamic tables,” in [2]). The size of the path is linear in the number of nodes it contains. Paths never need to be merged: Except for the active prefix leaf, no nodes are ever deleted. Between the active prefix leaf and the next prefix node closer to the root there is always one suffix node (see Theorem 8), so the two clusters can never merge. An inner node that is of type suffix represents a right branching substring w of t. Regardless of appending or prepending to t the substring w will always be branching.

To be able to split paths we use path pointers. A path pointer has a reference to the path, an index to the first node pointer, and a length. When a path is split, two path pointers share the same dynamic array structure. For example, the nodes $\overline{\texttt{ab}}$, $\overline{\texttt{aba}}$, $\overline{\texttt{abab}}$, $\overline{\texttt{ababa}}$ are in a common path in the CAT for $t = \texttt{aabababa}$ (Figure 2(a)). After the CAT is extended to $t = \texttt{aababababaa}$ (Figure 2(b)), the nodes $\overline{\texttt{aba}}$ and $\overline{\texttt{ababa}}$ are suffix nodes and no longer path members. The nodes $\overline{\texttt{ab}}$ and $\overline{\texttt{abab}}$ will now share the same dynamic array, but the array will not need to grow because the nodes $\overline{\texttt{a}}$, $\overline{\texttt{aba}}$, and $\overline{\texttt{ababa}}$ will never be deleted and no other nodes can be inserted between them.



(a) (b)

Fig. 2. The CATs for `aabababa` and for `aabababaa`.

The size of a CAT depends on the way the children of a node are stored. Assuming a constant size alphabet Σ, it makes no difference to the asymptotic running time whether we use linear lists, hash tables, or arrays. For a small alphabet Σ (e.g., $|\Sigma| = 4$), an array will be the smallest and fastest solution. In that case the size of an affix tree for a string of size n can be estimated as follows. Add the size of the two parent edges to the node they lead to, so for each node we count the two parent edges (each has two indices, one open edge flag, one type flag, and two pointers), the two children arrays (each has $|\Sigma|$ pointers), the two parent pointers (one pointer each), a path pointer (two indices, one pointer), and a path array structure (two pointers to dynamic arrays, five indices for size, length, and a reference count, one pointer to a node). The number of nodes is at most $2n$. The size of the underlying string (as a two way extendable dynamic array structure) is two pointers, four indices, and at most $2n$ characters. The size of a reference pair is one pointer and three indices. The two *asl*'s and their depth need two pointers and two indices. An upper bound for the total size of the CAT is then $\text{size}(\text{CAT}(t)) \leq n \cdot (22i + 2c + 8f + (22 + 4|\Sigma|)p) + 8p + 18i = i \cdot (22n + 18) + p \cdot ((22 + 4|\Sigma|)n + 8) + f \cdot 8n + c \cdot 2n$. Assuming a 4 byte machine architecture (flags of 2 byte, characters of 1 byte, pointer, and indices of 4 byte) and a 4 byte alphabet, the size is then at most $258n + 104$ bytes.

3. Construction of Compact Suffix Trees. In order to prepare the construction of CATs, we briefly describe how to construct CSTs by expanding the underlying string once to the right and once to the left.

These problems are solved by two already existing algorithms, namely, Ukkonen's algorithm [7] and Weiner's algorithm [8]. We assume some additional information readily available in the later algorithm for constructing CATs when constructing the CST by extending the string at the front, so the described algorithm is not identical to Weiner's algorithm (but simpler). The results of this section are due to Ukkonen and Weiner. We replicate them in detail here to prepare the presentation of the affix tree construction algorithm.

3.1. *On-Line Construction of CSTs.* Ukkonen's algorithm constructs a CST on-line from left to right, building $\text{CST}(ta)$ from $\text{CST}(t)$. The algorithm can be intuitively derived from an algorithm that constructs ASTs. It essentially works by lengthening all suffixes represented in CST(t) by the character a and inserting the empty suffix. Obviously, the resulting tree is CST(ta).

To analyze the algorithm the suffixes are divided into different types:

Type 1. Non-nested suffixes of t.
Type 2. Nested suffixes s of t, where sa is a substring in t.
Type 3. Nested suffixes s of t, where sa is not a substring in t. We call sa a *relevant suffix*.

We deal with each of these sets of suffixes when extending them by a:

Case 1. For all suffixes s of t of type 1 sa is non-nested in ta (otherwise sa would occur elsewhere in t and s would be nested). The suffix s was represented by a leaf (Lemma 5) in CST(t) and sa will be represented by a leaf in CST(ta). With the use of open edges for leaves, all leaves will extend automatically to the end of the string. Nothing needs to be done for these suffixes.
Case 2. For all suffixes s of t of type 2 sa is a substring of t. Hence, sa is a nested-suffix of ta. Nothing needs to be done for these suffixes.
Case 3. For all suffixes s of t of type 3 sa is a non-nested suffix and must thus be represented by a leaf in CST(ta). The suffix s is not represented by a leaf in CST(t).

If sa is a relevant suffix (type 3), then s is a nested suffix of t. Therefore, s must be a proper suffix of t and s occurs as a substring elsewhere in t ($t = \{\ \_s\_\_, \_\_\_s\ \}$). sa is non-nested in ta, so sa cannot occur in t. Therefore $ta = \{\ \_sb\_\_, \_\_\_sa\ \}$ with $b \neq a$ and s is right branching in ta. To update CST(t) with the relevant suffix sa, a node $\overline{\mathtt{s}}$ must be inserted (unless s was right branching in t and $\overline{\mathtt{s}}$ already exists) and a new leaf $\overline{\mathtt{sa}}$ must be added at the (possibly new) node $\overline{\mathtt{s}}$.

LEMMA 9. *All relevant suffixes sa are suffixes of $\alpha(t)a$ and $\alpha(ta)$ is a suffix of all relevant suffixes.*

PROOF. The active suffix $\alpha(t)$ is the longest nested suffix of t. If sa is a relevant suffix, s is a nested suffix of t. Hence s is a suffix of $\alpha(t)$ and sa a suffix of $\alpha(t)a$. $\alpha(ta)$ cannot be longer than $\alpha(t)a$. Suppose it was larger, then $\alpha(ta) = v\alpha(t)a$ for $v \in \Sigma^+$. Since $\alpha(ta)$ is nested in ta, $ta = \{\ \_\alpha(ta)\_, \_\_\alpha(ta)\ \}$. It follows that $t = \{\ \_v\alpha(t)a\_, \_\_v\alpha(t)\ \}$ and $v\alpha(t)$ would be a longer nested suffix of t. This contradicts the definition of the active suffix α.

$\alpha(ta)$ and $\alpha(t)a$ are suffixes of ta, therefore $\alpha(ta)$ is a suffix of $\alpha(t)a$. Since a relevant suffix sa is non-nested in ta, it is longer than $\alpha(ta)$, hence $\alpha(ta)$ is a suffix of sa. □

COROLLARY 10. *$\alpha(ta)$ is a suffix of $\alpha(t)a$.*

Let $\alpha(t) = t_{m_n} \cdots t_n$ and $\alpha(ta) = t_{m_{n+1}} \cdots t_n a$. The relevant suffixes are then $s_k = t_{m_n+k} \cdots t_n$ with $k \in \{0, \dots, m_{n+1} - m_n - 1\}$ (if $m_n = m_{n+1}$ there is no relevant suffix).

Procedure 1: canonize *reference pair* $(b, (o, l))$

```
    $(b, (o, l))$ is a reference pair that is to be canonized.

 1: if $l \neq 0$ then {Otherwise the reference pair is already canonical}
 2:   $e := getEdge(b, SUFFIX, t_o)$;
 3:   $len := getLength(e)$;
 4:   while $len \leq l$ do
 5:     $o := o + len$;
 6:     $l := l - len$;
 7:     $b := getEndNode(e)$;
 8:     $e := getEdge(b, SUFFIX, t_o)$;
 9:     $len := getLength(e)$;
10:   end while
11: end if
```

The (canonical) reference pair $asp_{suffix}(t) = (b, (o, l))$ represents the active suffix $\alpha(t)$ such that $\alpha(t) = \mathtt{path}(b)t_o \cdots t_{o+l-1}$ (here, $o + l - 1 = n$). Therefore b represents the prefix $t_{m_n} \cdots t_{o-1}$ of $\alpha(t)$. For all relevant suffixes, we can reach the location of s with $asp_{suffix}(t)$ as follows: $s_0 = \mathtt{path}(b)t_o \cdots t_{o+l-1}$ $(n = o + l - 1)$. For s_{k+1}, if b is not the root, it has a suffix link to b', such that $\mathtt{path}(b) = t_{m_n+k}\mathtt{path}(b')$ and $s_{k+1} = \mathtt{path}(b')t_o \cdots t_{o+l-1} = t_{m_n+k+1} \cdots t_n$. The new base is canonized (see Procedure 1), which also changes o and l. Otherwise, if b is the root, then $s_k = t_o \cdots t_{o+l-1}$ and we can get s_{k+1} by increasing o and decreasing l: $s_{k+1} = t_{o+1} \cdots t_{(o+1)+(l-1)-1}$. The relevant suffixes are thus reached by starting at $asp_{suffix}(t)$ and iteratively moving over suffix links and canonizing. It can easily be determined whether we have reached a non-relevant suffix (type 2) by checking whether the current reference pair asp_{suffix} can be extended by a. If so, the current represented suffix is nested and the current reference pair is $asp_{suffix}(ta)$.

The suffix links for the new nodes can be set during the process or afterwards, if we remember the inserted (or found) nodes w on a stack. Node w from an iteration of the while loop is linked to the w of the current iteration. The last w will get a suffix link to an already existing node (called the *end point*), which represents a string one character shorter than $\alpha(ta)$ and is either the base of $asl_{suffix}(ta)$ or one edge below the base. By Lemma 6, the end point always exists.

Procedure 1 takes a reference pair $(b, (o, l))$ and changes the base, the offset, and the length, so that the resulting reference pair represents the same string and is canonical. This is done by following edges, that represent suffixes of the string part of the reference pair, down the tree until the node is closest to the represented location.

LEMMA 11. *The amount of work performed during all calls to* canonize() *with the reference pair $asp_{suffix}(t)$ is $O(n)$.*

PROOF. The amount of work performed during canonizing the reference pair $asp_{suffix}(t)$ can be bound as follows: Each time a character is appended to t a coin is put on the character. The reference pair $asp_{suffix}(t)$ consists of a base b, a start index o into t, and a length l. The string part of the reference pair is $t_o \cdots t_n$ leading to the implicit location. Each time the base is moved along an edge e down the tree, the length l is decreased and the index o is increased by the length *len* of the edge. We take the coin from character t_o to pay for the step. Since o is never decreased, there is always a coin on t_o. As a result, canonizing has amortized constant complexity per call. □

THEOREM 12. *Ukkonen's algorithm constructs* CST(t) *on-line in time* $\mathcal{O}(|t|)$.

PROOF. The only part with non-constant running time per iteration is the insertion of the relevant suffixes. The number of nodes of CST(t) is at most $2n$, the number of leaves is at most n (there are at most n leaves, so there cannot be more than n branching nodes). Each time except once per iteration that a suffix link is taken at least one leaf is inserted. The number of links taken is therefore less than n. The amount of work performed during canonizing the reference pair $asp_{suffix}(t)$ is O (n) by Lemma 11. □

See Figure 3 which shows the intermediate steps of the on-line construction of CAT(`acabaabac`) from CAT(`acabaaba`). The reference pair starts out with the base $\overline{\texttt{a}}$ and the remaining string `ba`, i.e., as $(\overline{\texttt{a}}, (6, 2))$. After the insertion of the new leaf between (b) and (c), the suffix link from $\overline{\texttt{a}}$ to the root is taken. The edge starting with `b` has length 6 and thus no canonizing takes place. After the insertion of the second leaf between (c) and (d) the offset is increased and the length decreased (instead of taking a suffix link). The resulting reference pair is (root, (5, 1)) with remaining string `a`. The reference pair is then canonized to (`a`, (6, 0)), where it can be extended to (`a`, (6, 1)) with remaining string `c`.

3.2. *Anti-On-Line CST Construction with Additional Information.* Anti-on-line construction builds CST(at) from CST(t). CST(t) already represents all suffixes of CST(at) except at. Only the suffix at needs to be inserted. The leaf for at will branch at the location representing $\hat{\alpha}(at)$ in CST(t).

We assume that we have the additional information of knowing the length of $\hat{\alpha}(at)$ available in each iteration of the following algorithm. With this, we do not need the "indicator vector" used by Weiner to find the length of the new active prefix.

There are three possibilities. The node location may be represented by an inner node, the location is implicit in some edge, or the location is represented by a leaf. In the last case the new leaf does not add a new branch to the tree, but extends an old leaf that now represents a nested suffix. The node representing the old leaf needs to be deleted after attaching the new leaf to it (it represents a nested suffix now, see Lemma 5).

LEMMA 13. $\hat{\alpha}(at)$ *is a prefix of* $a\hat{\alpha}(t)$.

PROOF. Obviously, both strings are prefixes of $at = \{\ a\hat{\alpha}(t)\_\_\_\_,\ \hat{\alpha}(at)\_\_\_\_\ \}$. Either $\hat{\alpha}(at)$ is a prefix of $a\hat{\alpha}(t)$ or $a\hat{\alpha}(t)$ is a proper prefix of $\hat{\alpha}(at)$. If $a\hat{\alpha}(t)$ were a proper prefix of $\hat{\alpha}(at)$, then $\hat{\alpha}(at) = a\hat{\alpha}(t)v$ for $v \in \Sigma^+$. Hence, $t = \{\ \hat{\alpha}(t)v\_\_\_\_,\ \_\_a\hat{\alpha}(t)v\_\_\ \}$, which contradicts the definition of $\hat{\alpha}$. □

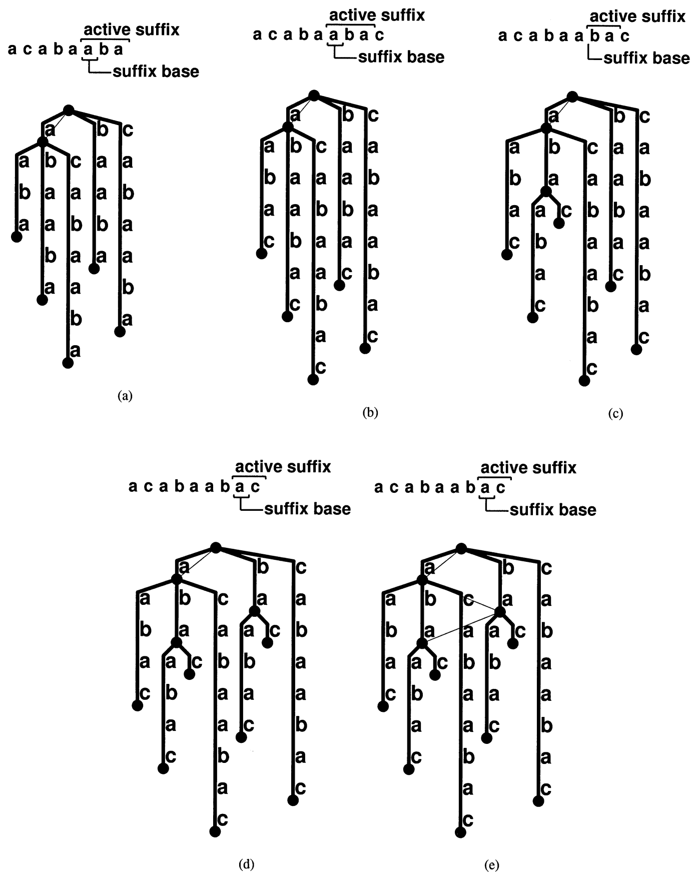


Fig. 3. Steps of constructing CST(acabaabac) from CST(acabaaba). The active suffix as represented by asp_{suffix} is shown above each tree. The substring marked suffix base is the part of the active suffix that is represented by the base of the (canonical) reference pair. The offset index of the reference pair points to the character one right of the substring represented by the base. (a) The starting point. (b) The open edges have grown with the text. (c) The tree after the first relevant suffix has been inserted and the base of the reference pair has been moved over a suffix link. (d) The tree after the second relevant suffix has been inserted, the reference pair has been at the front, canonized to $\overline{\text{a}}$, and could be enlarged by the new character. (e) The final tree with the updated suffix links.

Let $t = t_{-n} \cdots t_{-1}$ and $at = t_{-n-1}t_{-n} \cdots t_{-1}$ ($a = t_{-n-1}$). For the purpose of constructing the CST(at) from CST(t) we keep a canonical reference pair $app_{suffix}(t)$ for $\hat{\alpha}(t)$. Let $r_{n+1} = (b_{n+1}, v_{n+1})$ be a canonical reference pair for $\hat{\alpha}(at) = t_{-1-n} \cdots t_{m_{n+1}}$ in CST(t). The base b_{n+1} of r_{n+1} has a suffix link to a node s (Lemma 6). Hence, $r' = (s, v_{n+1})$ is a reference pair for $t_{-n} \cdots t_{m_{n+1}}$, which is a prefix of $\hat{\alpha}(t)$ (Lemma 13). Let $r_n = (b_n, v_n) = app_{suffix}(t)$. Since $t_{-n} \cdots t_{m_{n+1}}$ is a prefix of $\hat{\alpha}(t)$, there is a path $p = (s, \ldots, b_n)$ in CST(t) (p might have length 0). For any node $s' \in p$ if $s' \neq s$ and $s' \neq b_n$, there is no node q such that there is a prefix edge labeled a from s' to q: If a q existed, then it could either be a leaf or an inner node. If it is a leaf (assuming

Procedure 2: decanonize(*reference pair* $(b, (o, l))$, *character* a)

$(b, (o, l))$ is a reference pair that is to be decanonized, a is the character for which a prefix edge is to be found.

```
1: $e := getEdge(b, PREFIX, a)$; {If there is no edge, $e$ will be $nil$}
2: while $e = nil \vee getLength(e) \neq 1$ do
3:    $f := getParent(b, SUFFIX)$;
4:    if $f = nil$ then {If $b$ has no parent it is the root. The index $o$ should then point to
      the beginning of the old string. Make it point to the beginning of the new string.}
5:       $o := o - 1$;
6:       $l := l + 1$;
7:       return;
8:    else
9:       $len := getLength(f)$;
10:      $o := o - len$;
11:      $l := l + len$;
12:      $b := getStartNode(f)$;
13:   end if
14: end while
15: $b := getEndNode(e)$;
```

we have suffix links for leaves), then it must be the active prefix leaf because it is one character longer than the longest nested prefix. The leaf's suffix link would point to a node s' representing $\hat{\alpha}(t)$. Hence, $s' = b_n$. If q is an inner node, $\texttt{path}(q)$ would be right branching in t by Lemma 5. Because $\texttt{path}(q)$ is a nested prefix of at, it cannot be larger than $\hat{\alpha}(at)$. Therefore, q would be a node above b_{n+1} and r_{n+1} would not be canonical.

As a result, we have a way to find $app_{suffix}(at)$ from $app_{suffix}(t)$ by *decanonizing* $app_{suffix}(t) = (b, (i, l))$: Start at the base b and walk up towards the root until a prefix edge e labeled with a is found. For each edge found on the way up subtract its length from i. Replace b by the target b' of e. With the knowledge of the length of $\hat{\alpha}(at)$ we can set the length l to $|\hat{\alpha}(at)| - \texttt{depth}(b')$ and get $app_{suffix}(at)$.

Since we know the $\hat{\alpha}(at)$ at the beginning of the iteration it is easy to remember the node traversed at depth $|\hat{\alpha}(at)| - 1$ when decanonizing. A prefix edge labeled a is inserted from that node to the node where the new leaf is attached.

LEMMA 14. *The amount of work performed during all calls to* decanonize() *with the reference pair* $app_{prefix}(t)$ *is* $O(n)$.

PROOF. We apply amortized analysis as follows: Each time a character is prepended to t a coin is put on the character. As described above, for each edge the base is moved upwards, the index o of the string part of the reference pair is decreased. We take the coin on t_o to pay for the step. o stays unchanged when the base is replaced or when the length is adjusted. If decanonize() moves the base of $app_{prefix}(t) = (b, (o, l))$ to a leaf, l must be zero (otherwise there would not be a location for $app_{prefix}(t)$). Hence, the leaf will

Procedure 3: update-new-suffix(*reference pair* $(b, (o, l))$, character a, int newlen)

newlen is the new length of the reference pair after it has been decanonized.

```
1: decanonize $((b, (o, l)), a)$;
2: if $l = 0$ then
3:     $w = b$;
4: else
5:     Insert a node $w$ at the location of $(b, (o, l))$ by splitting the edge with starting char-
       acter $t_o$.
6: end if
7: Insert a new leaf $l$ linked to $w$ by an edge with string label $(o, n + 1)$.
```

need to be deleted when attaching the new suffix to it. The base of $app_{prefix}(t)$ will move up towards the next node, decreasing the offset o. Therefore, o never increases and there is always a coin on t_o. As a result, decanonizing has amortized constant complexity. □

THEOREM 15. *With the additional information of knowing the length of $\hat{\alpha}(s)$ for any suffix s of t before inserting it, it takes time $\mathcal{O}(|t|)$ to construct* CST(t) *in an anti-on-line manner, i.e., reading t in reverse direction from right to left.*

PROOF. Inserting the leaf, inserting an inner branching node, or deleting an old leaf takes constant time per step. The non-constant part of each iteration is the decanonizing of $app_{suffix}(t)$. By Lemma 14 the work can be done in constant amortized time. □

A more detailed discussion of Weiner's algorithm using the terms of Ukkonen's algorithm can be found in [3].

See Figure 4 where the construction of CPT(acabaabac) from CPT(acabaaba) is shown. If we do not have prefix links for leaves, the reference pair for $app_{prefix}(t)$ is decanonized by moving the base from $\overline{\texttt{aba}}$ over $\overline{\texttt{a}}$ to the root, where the offset decreased by one. The length is then adjusted to 2. The leaf will be deleted and therefore there is no need to canonize to $\overline{\texttt{ca}}$. If we have prefix links for leaves, there is a prefix edge labeled c from $\overline{\texttt{a}}$ to $\overline{\texttt{ca}}$, which is found after moving the base up to $\overline{\texttt{a}}$. Before deleting the $\overline{\texttt{ca}}$, the base of the reference pair needs to be moved back up to the root.

4. Constructing Compact Affix Trees On-Line.

A first algorithm for constructing CATs on-line was presented by Stoye in his Master's thesis [5]. We give a complete presentation of the results which can in parts also be found in Stoye's Master's thesis. Especially the relations between nodes and "neuralgic points," like the active suffix point and the active suffix leaf, are extensively studied in [5]. Our display of the algorithm differs from Stoye's by placing a stronger emphasis on the derivation from Ukkonen's and Weiner's suffix tree construction algorithms. The resulting global structure is similar to Stoye's. For the unidirectional construction the main difference lies in the use of paths to overcome the problem of non-atomic suffix links. It seems unlikely that an asymptotic

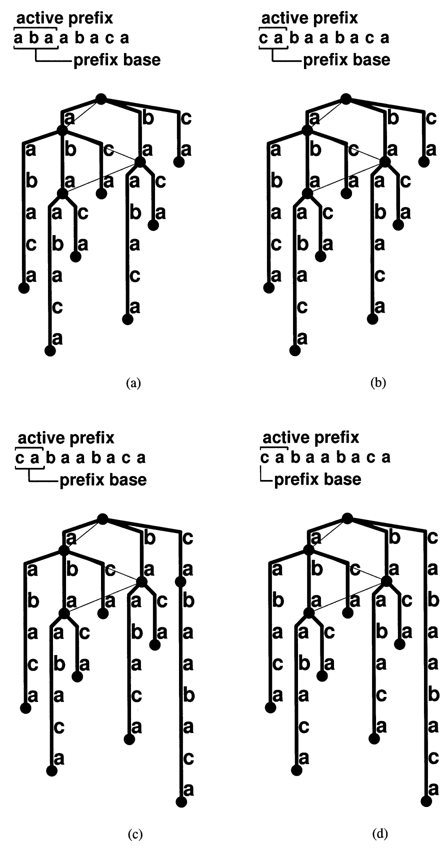


Fig. 4. Steps of constructing CPT(acabaabac) from CPT(acabaaba). The active prefix as represented by app_{prefix} is shown above each tree. The substring marked prefix base is the part of the active prefix that is represented by the base of the (canonical) reference pair. The offset index of the reference pair points to the character one right of the substring represented by the base. (a) The starting point. (b) The app_{prefix} has been decanonized and the location of $\hat{\alpha}$(acabaabac) has been found. (c) The tree with the new suffix inserted. It was added at a leaf which is then deleted to give the final tree in (d).

linear time bound can be proven without these paths. This is the first known linear time algorithm for the construction of CATs.

4.1. *Overview.* The algorithm is dual by design. The necessary steps to construct CAT(ta) from CAT(t) are the same as for constructing CAT(at) from CAT(t), we only have to exchange "suffix" with "prefix" in the description. For the sake of clarity we assume the view of constructing CAT(ta) from CAT(t). We call this a suffix iteration, while constructing CAT(at) from CAT(t) is called a prefix iteration.

The underlying string must be accessible from the two views in different ways. In the suffix view the string is accessed as usual. In the prefix view edges will be labeled with reverse strings, i.e., a prefix edge from the prefix node for $t_k \cdots t_{k+l}$ to the prefix node for $t_{k-j} \cdots t_{k+l}$ will be labeled with the reverse of $t_{k-j} \cdots t_k$, the start index would

Function 1: getEndNodeVirtualEdge(*edge e*) **returns** *node*

```
    e is a suffix edge that possibly ends in a prefix node.
 1: $n$ := $getEndNode(e)$;
 2: if $n$ has only one suffix child then {$n$ is a prefix node}
 3:     $p = getPath(n)$; {$n$ is contained in a path $p$}
 4:     $n' = getLast(n)$; {$n'$ is the last node of path $p$}
 5:     $e' = getOnlyEdge(n', SUFFIX)$; {$n'$ is a prefix node (in $p$) and has only one suffix
        child}
 6:     $n'' = getEndNode(e')$;
 7:     return($n''$);
 8: else
 9:     return($n$); {There is no path}
10: end if
```

be larger than the end index. To make the view truly dual, we will access the string with the index transformation $r(k) = -1 - k$ from the prefix side. Thus, we can label the edge above with $(-1 - k, -1 - k + j)$, which are increasing indices. We also keep the prefix reference pairs with the transformed indices. The transformation r is equal to its reverse, making index transformations between the two views as easy as possible. Characters appended in the reverse view will all get negative indices, regardless of the view (suffix or prefix), the first appended character has index 0 and the first prepended character has transformed index 0, too. For example, appending three characters `abc` and then prepending a character `d` will result in the string $t = \texttt{dabc}$ such that $t_{-1} = \texttt{d}$, $t_0 = \texttt{a}$, and so on in the suffix view, and $t^{-1} = \texttt{cbad}$ such that $t^{-1}_{-3} = \texttt{c}$, $t^{-1}_{-2} = \texttt{b}$, and so on in the prefix view.

The CAT for t consists of the suffix structure which is equivalent to CST(t) and the prefix structure equivalent to CST(t^{-1}). To create CAT(ta) from CAT(t), we apply the algorithm described in Section 3.1 (Ukkonen) to the suffix structure and apply the algorithm described in Section 3.2 (Weiner) to the prefix structure. We need to take precautions so that one algorithm does not disturb the other when merged together (the algorithms need to "see" the correct view). Additionally, the path structures must be updated correctly.

To achieve a linear running time, we treat paths as edges. Function 1 shows how prefix nodes clustered in a path can be treated as a single edge (in constant time). All other functions on edges can be implemented for paths in a similar way with constant running times.

Each iteration (a character is appended to t) of the algorithms consists of the following steps:

Step 1. Let p be the prefix parent of $asl_{suffix}(t)$. Remove the prefix edge from $asl_{suffix}(t)$ to p.
Step 2. Lengthen the text, thereby lengthening all open edges.
Step 3. Insert a prefix node for t as a suffix parent of $\overline{\texttt{ta}}$
Step 4. Insert relevant suffixes and update suffix links using $asp_{suffix}(t)$. Restore the prefix structure to include the prefix t.

Step 5. Decanonize $app_{prefix}(t)$ to find $app_{prefix}(ta)$, make its location explicit, and add a prefix edge to the current asl_{suffix}. If the location of $app_{prefix}(ta)$ is a prefix leaf, then the leaf must be deleted because it becomes a non-branching inner node in this step.

4.2. *Detailed Description.* **Step 1** is necessary because the suffix edge is an open edge and will grow when the string is enlarged, while the prefix edge is not an open edge. After this step the prefix structure is still complete except for the prefix for t, which is no longer represented. The step can be easily implemented in constant time.

The underlying string can be implemented as a dynamically growing array. When a character is appended in **Step 2**, all open edges and thus all suffixes of type 1 grow automatically. The prefix link of the active suffix leaf was deleted in the previous step because otherwise $\texttt{depth}(asl_{suffix})$ would be different in the prefix and in the suffix part of the CAT.

Step 3 reinserts the prefix node for t. The prefix edge will be inserted in the next step. We can find the correct location to insert the new node easily, if we keep a reference to the node that represents the complete string (the *end node*). The node will never change and automatically grow by its open edge. It can never be deleted because in Step 5 leaves are only deleted if they become nested. The new prefix node for t is not a suffix node and will be added to a path. If t is not the only prefix leaf representing a proper non-nested prefix of the complete string, there will be a prefix leaf l representing $t_0 \cdots t_{n-1}$ and the new leaf can be added to l's path.

Step 4 is basically the same algorithm as described in Section 3.1. For each new suffix leaf, a prefix edge to the current asl_{suffix} is inserted and the new leaf becomes the asl_{suffix}.

Although the location of asp_{suffix} may be implicit in the suffix structure each time a new node w would be inserted for a CST, there can be an existing node w that represents a left branching substring of t and belongs to the prefix structure. In this case the path that contains w must be split and w must thus be made visible in the suffix structure. This corresponds to splitting an edge and can be implemented in constant time analogously to Function 1.

The newly inserted inner nodes must be threaded in the prefix structure, which corresponds to setting their suffix links. By Lemma 6 there is always an end point (the prefix parent to the last node where a leaf was added) which can be used as a starting point. If new nodes were inserted these belong to the suffix structure only and the corresponding paths must be added or changed.

update-relevant-suffixes(asp_{suffix}, a) will put the traversed nodes on a stack and their suffix links will be set in reverse order while the paths are augmented at the same time. By traversing the nodes in reverse order, we can assure that we are able to grow existing paths.

LEMMA 16. *Let w be a (new) branching node to which a relevant suffix is attached. Then w is a prefix parent or prefix ancestor of t.*

PROOF. Let w be as in the prerequisite. Then w represents a string $s = \texttt{path}(w)$ which is a nested suffix of t. Hence, there is a prefix path from w to t. □

When setting the suffix links and adding nodes to paths, we must check whether old prefix edges existed and if so remove them.

LEMMA 17. *The string* $\texttt{path}(p)$ *represented by the node* p *remembered in Step* 1 *is a suffix of* $\alpha(t)$.

PROOF. p is the prefix parent of $asl_{suffix}(t)$. Therefore, $\texttt{path}(p)$ is a suffix of $\texttt{path}(asl_{suffix}(t))$. Since $\texttt{path}(asl_{suffix}(t))$ is a suffix of t, so is $\texttt{path}(p)$. p is either an inner prefix node or an inner suffix node. In both cases it represents a nested substring of t by Lemma 5. We have $|\texttt{path}(p)| \leq |\alpha(t)|$ because $\alpha(t)$ is the largest nested suffix of t. Hence $\texttt{path}(p)$ is a suffix of $\alpha(t)$. □

Let w be the first node that was either inserted or that resulted from a split path, in which case w was a prefix but no suffix node in CAT(t). It follows that w (representing $\alpha(t)$) is either a descendant of p or equals p. Hence, we insert a prefix edge from w to the node inserted for t in Step 3. If no relevant suffixes were inserted, we add a prefix edge from node p (of Step 1) to the prefix node for t.

While the prefix t was removed from the prefix structure in Step 1 it is now inserted again. The newly inserted inner nodes are in the suffix structure only and are hidden in paths. After this step the prefix structure corresponds to CPT(t) again.

Step 5 inserts the prefix ta and basically corresponds to the algorithm described in Section 3.2. The suffix location is easily found with $asp_{suffix}(ta)$ and so the node can be inserted in the suffix structure, too (possibly in a path). The length of $\hat{\alpha}(ta)$ is the same as the length of $\alpha(ta)$ and the information is taken from the previous step.

If a new prefix node is inserted, it may either be added to a path on its own or to the path of its suffix parent. If the location is a prefix leaf, it must be removed from the path which contains all the leaves before it is deleted.

All suffix leaves are suffixes of ta. They are connected by prefix edges labeled with the first characters of ta: the largest suffix is $t_0 \cdots t_n a$, the second largest suffix is $t_1 \cdots t_n a$. They are connected by a prefix edge labeled with t_0. Hence, the prefix edges connecting the $k + 1$ largest suffix leaves describe the prefix $t_0 \cdots t_k$. All leaves but the the leaf for ta are in a path and not visible in the prefix tree. They are the end of the largest prefix ta. Linking asl_{suffix}, the shortest suffix leaf, to the location $app_{prefix}(ta)$ is equivalent to inserting a new prefix leaf for ta.

Hence, Step 5 correctly updates the prefix structure of the CAT and the resulting tree is CAT(ta).

4.3. *Example Iteration.* Figures 5 and 6 show an example iteration of building CAT($\texttt{acabaabac}$) from CAT($\texttt{acabaaba}$). Figure 5 shows the suffix view and Figure 6 shows the prefix view. The trees correspond to each other. Part (a) shows the CAT for the string $\texttt{acabaaba}$. After Step 1 the prefix link of asl_{suffix} (in the suffix view $\overline{\texttt{aaba}}$) to p (in the suffix view $\overline{\texttt{aba}}$) is removed. The prefix for $t = \texttt{acabaaba}$ is no longer represented, which is shown in (b). Part (c) shows the tree after the string has grown in Step 2. As the character was appended to the right, this has no effect on the prefix structure, while all suffix leaves grow by the open edges. Step 3 reinserts the node for the prefix location of t. The node is added to the path that contains the prefix leaves as shown

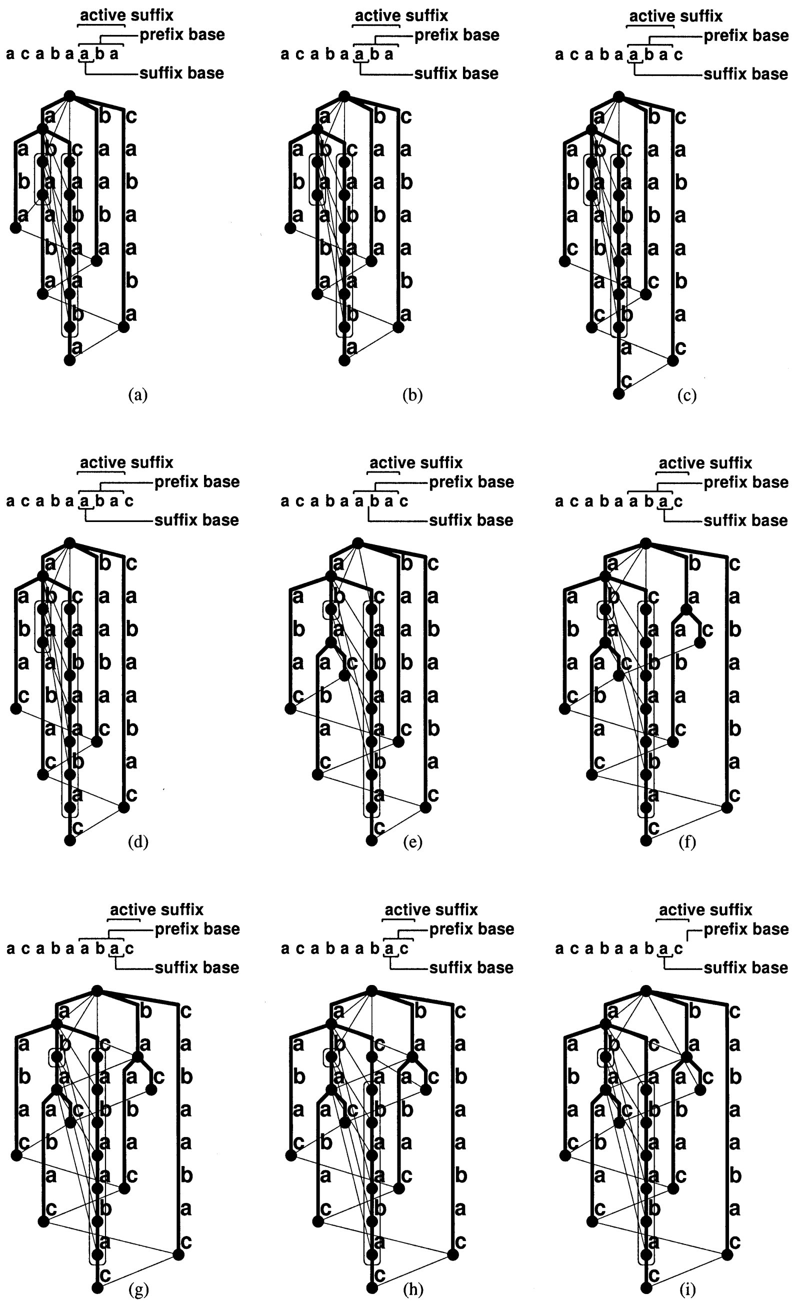


Fig. 5. Steps of an iteration from CAT(acabaaba) to CAT(acabaabac) in the suffix view: (a) CAT(acabaaba), (b) after Step 4.1, (c) after Step 4.1, (d) after Step 4.1, (e) after inserting the first new relevant suffix, (f) after inserting the second new relevant suffix, (g) after Step 4.1, (h) after decanonizing app_{prefix} and inserting the prefix location of ta, and (i) after the prefix leaf has been deleted the resulting tree is CAT(acabaabac).

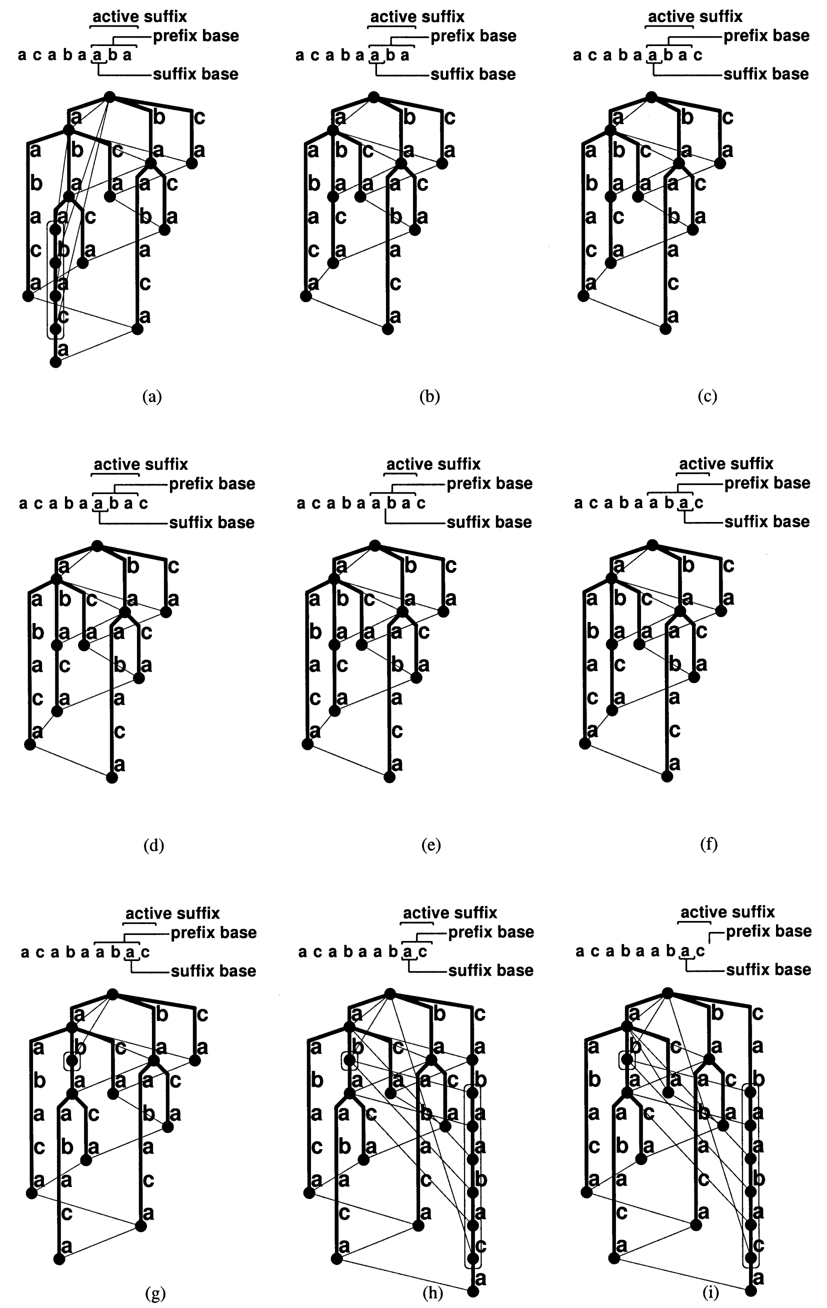


Fig. 6. Steps of an iteration from CAT(acabaaba) to CAT(acabaabac) in the prefix view: (a) CAT(acabaaba), (b) after Step 4.1, (c) after Step 4.1, (d) after Step 4.1, (e) after inserting the first new relevant suffix, (f) after inserting the second new relevant suffix, (g) after Step 4.1, (h) after decanonizing app_{prefix} and inserting the prefix location of ta, and (i) after the prefix leaf has been deleted the resulting tree is CAT(acabaabac).

in (d). The first relevant suffix is inserted, resulting in (e). The branching node $w = \overline{\mathtt{aba}}$ would have been inserted in a suffix tree construction. In the affix tree the node already existed and will only be made a suffix node by removing it from the path. This has no effect on the prefix structure. Part (f) shows the tree after the second relevant suffix has been inserted. A new branching node $w = \overline{\mathtt{ba}}$ was created in the suffix structure. The new node is not a prefix node and is thus added to a path. It appears hidden in a path in the prefix structure after the new nodes have been threaded in from the stack. This can be seen in (g), which shows the tree after Step 4. The reference pair app_{prefix} is decanonized, its base moves from $\overline{\mathtt{aba}}$ to $\overline{\mathtt{a}}$, where a suffix edge leads to $\overline{\mathtt{ca}}$, which becomes the new base. The new length is set to 2 and a prefix edge is inserted to $asl_{suffix} = \overline{\mathtt{cab}}$. The resulting tree is shown in (h). Since $\overline{\mathtt{ca}}$ was a prefix leaf, it needs to be deleted. The resulting tree is CAT(acabaabac) and it is shown in (i).

4.4. *Complexity*

THEOREM 18 (Complexity of the Unidirectional Construction). CAT(t) *can be constructed in an on-line manner from left to right or from right to left in time* $\mathcal{O}(|t|)$.

PROOF. The algorithm is dual, so it does not matter whether we always append to the right or always append to the left. Exchanging the words suffix and prefix in the description of the algorithm does not change it.

With the use of paths (see Function 1 for an example), Step 4 will only work with nodes that would be in the corresponding suffix tree. The other nodes are hidden in paths and will be skipped like edges. Similarly, Step 5 will only work with nodes that would be in the corresponding prefix tree. New nodes are added to paths and the invariant is kept. Therefore, Lemmas 11 and 14 can be applied and both steps take amortized constant time. If we use the index transformation as described in Section 4.1, we do not even need to change the amortized analysis of Lemma 14—the coins will simply be taken from t_{-1-o} (recall that o is the offset of the reference pair app_{prefix}). As o only decreases, $-1-o$ only increases. We always put a coin on a newly added character. Hence, there is always a coin on t_{-1-o}. The other three steps (1–3) take constant time each. As a result, the algorithm runs in linear time. □

5. Bidirectional Construction of Compact Affix Trees.

The topic of bidirectional construction is only briefly mentioned in Stoye's Master's thesis [5]. Besides the need to update the important locations like the reference pair for the active prefix point, additional measures must be taken to achieve a linear running time. Lemmas 19 and 20 can be found in a similar form in [5] and are replicated for clarity and completeness. The importance of Lemma 20 reveals itself only in the context of the amortized analysis for the linear bidirectional construction of CATs.

5.1. *Additional Steps.* The affix tree construction algorithm is dual, so the tree can be constructed by appending characters at the end or at the front of the underlying string. To be able to do this interchangeably, we need to remember the reference pairs for the active prefix point (asp_{prefix} and app_{suffix}), too, and we need to keep them canonical and up-to-date. For this we add a new step to the steps described in Section 4:

Step 6. Update the active prefix.

By appending a character a to the end of t the active prefix $\hat{\alpha}$ cannot become smaller ($\hat{\alpha}$ stays a prefix and if it is nested in t, it is also nested in ta). The active prefix can grow by a character if $t = \{\ \hat{\alpha}a\_\_\_, \_\_\_\hat{\alpha}\ \}$. The following lemma can be used to determine if the active prefix grows.

LEMMA 19. *The active prefix grows by one character iff the prefix location of the new active suffix is represented by a prefix leaf in Step* 5.

PROOF. If the prefix location of the new active suffix is a prefix leaf, then the new active suffix is also a prefix of ta: $ta = \{\ \alpha(ta)\_\_\_, \_\_\_\alpha(ta)\ \}$. Obviously, $\alpha(ta)$ is a nested prefix. Since $\alpha(ta)$ is represented by a prefix leaf in CAT(t), it was a non-nested prefix of t before. Therefore, the active prefix $\hat{\alpha}(t)$ is smaller than $\alpha(ta)$, which is a nested prefix of ta. Hence, $\hat{\alpha}(ta)$ must grow.

If the active prefix grows by a character, its second occurrence in ta must be as a suffix and hence we have $ta = \{\ \hat{\alpha}(ta)\_\_\_\_, \_\_\_\_\hat{\alpha}(ta)\ \}$. $\hat{\alpha}(ta)$ is a nested suffix of ta. Suppose the active suffix would be larger, then $ta = \{\ \hat{\alpha}(ta)\_\_\_\_, \_\_\_\_\hat{\alpha}(ta), \_\_\_\_\alpha(ta), \_\_\alpha(ta)\_\_\ \}$ and $v\hat{\alpha}(ta) = \alpha(ta)$ for a $v \in \Sigma^+$. As a result, we would have $t = \{\ \hat{\alpha}(ta)\_\_\_\_, \_\_v\hat{\alpha}(ta)\_\_\ \}$, a contradiction. Therefore we must have $\hat{\alpha}(ta) = \alpha(ta)$. The active prefix is the longest nested prefix. The next longer prefix is non-nested and therefore represented by a prefix leaf. Hence, if the active prefix grows, it is identical to the active suffix, and it must be represented by a leaf. $\square$

We find the prefix location of the new active suffix in Step 5. We then check explicitly whether it is a leaf (and whether we must delete it) or not. Step 6 is easily implemented by canonizing both reference pairs if no node is deleted in Step 5 (we canonize because nodes might have been inserted), and by lengthening the active prefix when a leaf is deleted.

If the active prefix grows, we must keep its reference pairs up-to-date. To find the location of $asp_{prefix}(ta)$ from $asp_{prefix}(t)$ we must find a suffix edge labeled a to move its base so that the new character can be prepended. This can be done by calling decanonize($asp_{prefix}(t), a$). The location of $app_{suffix}(ta)$ can be found by lengthening $app_{suffix}(t)$ by one character. After that canonize($app_{suffix}(ta)$) must be called to keep the reference pair canonical.

If the active prefix does not grow, we must canonize $asp_{prefix}(t)$ and $app_{suffix}(t)$ because prefix and suffix nodes might have been inserted. Calling canonize at the end of an iteration is equivalent simply to testing whether a newly inserted suffix (prefix) node is the new canonical base of $app_{suffix}(asp_{prefix})$ each time a new node is added. This adds constant overhead to the insertion of nodes, only. Step 6 hence adds the additional overhead of canonizing $app_{suffix}(t)$ and decanonizing $asp_{prefix}(t)$.

5.2. *Enhancing the Running Time.*

If the algorithm is executed as described above, a linear time cannot be guaranteed if characters are appended and prepended in arbitrary order. As an illustration, suppose that after some iterations where characters were appended to the string, characters are now prepended. In such a "reverse" iteration, the active suffix might grow. In Step 6 of the reverse iteration the reference pair $asp_{suffix} = (b, (o, l))$ is decanonized. Thus, its index o into the string is decreased and Lemma 11 cannot be applied anymore. Similar conditions may increase the index o of app_{prefix} and Lemma 14 cannot be applied.

To prevent costly oscillation of the indices and to reduce work to amortized linear time, we must conserve some work. We introduce two enhancements to Steps 5 and 6 that ensure a linear time bound.

TRICK 1. If the new prefix location of the active suffix is already explicit as node n, this node must occur in a path starting below the base b of the $asp_{suffix} = (b, (o, l))$. We can thus get the node in constant time by getting the $(l - 1)$th element in the path starting at the child t_o of b. In Step 5 we will thus decanonize only if the node cannot be found in constant time by the method described above.

In the example shown in Figures 5 and 6, the canonical suffix reference pair $asp_{suffix}(ta)$ after Step 4 (see part (g)) has base $\overline{\texttt{a}}$ and length 1. This location is represented by node $\overline{\texttt{ca}}$ in the prefix structure and can be retrieved in constant time from node $\overline{\texttt{a}}$.

The following lemma will make Step 6 run in constant time altogether.

LEMMA 20. *If the active prefix $\hat{\alpha}(t)$ grows, the new active prefix $\hat{\alpha}(ta)$ is equal to the new active suffix $\alpha(ta)$.*

PROOF. If the active prefix grows, then the original string was $t = \{\ \hat{\alpha}a\_\_\_, \_\_\_\hat{\alpha}\ \}$ and there is no other $\hat{\alpha}a$ anywhere in t (otherwise $\hat{\alpha}a$ is a larger nested prefix—a contradiction to $\hat{\alpha}$ being the active prefix). The new string is $t = \{\ \hat{\alpha}a\_\_\_, \_\_\_\hat{\alpha}a\ \}$. Hence, $\hat{\alpha}a$ is a nested suffix. Suppose there is a larger nested suffix $w = v\hat{\alpha}a$ in ta. Then $ta = \{\ \hat{\alpha}a\_\_\_, \_\_\_v\hat{\alpha}a, \_v\hat{\alpha}a\_\_\ \}$ and $t = \{\ \hat{\alpha}a\_\_\_, \_\_\_v\hat{\alpha}, \_v\hat{\alpha}a\_\_\ \}$. There would be another occurrence of $\hat{\alpha}a$ in t which is a contradiction to $\hat{\alpha}$ being the active prefix of t. As a result, $\hat{\alpha}(ta) = \hat{\alpha}(t)a = \alpha(ta)$. □

TRICK 2. If the active prefix grows, we simply assign $asp_{suffix}(ta)$ to $app_{suffix}(ta)$ and $app_{prefix}(ta)$ to $asp_{prefix}(ta)$. By Lemma 20 this is correct and Step 6 thus runs in constant time.

5.3. *Analysis of the Bidirectional Construction.* Step 6 is made to run in constant time by Trick 2. The cost of canonizing asp_{prefix} and app_{suffix} when the active prefix does not grow can be simply assigned to the insertion cost of a node at no further asymptotic charge. Hence, all that is left to take care of are the side effects a reverse iteration has on later iterations. Since the reference pairs are all kept up-to-date and the upkeeping of asp_{prefix} and app_{suffix} comes for free, the main concern lies on changes of the reference pairs for the active suffix in iterations where characters are appended to the front of the underlying string.

5.3.1. *Overview.* The bidirectional version of the algorithm has four steps that do not run in constant time. These are Steps 4 and 5 in the suffix and in the prefix iteration (procedures canonize() and decanonize()). We will distinguish between a suffix iteration (a "normal iteration") where a character a is appended to t, resulting in ta, and a prefix iteration (a "reverse iteration") where a character is a prepended to t, resulting in at.

We need to show that all calls of canonize() and of decanonize() in the prefix and in the suffix iteration have constant amortized cost. To show this we use accounted amortized analysis and assign each character of t four purses:

- A "suffix/canonize" purse,
- a "suffix/decanonize" purse,
- a "prefix/canonize" purse, and
- a "prefix/decanonize" purse.

In the suffix iterations the cost of a repetition of the while loop in canonize() will be payed from a "suffix/canonize" purse and the cost of a repetition of the while loop in decanonize() will be payed from a "suffix/decanonize" purse. Analogously, in prefix iterations the "prefix/canonize" and the "prefix/decanonize" purses will be used. The purses will be filled when characters are appended or prepended to the string or when new nodes are inserted into the tree (the details will be given further below).

Invariant I. Whenever the body of the while loop of canonize() in a suffix iteration is executed, there is a coin in the "suffix/canonize" purse of character t_o (o is the offset index of asp_{suffix}).

Invariant II. Whenever the body of the while loop of decanonize() in a suffix iteration is executed, there is a coin in the "suffix/decanonize" purse of character t_o (o is the offset index of app_{prefix}).

Invariant III. Whenever the body of the while loop of canonize() in a prefix iteration is executed, there is a coin in the "prefix/canonize" purse of character t_o (o is the offset index of asp_{prefix}).

Invariant IV. Whenever the body of the while loop of decanonize() in a prefix iteration is executed, there is a coin in the "prefix/decanonize" purse of character t_o (o is the offset index of app_{suffix}).

THEOREM 21 (Complexity of Bidirectional Construction of CATs). *If the four invariants are true, then the bidirectional construction of the CAT for the string t has linear running time $\mathcal{O}(|t|)$.*

PROOF. If the above four conditions are met, then each call to one of the procedures has constant amortized cost. In each iteration at most one call to decanonize() is made and for each suffix leaf inserted in a suffix iteration and each prefix leaf inserted in a prefix iteration at most one call is made to canonize(). Since there are at most n iterations, no more than n leaves can be deleted. The final CAT has at most n suffix leaves and at most n prefix leaves. Hence, the number of leaves inserted is $\mathcal{O}(n)$ and there are no more than $\mathcal{O}(n)$ calls to canonize(). □

As a result of the above theorem, each iteration has constant amortized cost.

We will show that even though an arbitrary number of reverse iterations is executed between normal iterations, there will always be a coin in the "suffix/canonize" purse of character t_o when the body of the while loop of canonize in a suffix iteration is executed and there will always be a coin in the "suffix/decanonize" purse of character t_o when the body of the while loop of decanonize in a suffix iteration is executed (Invariants I and II). By the duality of the algorithm this will show that all four invariants are true and that the algorithm is linear in the length $|t|$ of the final string t.

We initialize each purse as follows:

- If a character a is appended to t, then the new string is $t_0 \cdots t_n t_{n+1}$ $(a = t_{n+1})$ and we put a coin in the "suffix/canonize" purse of t_{n+1} and in the "suffix/decanonize" purse of t_{n+1}.
- If a character a is prepended to t, then the new string is $t_{-1} t_0 \cdots t_n$ $(a = t_{-1})$ and we put a coin in the "prefix/canonize" purse of t_{-1} and in the "prefix/decanonize" purse of t_{-1}.

If we solely perform suffix iterations, the offset index o of asp_{suffix} is only increased and the offset index o' of app_{prefix} is only decreased (so that the transformed index $-1 - o'$ is increased only). If no reverse iterations are performed, only Invariants I and II or III and IV are relevant, and Lemma 22 tells us that they will always be true. Therefore, the construction is linear in time.

LEMMA 22. *If the offset index o of asp_{suffix} and of asp_{prefix} only increases, Invariants* I *and* III *will always be true.*

If the offset index o' of app_{prefix} and of app_{suffix} only decreases, Invariants II *and* IV *will always be true.*

PROOF. Characters appended to the underlying string appear at higher indices than the index offset o of asp_{suffix} and at lower indices (in the reverse view of the string) than the index o' of app_{prefix}. Similarly, characters prepended to the underlying string appear at higher indices (in the reverse view of the string) than the index offset o of asp_{prefix} and at lower indices than the index o' of app_{suffix}. □

If intermediate prefix iterations are performed, the offset indices of asp_{suffix} and app_{prefix} can decrease (respectively increase). We have to prove that the additional coins needed can be put into the purses with amortized constant, extra cost. The remaining two sections will show how this is done and complete the proof.

5.3.2. *Changes to asp_{suffix} in a Reverse Iteration—Invariant* I. Changes to asp_{suffix} concern Invariant I only. The following results can be applied analogously to asp_{prefix} (by exchanging suffix and prefix) and Invariant III.

LEMMA 23. *Assigning $app_{suffix}(at)$ to $asp_{suffix}(ta)$ in Step* 6 *adds amortized constant overhead to successive calls to canonize(asp_{suffix}) and keeps Invariant* I.

PROOF. In a prefix iteration $t_{-m} \cdots t_n$ to $t_{-m-1} \cdots t_n$ (the character $t_{-m-1} = a$ was prepended), the active suffix may grow by one. Although asp_{suffix} is not decanonized explicitly (Trick 2), it is changed as if it had been decanonized. Let b^s_{new} be the new base of asp_{suffix} and let b^s_{old} be the old base, then there is a suffix node v such that v is the prefix parent of b^s_{new} (the prefix edge is labeled as the prepended character a) and b^s_{old} is a suffix descendant of v. (See Figure 6(g), where the active prefix grows from $\mathtt{a}$ to $\mathtt{ac}$. We have $b^s_{old} = \overline{\mathtt{aba}}$, $v =$ root, and $b^s_{new} =$ root because of the special case in decanonize.) Let $V_s = \{v_1, \ldots, v_k\}$ be the nodes that are on the path from v to b^s_{old} ($V_s = \{\overline{\mathtt{a}}\}$ in our example). In the next call to canonize() in a suffix iteration the nodes in V_s have to be traversed additionally. The purses that are expected to have a coin are assigned to the characters with the indices $I_s = \{i_j \mid i_j = n - |\alpha(t_{-m} \cdots t_n)| + \mathtt{depth}(v_j) + 1, v_j \in V_s\}$. The coins will come from the reverse iteration.

In the reverse iteration asp_{suffix} was changed by an assignment from app_{suffix}. Usually, app_{suffix} is decanonized in Step 5 of the reverse iteration. In the case that the active suffix grows in a reverse iteration, the suffix location of app_{suffix} is represented by a suffix leaf (by Lemma 19). The node is found in constant time by the use of Trick 1. Hence, there are coins, which are not used, in "prefix/decanonize" purses. (In the example, these are the coins that would be needed to decanonize over $\overline{\mathtt{a}}$.) Let b^p_{old} be the base of $app_{suffix}(t_{-m} \cdots t_n)$ and let b^p_{new} be the base of $app_{suffix}(t_{-m-1} \cdots t_n)$. We know that $b^p_{new} = b^s_{new}$ and, by Corollary 10, we have $|\alpha(t)| + 1 = |\alpha(ta)| = |\hat{\alpha}(ta)| \le |\hat{\alpha}(t)| + 1$, so $|\alpha(t)| \le |\hat{\alpha}(t)|$ and b^p_{old} is a descendant or equals b^s_{old}. Therefore, if we had decanonized app_{suffix}, the nodes V_p ($V_p = \{\overline{\mathtt{a}}\}$ in the example) we would have met in decanonize() would be a superset of the nodes in V_s ($V_s \subseteq V_p$). The string indices are $I_p = \{i_j \mid i_j = -m + |\hat{\alpha}(t_{-m} \cdots t_n)| - \mathtt{depth}(v_j) + 1, v_j \in V_p\}$. Obviously, $|I_p| > |I_s|$ and since the coins were not needed for decanonizing app_{suffix}, we can take the money from the "prefix/decanonize" purses of the characters with indices I_p and put one coin in the "suffix/canonize" purse of each character with an index in I_s. In the analysis of decanonize(), we assume that the coin from the characters with indices I_p were used and so we can take them without scruples. Invariant I is kept, the additional work can thus be payed for. □

Further prefix iterations that occur before canonize is called in a suffix iteration can lengthen the active suffix again, leading to the same money transfer. Additionally, they can insert new nodes.

LEMMA 24. *Only constant work is needed to keep Invariant* I *when adding additional suffix nodes in prefix iterations.*

PROOF. Let b^s_{new}, b^s_{old}, v be the nodes of a previous prefix iteration $t_{-m} \cdots t_n$ to $t_{-m-1} \cdots t_n$ as described above in the proof of Lemma 23, where the active suffix has grown and the active suffix has not been canonized yet. For each node w that is inserted between v and b^s_{old} an additional repetition of the while loop is needed in a successive call to canonize() (V_s grows). The index of the character with the "suffix/canonize" purse where a coin has to be inserted is $i_w = n - |\alpha(t_{-m} \cdots t_n)| + \mathtt{depth}(w) + 1$.

The new suffix node cannot lie between two different b^s_{old}, v pairs. Suppose b'^s_{old}, v' are the nodes of the next case where the active suffix was lengthened with $b^s_{new} = b'^s_{old}$. A node w between b'^s_{old}, v' and b^s_{old}, v must have depth d with $d > \mathtt{depth}(v) = \mathtt{depth}(b^s_{new}) - 1 = \mathtt{depth}(b'^s_{old}) - 1$ and $d < \mathtt{depth}(b'^s_{old})$, which is not possible.

There is at most one suffix node inserted per reverse iteration, so paying the additional coin only adds constant extra cost. □

Note that suffix iterations cannot insert any such nodes because all nodes are inserted above or at the base of the asp_{suffix}. The following lemma can be proven trivially.

LEMMA 25. *Canonizing* $asp_{suffix}(ta)$ *during prefix iterations keeps Invariant* I.

As a result, Invariant I is always kept valid.

5.3.3. *Changes to* app_{prefix} *in a Reverse Iteration—Invariant* II. Changes to app_{prefix} concern Invariant II only. The following results can be applied analogously to app_{suffix} (by exchanging suffix and prefix) and Invariant IV.

By Lemma 22, the invariant might be destroyed if the offset index o of app_{prefix} is increased (in the prefix view of the string).

LEMMA 26. *Assigning* $asp_{prefix}(at)$ *to* $app_{prefix}(ta)$ *in Step* 6 *adds amortized constant overhead to successive calls to* $decanonize(app_{prefix})$ *and keeps Invariant* II.

PROOF. Let $t = t_{-m} \cdots t_n$. All prefix nodes that might have been added during Step 4 in this prefix iteration (if any) are added at a depth larger than the length of $\alpha(at)$. Assigning $asp_{prefix}(at)$ to $app_{prefix}(ta)$ is equivalent to increasing the length of $app_{prefix}(t)$ by one. If $app_{prefix}(t)$ was canonical with base b_{old}, then the base b_{new} of the canonical reference pair $app_{prefix}(ta)$ is either the same as b_{old} or a child of b_{old}. In the later case $\texttt{depth}(b_{new}) = |\alpha(at)|$, because there cannot be a node w with $\texttt{depth}(w) \leq |\alpha(t)|$, if $app_{prefix}(t)$ was canonical. We can put a coin in the "suffix/decanonize" purse of t_n to keep Invariant II. Since we need to put at most one additional coin in a "suffix/decanonize" per iteration, this adds at most constant overhead. □

We will look at canonizing app_{prefix} when the active suffix does not grow. We only need to canonize if new prefix nodes are inserted. The canonization can be seen as taking place each time a new node is inserted. For each new node we check whether the node is the new canonical base for app_{prefix}. The following lemma deals with all inserted nodes.

LEMMA 27. *Only amortized constant work is needed to keep Invariant* II *when adding additional prefix nodes in prefix iterations.*

PROOF. Let o_{min} be the minimal value that the offset index ever had in all previous iterations. For the node v let i_v be the index with which it would appear in a call to decanonize() (in a suffix iteration) $i_v = n - \texttt{depth}(v) - 1$ (the transformed index for the prefix view is $-1 - i_v = \texttt{depth}(v) - n$, $n - \texttt{depth}(v) - 1$ is the real index). Let s be the node with the largest string depth, such that $-1 - i_s < o_{min}$ and let b be the current base of app_{prefix}.

A node v inserted below s will not destroy Invariant II, because there is still the originally inserted coin on character t_{i_v}. We need to provide extra money for the nodes inserted above s to keep the invariant. Let v be inserted above s and below b with

$-1 - i_v \geq o_{min}$. We will add a coin to the "suffix/decanonize" purse of character t_{i_v} (prefix view of the string). If v is inserted above b so that app_{prefix} is canonized to the new base $b' = v$, we will add a coin to the "suffix/decanonize" purse of character t_{i_b} and v will be the new base. By the above accounting we now have the situation that below o_{min} there is a coin in every characters purse and between o_{min} and o there is a coin in each purse for a character corresponding to a node.

When decanonize() is called on app_{prefix} and we move over a node v in the while loop and o is still larger than o_{min}, the above payment guarantees that there is a coin in the "suffix/decanonize" purse of character t_o. We will prove this by induction:

Base case. We have not yet moved over a prefix edge. In this case we meet the exact same nodes that we have added the coins for to the corresponding characters.

Inductive case. We have moved over a prefix edge before. We have $i_v = n - \mathtt{depth}(v) - 1$ and v is a prefix node. Let v' be the suffix parent of v, by Lemma 6 there is a suffix parent and $\mathtt{depth}(v) = \mathtt{depth}(v') + 1$. By induction there is a coin in the "suffix/decanonize" purse of character $i_{v'}$. The index is $i_{v'} = n' - \mathtt{depth}(v') - 1$, where n' was the rightmost string index in the step when the base was moved along the prefix edge the last time.

The base of app_{prefix} is moved over a prefix edge only if a character is appended. Hence, $n' = n - 1$ and $i_{v'} = n - 1 - \mathtt{depth}(v') - 1 = n - 1 - (\mathtt{depth}(v) - 1) - 1 = n - \mathtt{depth}(v) - 1$. Hence, there is a coin in the "suffix/decanonize" purse of character $t_{i_{v'}}$.

We insert at most $2n$ prefix nodes in reverse iterations, thus we only pay an amortized constant amount of additional coins. □

Note again, that suffix iterations cannot insert any such prefix nodes because all prefix nodes are inserted above or at the base of the app_{prefix}.

As a result, Invariant II is always kept valid.

6. Conclusion. Suffix trees are widely known and intensively studied. A less known fact is the duality of suffix trees with regard to the suffix links that is described in [3] and is implicitly stated in [1] and other papers.

The basic affix tree data structure was first defined by Stoye, but the resulting algorithm seems to be non-linear [5]. The introduction of paths improved the algorithm and a linear behavior could be proven. Some further modifications are necessary to achieve linear time in a truly bidirectional construction.

Affix trees (especially augmented by paths) have the same properties as suffix trees and more. The construction is bidirectional and on-line. Search strings can be extended in both directions. In affix trees, not only the suffixes but also the prefixes of a string are given, and the tree view can be switched from suffixes to prefixes at any time and node, which cannot be achieved with two single suffix trees. It is unclear whether there exists a method of parallel suffix tree construction such that nodes representing the same string can be associated. At any position in the affix tree one can tell immediately whether the represented string is left branching or right branching or both.

References

[1] A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *J. Assoc. Comput. Mach.*, 34(3):578–595, July 1987.

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*, 1st edition. The MIT Press, Cambridge, MA, 1990.

[3] R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: a unifying view of linear-time suffix tree construction. *Algorithmica*, 19:331–353, 1997.

[4] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. Accoc. Comput. Mach.*, 23(2):262–272, April 1976.

[5] J. Stoye. Affixbäume. Master's thesis, Universität Bielefeld, May 1995.

[6] J. Stoye. Affix Trees. Technical Report 2000-04, Universität Bielefeld, Technische Fakultät, ftp://ftp.uni-bielefeld.de/pub/papers/techfak/pi/Report00-04.ps.gz, 2000.

[7] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.

[8] P. Weiner. Linear pattern matching. In *Proceedings of the IEEE 14th Annual Symposium on Switching and Automata Theory*, pages 1–11. IEEE, New York, 1973.