Linear Bidirectional On-Line Construction of Affix Trees

Moritz G. Maaß

TU München, Fakultät für Informatik
maass@informatik.tu-muenchen.de

Abstract. Affix trees are a generalization of suffix trees that is 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 has linear time complexity.

1 Introduction

Suffix trees are a widely studied and well known data structure. They can be applied to a variety of text problems and have become more popular with their application in problems related to molecular biology. There exist many linear algorithms for their construction [Ukk95], [McC76]. They all make use of auxiliary edges called suffix links. Usually, when taking edges in the suffix tree, the string represented by the current node is lengthened or shortened at the end. 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 all suffixes can be sped up greatly by the use of suffix links.

As proven by Giegerich and Kurtz in [GK97], there is a strong relationship between the suffix tree built from the reverse string (often called reverse prefix tree) and the suffix links of the suffix tree built from the original string.

Obviously, the suffix links form a tree by themselves (with edges pointing to the root, see Figures 1c, 1d, and 2). Giegerich and Kurtz have shown that the suffix links of the atomic suffix tree (AST - sometimes also referred to as suffix trie) for a string t are exactly the edges of the AST for the reverse string t^{-1} (i.e., the reversed suffix links labeled with the letters they represent and the same nodes are the AST for the reverse string). This property is partially lost when turning to the construction of the compact suffix tree (CST - most often referred to simply as suffix tree). The CST for a string t contains a subset of the nodes of the CST for the reverse string (which we will call compact prefix tree (CPT)) and the suffix links of the CST represent a subset of the edges of the CPT.

A similar relationship was already shown by Blumer et al. in [BBH$^+$87] 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,

Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching, June 2000, Montreal, Canada

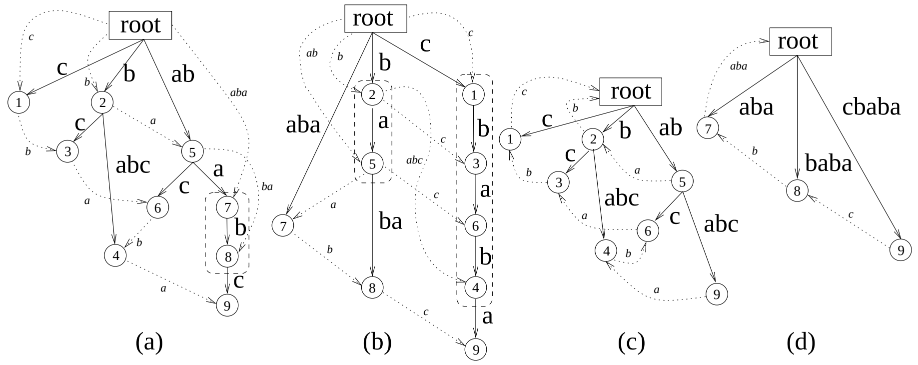

Fig. 1. Affix tree for `ababc` in suffix (a) and prefix (b) view (dotted lines represent the prefix edges, dashed boxes mark the paths) and suffix tree for `ababc` (c) and `cbaba` (d) (dotted lines represent the suffix links).

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.

The goal of this paper is to show how a tree that incorporates both the CST and the CPT can be built on-line in linear time by appending letters in any order at either side of the underlying string. This tree will be called compact affix tree (CAT). It has been introduced by Stoye in his Master's thesis [Sto95] (an English translation is available [Sto00]). Our algorithm essentially has the same steps as Stoye's algorithm, but we will use additional data collected in paths (see section 2.2 and Figures 1a and 1b) to assure a suffix-tree-like view. Stoye could show that the CAT takes linear space, but he was unable to show linear time complexity of his algorithm. 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 similar to Weiner's algorithm [Wei73]).

The core problem for achieving linear complexity is that Ukkonen's algorithm (or any other suffix tree algorithm) expects suffix links of inner nodes to be atomic (i.e., having length one). As can easily be seen in Figures 1a, 1b, and 2, this is the case for suffix trees, but not for affix trees (nodes 7, 8 in Figure 1a and 4, 5 in Figure 1b). Stoye tried to solve this problem by traversing the tree to find the appropriate atomic suffix links. In general, this approach might lead to quadratic behavior, which is the reason why Stoye could not prove linear time complexity of his algorithm. We will use additional data collected in paths to establish a view of the CAT, as if it were the corresponding CST (this can be seen comparing Figures 1a, 1b with Figures 1c, 1d).

We will 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 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. We finish with a section on the implementation of paths.

2 Data Structures

In the following, let Σ be an arbitrary finite alphabet. Let Σ^* denote the set of all finite strings over Σ, let λ be the empty string and let $\Sigma^+ = \Sigma^* \setminus \{\lambda\}$ denote the set of all non-empty strings over Σ. If $t \in \Sigma^*$ is a string, then let $|t|$ be its length and let t^{-1} be the reverse string. If $t = uvw$ for any $u, v, w \in \Sigma^*$, then u is a *prefix*, w a *suffix*, and v a *substring* of t. A suffix w of t is called *nested* if $t = uwv$ for $v \in \Sigma^+$ and $u \in \Sigma^*$ (nested prefix is defined analogously). A substring w of t is called *right branching* if $a, b \in \Sigma$ are two distinct characters ($a \neq b$) and $t = uwav$ and $t = u'wbv'$ for any $u, u', v, v' \in \Sigma^*$ (left branching is defined analogously).

2.1 Trees

We will define the affix tree data structure in terms of [GK97] beginning with Σ^+-trees.

Definition 1 (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 letters. A Σ^+-tree is called* compact*, if all nodes other than the root are either leaves or branching nodes.*

We represent the edge labels by means of pointers into the underlying string, thus keeping the size for each label constant. Each path from the root to an arbitrary node n in T represents a unique string $w = \mathsf{path}(n)$. If $u = \mathsf{path}(n)$, we can identify the node by the string and we will write $\overline{u} = n$. A string $u \in \Sigma^*$ is in $words(T)$ iff there is a node n in T with $\mathsf{path}(n) = uv$ with $v \in \Sigma^*$. If a string u is represented in T by a node n (i.e., $u = \mathsf{path}(n)$), we call the location of the string explicit. Otherwise, if the string u is represented but not by a node (i.e., there exists a node n and a string $v \in \Sigma^+$ s.t. $uv = \mathsf{path}(n)$), we say the location of the string is implicit.

A *suffix link* is an auxiliary edge from node n to node m where m is the node s.t. $\mathsf{path}(m)$ is the longest proper suffix of $\mathsf{path}(n)$ represented by a node in T. Suffix links can be labeled similarly to edges, but with the string label reversed (see Figure 1). For example, a suffix link leading from a node representing the string `cbab` to a node representing the string `b` will be labeled `abc`.

Definition 2 (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 is reversed, but the label is kept.*

Definition 3 (Suffix tree). *A suffix tree of string t is a Σ^+-tree with $words(T) = \{u|u \text{ is a substring of } t\}$.*

As stated above, Giegerich and Kurtz have proven in [GK97] that the AST T for the reverse string t^{-1} is the same as the reverse AST T^{-1} for the string t.

Affix trees can be defined as 'dual' suffix trees.

Definition 4 (Affix tree). *An affix tree T of a string t is a Σ^+-tree s.t.*
$words(T) = \{u|u \text{ is a substring of } t\}$ and
$words(T^{-1}) = \{u|u \text{ is a substring of } t^{-1}\}$.

Note that a string might have two different locations in CAT if it is not explicit. The following lemma presents an easily verifiable fact about CSTs.

Lemma 1. *In the CST for string t, each node represents either the empty string (the root), a right branching substring, or a non-nested suffix of t.*

The CAT contains all nodes that would be part of the CST and the CPT. With this lemma, each node can easily be checked to belong either to the suffix part, the prefix part, or both, thus defining a node type. For a node to be a suffix node it needs to be the root or have zero or more than one suffix child (because it is a leaf or because it is branching). The attribute needs not be saved explicitly but can be reconstructed on-line. Edges are either suffix or prefix edges. In CATs the suffix links are the (reversed) prefix edges and vice versa. All edges need to be traversable in both directions.

Compare Figures 1a and 1b with Figures 1c and 1d to see the differences between CAT and CST. In the following let CAT(t) denote the CAT for the string t (CST(t), CPT(t) analogously).

2.2 Additional Data for The Construction of CAT

Let $\alpha(t)$ be the longest nested suffix of string t and $\hat{\alpha}(t)$ the longest nested prefix of t. There are some important locations in CAT(t) that are needed for the building process. These locations are (the string arguments are dropped if they are clear from context):

- The active suffix point $asp(t)$, which represents the suffix location of the longest nested suffix (the active suffix) $\alpha(t)$.
- The active suffix leaf $asl(t)$, which represents the shortest non-nested suffix of t.
- The active prefix point $app(t)$, which represents the prefix location of the longest nested prefix (the active prefix) $\hat{\alpha}(t)$.
- The active prefix leaf $apl(t)$, which represents the shortest non-nested prefix of t.

The *asl* has a special role regarding suffix links.

Lemma 2. *In $CST(t)$ all suffix nodes except for the asl and the root have atomic suffix links.*

The proof should be obvious and follows from Lemma 1.

To keep track of *asp* and *app*, which might both be implicit, we will use *reference pairs* (rps) (similarly defined by Ukkonen [Ukk95]). A rp is a pair (n, u) where n is a node (called the base of the rp), $u \in \Sigma^*$ is a string, and $\mathsf{path}(n)u$ is the represented location. Since there are always two implicit locations of any string not represented by a node in CAT (e.g. bab in Figure 1a and 1b), the rps will have a type. A suffix rp will refer to a suffix location (which might be implicit in a suffix edge) and use a suffix node as base. With *canonical reference pair* (crp) we will refer to a rp (n, u) that is closest to the represented location while still being of the referenced type (i.e., if (n, u) is a suffix crp, then n is a suffix node and there exists no rp (n', u') for the same location with $|u'| < |u|$ and where n' is also a suffix node). For *asp* a suffix crp and for *app* a prefix crp is kept during all iterations.

By Lemma 2, all nodes that belong to the suffix part are connected by atomic suffix links. If they belong exclusively to the suffix part, they cannot be branching in the prefix part and can therefore be combined to clusters of "suffix only" nodes connected with atomic prefix edges in the prefix part. Each such cluster will be combined in a *path* (*asl* might have a non-atomic edge, but that will be dealt with in section 6). See the dashed boxes marking the paths in Figure 1a and 1b. The paths contain all nodes that would not be part of the corresponding CST (compare Figures 1a and 1b with 1c and 1d). The same holds true for nodes that only belong to the prefix part. The paths allow the following operations in constant time, thus behaving like edges:

- Access the first and the last node from either side.
- Access the i-th node.
- Split the path similar to edges at a given position (e.g. making node 6 of Figure 1b a prefix node).
- Append or prepend a node to a path.

With these paths, the CAT can be traversed ignoring nodes of the reverse kind: if a prefix only node is hit while traversing the suffix part, it must have a pointer to a path. The last node of that path is again followed by a suffix node. When taking a suffix view and using paths like edges the CAT behaves just like a CST. This is the key to solving the problem of Stoye's algorithm and to achieve linear time.

We will also use *open edges* as introduced by Ukkonen [Ukk95]. 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.

3 On-Line Linear Construction of Compact Suffix Trees

We will briefly describe how to construct a CST by expanding the underlying string to the right and how to construct a CST by expanding to the left. To solve these problems we can make use of two already existing algorithms, namely Ukkonen's algorithm [Ukk95] and Weiner's algorithm [Wei73]. Weiner's algorithm is rather complicated, so we will assume some additional information readily available in the later algorithm for constructing CATs.

3.1 Normal On-Line CST Construction

Ukkonen's algorithm constructs a CST on-line from left to right, building CST(ta) from CST(t). It essentially works by lengthening all suffixes represented in CST(t) by the character a and inserting the empty suffix. It should be obvious that the resulting tree is CST(ta). To keep the linear time bound two ideas help: Through the use of open edges the suffixes representing leaves are automatically lengthened. A reference to the longest represented nested suffix (*asp*) is always kept by means of a crp. It is then checked whether the lengthened suffix is still represented in the tree. If not, the suffix is not nested any more and a leaf is inserted. Such a suffix is called a *relevant suffix* (every non-nested suffix sa of string ta is a relevant suffix iff s is a nested suffix of t). After that the next shortest nested suffix is examined. As soon as one suffix is still nested after lengthening it by a all shorter suffixes will also stay nested and the update is complete. The location of that first nested suffix s of t, where sa is also nested in ta, is called the *endpoint*. sa is then $\alpha(ta)$, the new active suffix. To reach the next shortest suffix the base of the crp is replaced by the node its suffix link points to. After that the new rp is canonized by moving the base down the tree as far as possible thus shortening the string part. The number of nodes traversed in this way can be estimated by the length change of the string part leading to an overall linear complexity.

Theorem 1. *Ukkonen's algorithm constructs CST(t) on-line in time $\mathcal{O}(|t|)$.*

The proof and the algorithmic details can be found in [Ukk95]. An example of one iteration can be seen in Figure 2c and 2d. The procedure for one iteration as described above will be called update-suffix.

3.2 Anti-On-Line CST Construction with Additional Help

We will describe how to build CST(at) from CST(t). CST(t) already represents all suffixes of CST(at) but at. The leaf for at will branch at the location representing $\hat{\alpha}(at)$ in CST(t) (which is *app*(at)). 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 (see Lemma 1).

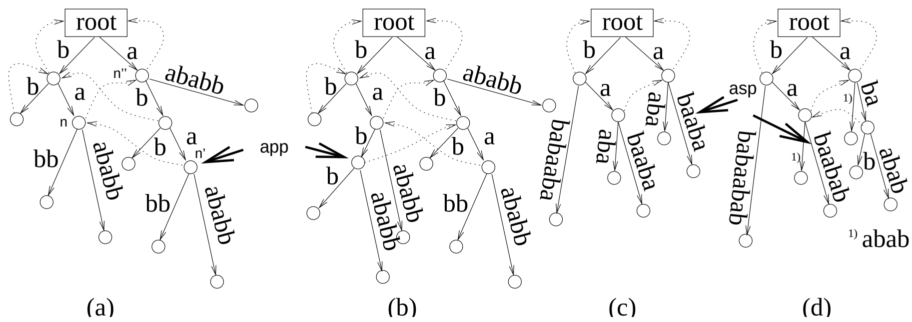

Fig. 2. CST(abaababb) and CST(babaababb) with *app* marked and CST(bbabaaba) and CST(bbabaabab) with *asp* marked.

See Figure 2a and 2b for an example of one iteration where a new branching node is inserted. To find the location of $app(at)$ let (n, u) be a crp for $app(at)$ and (n', u') a crp for $app(t)$ in CST(t). $app(at)$ needs to be found in CST(t). If n is not the root, it has a suffix link to some node n'' that is on the path from n' to the root. Hence n can be found starting from n' by moving up the tree until a node n'' is found that is target of an atomic suffix link that lengthens the represented string by a at the front (this means traversing the suffix link backwards). These three nodes are marked in Figure 2a. If we knew the final length of $\hat{\alpha}(at)$, we could easily determine the length of u. Because we will later be constructing CATs we allow the additional information of knowing that length. The length is also used to identify the node that might be target of an additional suffix link for a possibly inserted branching inner node.

Clearly, $|\hat{\alpha}(t)| + 1 \geq |\hat{\alpha}(at)|$ and there exists a string $v \in \Sigma^*$ s.t. $\hat{\alpha}(t) = vv'$ and $\hat{\alpha}(at) = av$ with $v' \in \Sigma^*$. The number of traversed nodes is limited by the length of the active prefix, so the following theorem is established easily.

Theorem 2. *With the additional information of knowing the length of $\hat{\alpha}(s)$ for any suffix s of t before inserting it, it takes $\mathcal{O}(|t|)$ time to construct CST(t) in an anti-on-line manner.*

Proof. Inserting the leaf, inserting an inner branching node, or deleting an old leaf takes constant time per step. To find the base n_i of the crp for the new *app* from the old base n_{i-1}, a number of nodes need to be traversed. For each node encountered the string represented by the new base will be at least one character shorter. Taking the reverse suffix link lengthens the path by one. If s_i is the number of nodes encountered in iteration i, the following inequality therefore holds:

$$|\mathsf{path}(n_i)| \leq |\mathsf{path}(n_{i-1})| - s_i + 1 \Leftrightarrow s_i \leq |\mathsf{path}(n_{i-1})| - |\mathsf{path}(n_i)| + 1 \qquad (1)$$

Thus the total number of nodes traversed is $\sum_{i=1}^{|t|} s_i \leq \sum_{i=1}^{|t|} |\mathsf{path}(n_{i-1})| - |\mathsf{path}(n_i)| + 1 \leq |t| + |\mathsf{path}(n_0)| - |\mathsf{path}(n_{|t|})| \leq |t| = \mathcal{O}(|t|)$ □

A more detailed discussion of Weiner's algorithm using the terms of Ukkonen's algorithm can be found in [GK97]. The procedure for one iteration as described above will be called update-prefix.

4 Constructing Compact Affix Trees On-Line

Obviously, the algorithm is dual. The necessary steps to construct CAT(ta) from CAT(t) are the same as for constructing CAT(at) from CAT(t). For clarity we will assume the view of constructing CAT(ta) from CAT(t). We call this a suffix iteration, while CAT(at) from CAT(t) is called a prefix iteration. The part representing CST(t) will be referred to as suffix part and the part representing CST(t^{-1}) as prefix part.

We will merge the above given algorithms (section 3) to get the final algorithm for the construction of CATs. Remember that both algorithms rely heavily on the fact that all suffix links are atomic. With the use of paths we can establish a CST-like view, which guarantees atomic suffix links by Lemma 2. When inserting new nodes that belong only to one part it is important to add them to paths (either new or existing ones). Given the above path properties (see section 2.2), this will pose no problem. Another modification common to both parts is the use of paths and of suffix and prefix crps so that the CAT behaves like a CST (respectively CPT) for each algorithm part.

To update the suffix part we apply a modified update-suffix. Except for the general modifications the leaves need to get suffix links, too. This is where the *asl* is needed. For each new leaf l, *asl* will get a suffix link to l, and l will become the new *asl*. *asl* initially has a suffix link to some node s. The link will be removed before lengthening the text since *asl* will "grow" because of the open edge leading to it and its suffix link cannot grow, leading to an inconsistency. Nodes s and *asl* will be used for the next part.

To update the prefix part we apply a modified update-prefix, where – in order not to switch the view – we talk of prefixes and not of suffixes (replace suffix by prefix and vice versa in section 3.2). Recall that the prefix edges are the reversed suffix links. Note that the suffix leaves form a chain connected by suffix links that represent a prefix of t and hence also of ta (see Figure 1a and 1b). After update-suffix they still form a chain representing a (possibly longer) prefix of ta, thus representing part of the prefix that needs to be inserted in the prefix part. By removing the suffix link from *asl* to s, the prefix location for t was lost. The *asl* has not been relinked yet. Hence the prefix location for t needs to be reinserted and *asl* needs to be suffix linked correctly to form the prefix location for ta. The first step is obvious because we have node s and we know that the prefix leaf for t is suffix parent to the prefix leaf for ta (see Figure 1; the node representing the whole string is easily kept globally, since it never changes due to the open edges). The second step involves the same search as in update-prefix to find the prefix location of *asp*(ta). The additional information is easy to give because we already know the length of $\alpha(ta)$ from update-suffix. Furthermore, it may happen that we take a starting point that is above the base n' for the crp for $\alpha(t)$ in the prefix part because we have inserted relevant suffixes. In Figure 2d the endpoint of update-suffix is $\overline{ba}$, which is $\overline{ab}$ in Figure 2a and lies between n' and n''. In case the location of $\alpha(ta)$ is already an explicit prefix node we can even omit the search, since it will be the $|u|$-th element of the path starting after the base n (or n if $|u| = 0$), if (n, u) is the crp for *asp*(ta) (see Figure 3d, where the root is the base for the crp to *asp*, and nodes 2, 4, and 6 form a path starting at the root). Section 4.1 describes how the new starting point is found.

Because the algorithm is bidirectional we need to update *app* and *apl*. *apl* only changes if a new "shortest" leaf is inserted (i.e., if the previous *apl* was t, which is only the case if $t = a^{\mathrm{k}}$ for $a \in \Sigma$), or if *app* grows. *app* might change in accordance with a lemma by Stoye [Sto95] (Lemma 4.8):

Lemma 3. *The active prefix will grow in the iteration from t to ta iff the new active suffix* $\alpha(ta)$ *is represented by a prefix leaf in* $CAT(t)$.

That is the case when a node needs to be deleted.

These are essentially the same steps that Stoye's algorithm performs. (The reason behind each step is more elaborately described in [Sto95], [Sto00].) The important difference is that our algorithm makes use of paths (as described in section 2.2 and section 6) in order to skip over prefix nodes when updating the suffix part and vice versa. In summary, the six necessary steps of each iteration of the algorithm are:

1. Remove the suffix link from *asl* to s.
2. Lengthen the text, thereby lengthening all open edges.
3. Insert the prefix node for t as suffix parent of $\overline{ta}$ and link it to s.
4. Insert relevant suffixes and update suffix links (update-suffix).
5. Make the location of the new active suffix $\alpha(ta)$ explicit and add a suffix link from the new *asl* to it (main part of update-prefix).
6. Update the active prefix, possibly deleting a node.

Step 4 is nearly the same as update-suffix, so we will clarify the other steps by means of a small example. Figure 3 shows the CAT for `baa` (a), the tree after step 2 (b) (node 3 is the *asl*, node 5 is s, which is only accidentally the same as *asp*), the tree after step 3 (c) (node 6 was inserted to represent the prefix `baa` and it was linked to s), the tree after step 5 (d) (node 7 was inserted for the relevant suffix `ab`, the root was the endpoint, *asp* was already an explicit prefix node, it is node 2, the new *asl* is node 7, it was prefix linked to node 2), and finally the new CAT for `baab` after step 6 (e) (node 2 had to be deleted because it is neither part of CST nor of CPT for `baab`, which can easily be seen because it has exactly one incoming and one outgoing edge of either type and it was a leaf before). `b` is now the active prefix and the active suffix.

4.1 Finding the New Starting Point for the Search for the Prefix Location of $\alpha(ta)$

If a starting point is needed it will always be a prefix node that represents the string u (where $ua = \alpha(ta)$), which is the endpoint of that iteration. The following lemma will be helpful:

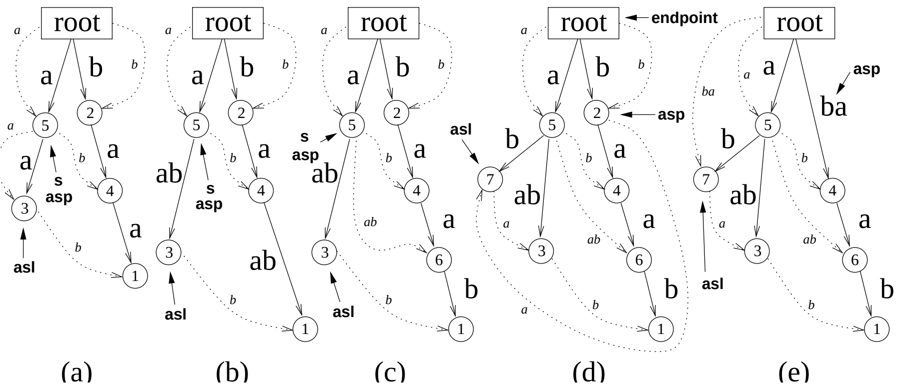

Fig. 3. Step-by-step construction of affix tree for `baab` from affix tree for `baa` in suffix view.

Lemma 4. *Assume that in the iteration from $CAT(t)$ to $CAT(ta)$ at least one relevant suffix is inserted before the endpoint g is reached. Let $\gamma = \mathsf{path}(g)$.*

1. *The location of the endpoint is always represented by a suffix node.*
2. *If the endpoint is not a prefix node, it is the active prefix of $CAT(t)$ and γa will be the new active prefix of $CAT(ta)$.*

Proof. Let $\alpha(t) = ud\gamma$ for some $u \in \Sigma^*$, $d \in \Sigma$. If relevant suffixes are inserted, then $t = vud\gamma$ (1), $t = v'ud\gamma bw$ (2) and $t = v''c\gamma aw'$ (3a) or $t = \gamma aw''$ (3b) where $v, w \in \Sigma^+$, $v', v'', w', w'' \in \Sigma^*$, $a, b, c, d \in \Sigma$ and $a \neq b, c \neq d$. Hence by (2) and (3a) / (3b) γ is right branching (proving 1).

If γa appears somewhere in t, where it is not a prefix of t, by (3a), γ must be left branching and $\overline{\gamma}$ is a prefix node. Otherwise γa is a prefix of t and no further γa in t, so γ is the active prefix of t. γa is also a suffix of ta, so γa will be the new active prefix of ta (proving 2). $\square$

To find the prefix location of $\alpha(ta)$, we will distinguish the following cases:

1. No relevant suffixes were inserted, the active prefix got longer in the previous iteration.
2. No relevant suffixes were inserted, the active prefix did not change in the previous iteration.
3. Relevant suffixes were inserted, the endpoint is a prefix node.
4. Relevant suffixes were inserted, the endpoint is not a prefix node.

In case 1 the prefix location is already explicit in the prefix tree, it is a prefix leaf. In the previous iteration the active prefix got longer and was equal to the active suffix. By Lemma 3, a node that was previously a leaf got deleted (the active prefix is the longest nested prefix, so the next longer prefix must be a leaf). Since all leaves form a chain with inner distances one, the prefix location of the *asp* must be the next leaf (i.e., the *apl*). It can be easily found by the path starting at the base of the suffix crp for *asp*.

In case 2 the active suffix was made an explicit prefix location in the previous iteration. This location will be the starting point for the search in the prefix tree.

In case 3 the endpoint is a prefix node as well as a suffix node. It will be the starting point for the search in the prefix tree.

Case 4 can be dealt with similar to case 1, since the endpoint is the active prefix by Lemma 4. Therefore, the prefix location is already explicit in the prefix tree as a prefix leaf (it is the *apl*).

Note that it is important to have a prefix node as starting point in the tree because this enables us to use the paths in the prefix tree (otherwise it might be necessary to search a prefix node first which would add extra complexity).

4.2 Complexity of the Unidirectional Construction

Since all operations of a single iteration are either constant (steps 1, 2, 3, 6) or have a linear amortized complexity by Theorem 1 (step 4) and Theorem 2 (step 5), the following theorem follows.

Theorem 3. *$CAT(t)$ can be constructed in an on-line manner from left to right or from right to left in time $\mathcal{O}(|t|)$.*

5 Complexity of Bidirectional Construction

The affix tree construction algorithm is dual, so the tree can be constructed by any sequence of appending characters to any side of the string. We will prove the following:

Theorem 4. *Bidirectional construction of affix trees has linear time complexity.*

Because part of the complexity analysis is amortized, it is necessary to show that these parts do not interfere. The amortized complexity analysis depends upon step 4 and step 5. Their analyses are both based on the rps for the active suffix α_i and for the active prefix $\hat{\alpha}_i$ in iteration i. The amortized complexity analysis counts the nodes encountered using string lengths as an upper bound.

For the following lemmas and their proofs let $t^{(i)}$ be the string that is represented in the CAT after iteration i. Let (μ_i, ν_i) be the suffix crp for the suffix location of α_i, (u_i, v_i) be a (pseudo) prefix crp for the prefix location of α_i (pseudo because the prefix location of α_i might exist if it is a prefix node or when it will be inserted). Analogously, let $(\hat{u}_i, \hat{v}_i)$ $[(\hat{\mu}_i, \hat{\nu}_i)]$ be the suffix [prefix] crp for $\hat{\alpha}_i$, and let $\hat{\gamma}_i$ be the prefix endpoint in iteration i (which might be $\hat{\alpha}_{i-1}$). For $\hat{\gamma}_i$ let $(\hat{u}'_i, \hat{v}'_i)$ $[(\hat{\mu}'_i, \hat{\nu}'_i)]$ be the suffix [prefix] crp.

For step 4, additional work might occur when the active suffix grows in a prefix iteration s.t. the string distance from its location to the base of its crp grows (i.e., $|\nu_l| > |\nu_{l-1}|$). (Consider inserting $\mathtt{ba}^{k+1}\mathtt{ba}^{k+1}\mathtt{ac}$ from left to right versus inserting $\mathtt{a}^{k+1}\mathtt{ba}^{k+1}$ from left to right, appending b to the left and inserting ac on the right).

Lemma 5. *Additional k interim reverse (prefix) iterations add at most $\mathcal{O}(k)$ work to be accounted for in the amortized analysis of step 4.*

Proof. For step 4 we account the distance that is traversed by canonizing the rp for *asp*. For a given rp for *asp* we can determine the potential path (called *traversal path*) that would be taken in step 4 (e.g., if a previously unseen letter is appended). The nodes on this path make up the accounted complexity. The number of nodes is always less or equals to the string distance of the path (which we use in Theorem 1). We call a part of the path where the work is accounted by the traversed string distance *string-accounted*. If the work is accounted by the number of nodes, we call the part *node-accounted*. (Lemmas 6 and 7 introduce node-accounted parts in the analysis.) Furthermore, it is possible that the traversal path contains parts of the tree multiple times (the same nodes are traversed more than once, e.g. in the iteration from $\mathrm{CAT}(\mathtt{aab(ab)^i})$ to $\mathrm{CAT}(\mathtt{aab(ab)^ic})$). We will call such parts of the path *nested*. We take a look at two cases. The case where the base of the suffix crp for the *asp* changes and the case where suffix nodes are inserted in reverse iterations.

For the first case, suppose the active suffix grows in iteration l, when appending from the left: $t^{(l)} = at^{(l-1)}$, $\hat{\alpha}_l = a\hat{\gamma}_l$, $\alpha_l = a\alpha_{l-1}$ and $\hat{\alpha}_l = \alpha_l \Rightarrow \hat{\gamma}_l = \alpha_{l-1}$. It follows that the crps must be equal: $(\mu_l, \nu_l) = (\hat{u}_l, \hat{v}_l)$ and $(\mu_{l-1}, \nu_{l-1}) = (\hat{u}'_l, \hat{v}'_l)$. Therefore, the number of nodes that are accounted for step 5 is exactly the number of nodes added to the traversal path of step 4. Since the active suffix grows by Lemma 3, the new suffix location is a leaf and can be found in constant time (see section 4.1, cases 1 and 4). Hence, the additional work imposed by this intermediate reverse iteration is already accounted for. A node-accounted part is added to the front of the traversal path.

For case two, we will distinguish the different parts of the tree and of the traversal path. Obviously, inserting a node into a string-accounted part does not matter. A node inserted into a node-accounted part of the path that is not nested will add $\mathcal{O}(1)$ work. Only if a nested, node-accounted part of the traversal path existed, would a node add more than $\mathcal{O}(1)$ work. Node-accounted parts can be appended to the front of the traversal path by case 1. For a part of the tree to be added a second time, the base of the rp for *asp* needs to be moved back down below that part without the part being traversed in step 4. This can only happen when the base is moved down in the tree over reverse suffix links in reverse iterations. It takes at least k iterations to move the base down by string distance k. For each iteration we can save a coin and use the coins to turn a previously node-accounted part of the traversal path string-accounted. Thus, before adding a part for the second time to the traversal path the first instance in the traversal path can be turned string-accounted. As a result, there cannot be node-accounted, nested parts of the traversal path.

Hence, the overall cost for k reverse (prefix) iterations is $\mathcal{O}(k)$. □

The following lemma will give us a bound on the remaining work yet to be done in step 5 by means of the current data structure.

Lemma 6. *For every sequence of suffix iterations from $CAT(t^{(i)})$ to $CAT(t^{(j)})$, $i < j$, let $\dot{s}_i$ be the number of prefix nodes on the prefix path from u_i to the root in iteration i and let $\dot{s}_j$ be the number of prefix nodes on the prefix path from u_j to the root in iteration j. Let s_k be the number of prefix nodes traversed in step 5 of iteration k, then $\dot{s}_i + (j - i) \geq \dot{s}_j + \sum_{k=i+1}^{j} s_k$.*

Proof. By Lemma 2, every prefix node w on the path from u_j to the root has an atomic prefix link starting a chain of atomic prefix links leading to the root. All nodes on the path are prefix nodes. The chain has length $|\mathsf{path}(w)|$. Every suffix iteration moves u_j over a reverse prefix link to a new prefix branch (step 5). Obviously, u_i and u_j have a distance of $(j - i)$ prefix links. Thus, for at most $(j - i)$ nodes $|\mathsf{path}(w)| \leq (j - i)$ and the chain ends at the root before reaching the path from u_i to the root. Every other node with $|\mathsf{path}(w)| > (j - i)$ can be coupled with a node on the path from u_i to the root by following its prefix link chain. Hence $\dot{s}_i + (j - i) \geq \dot{s}_j$.

Let $j = i + 1$ and s_j nodes be traversed from u_i in step 5 until the atomic prefix link to u_j is found. Then $\dot{s}_i + 1 \geq \dot{s}_j + s_j = \dot{s}_{i+1} + s_{i+1}$. Continuing the argument, $\dot{s}_i + 2 \geq \dot{s}_{i+1} + 1 + s_{i+1} \geq \dot{s}_{i+2} + s_{i+2} + s_{i+1}$. Induction easily yields $\dot{s}_i + (j - i) \geq \dot{s}_j + \sum_{k=i+1}^{j} s_k$. □

With this lemma, the number of prefix nodes on the prefix path from u_i to the root in iteration i can be used to limit the work accounted for in all subsequent steps 5 and limit the complexity of the unidirectional construction in terms of nodes: $\dot{s}_0 + n \geq \dot{s}_n + \sum_{k=1}^{n} s_k \Longrightarrow \sum_{k=1}^{n} s_k \leq n - \dot{s}_n$.

Lemma 7. *The insertion of n_k prefix nodes in k interim reverse (prefix) iterations adds at most $\mathcal{O}(n_k + k)$ work for any following steps 5 of a suffix iteration.*

Proof. Suppose after k interim reverse iterations $|\mathsf{path}(u_{i+k})| > |\mathsf{path}(u_i)|$. Since the active suffix can only be enlarged along prefix edges k times, only k existing prefix nodes may be between $\overline{\alpha_{i+k}}$ (or its location) and u_i. All other prefix nodes must be nodes inserted in the prefix iterations. Hence a maximum of $n_k + k - 1$ prefix nodes can lie between u_{i+k} and u_i. Let $\dot{s}_i$ be the number of prefix nodes above u_i in iteration i and $\dot{s}_{i+k}$ be the number of prefix nodes above u_{i+k} after the k interim iterations. It follows that $\dot{s}_{i+k} - \dot{s}_i \leq n_k + k - 1$.

With Lemma 6, after iteration i the amortized work for i suffix iterations and additional m subsequent suffix iterations can be limited by $\sum_{j=1}^{i} s_j + \dot{s}_i + m$, where $\sum_{j=1}^{i} s_j$ is already performed and $\dot{s}_i + m$ is remaining. With k interim reverse iterations, the accounted work is then $\sum_{j=1}^{i} s_j + \dot{s}_{i+k} + m \leq \sum_{j=1}^{i} s_j + \dot{s}_i + m + n_k + k - 1$. The remaining work increased by $n_k + k - 1$.

□

Proof (Proof of Theorem 4).

Let the CAT T be constructed by l suffix iterations interleaved with k prefix iterations. Let $n = l + k$.

By Lemma 5, k additional reverse iterations add $\mathcal{O}(k)$ additional work for step 4. By Lemma 7, each preceding sequence of i prefix (suffix) iterations adds at most $\mathcal{O}(n_i+i)$ additional work for all following suffix (prefix) iterations, where n_i is the number of prefix (suffix) nodes inserted in the sequence.

The total cost is therefore $\mathcal{O}(n) + \mathcal{O}(k+l) + \mathcal{O}(n_k + k + n_l + l) = \mathcal{O}(n + k + l + n_k + k + n_l + l) = \mathcal{O}(n)$ □

6 Implementation of Paths

We will show that paths with the properties used above can be maintained on-line. Each path is represented by two dynamic arrays which can grow dynamically with constant amortized cost per operation (see section 18.4 "Dynamic tables" in [CLR90]). With one array growing downward and the other one upward, a path can grow in either direction (the underlying string can be represented similarly).

Only the first and the last node of a path need to have access to the path. They hold a pointer to the double-sided dynamic array object, an offset, and a length. Thus the first, the last, and the i-th element can be retrieved in constant time. Appending or prepending elements can also be done easily in constant time. When the path is split somewhere in the middle, both new paths will use the same array with different lengths and offsets, thus this operation can be done in constant time, too. Since all nodes in a path are connected by atomic edges, the paths can never grow on a side where they were split, hence no problems will arise through the split operation. Only the *asl* might have a non-atomic suffix link or might be deleted, thus disturbing this property. Fortunately, this is not the case due to the following lemma:

Lemma 8. *The prefix parent of the active suffix leaf (asl) is (also) a prefix node.*

Proof (Idea). The proof is a purely technical case distinction and we will therefore only give the basic proof idea. The proof will be done by contradiction. We will assume *asl*'s parent $\overline{\omega}$ to be a suffix only node and make a case distinction over the letters preceding ω in all occurrences of ω in t (which are at least three, one because ω is a suffix of $\alpha(t)$, two other because ω is right branching). In the case where ω is a prefix of t, the characters preceding the other two occurrences must be examined iteratively until the left of the string is reached, leading to the contradiction in the final case. □

The total number of nodes contained in all paths is $\mathcal{O}(n)$. If a path is split in the middle a hole is in that path representing a prefix and suffix node. The total number of holes is also bounded by $\mathcal{O}(n)$, so the space requirement is linear.

7 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 [GK97] and is implicitly stated in [BBH$^+$87] and other papers. Stoye developed a first algorithm that could construct affix trees in an on-line manner, but could not prove that his algorithm is linear [Sto95]. Through the introduction of paths, the algorithm could be improved and a linear behavior could be proven even for bidirectional construction.

Affix trees (especially augmented by paths) have the same properties as suffix trees and more. Construction is bidirectional and on-line. 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. At any position in the affix tree one can tell immediately whether the represented string is left branching or right branching or both.

Applications might be in pattern matching problems, where both the original and the reverse string are represented in one tree (e.g. finding all maximal repeats [Gus97]). Other new applications might come from the use of affix trees as two-head automaton with “edge compression” (see [GK97]).

8 Acknowledgements

I am in debt to Thomas Erlebach for thorough discussion and valuable criticism of this paper. I also want to thank Ernst W. Mayr for his help with the structuring of the final paper.

References

[BBH+87] A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *J. ACM*, 34(3):578–595, July 1987.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1st edition, 1990.

[GK97] 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.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[McC76] Edward M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *J. ACM*, 23(2):262–272, April 1976.

[Sto95] J. Stoye. Affixbäume. Master’s thesis, Universität Bielefeld, May 1995.

[Sto00] 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.

[Ukk95] E. Ukkonen. On-Line Construction of Suffix Trees. *Algorithmica*, 14:249–260, 1995.

[Wei73] P. Weiner. Linear pattern matching. In *IEEE 14th Annual Symposium on Switching and Automata Theory*, pages 1–11. IEEE, 1973.