Algorithmica (1997) 19: 331–353

Algorithmica

From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction[1]

R. Giegerich[2] and S. Kurtz[2]

Abstract. We review the linear-time suffix tree constructions by Weiner, McCreight, and Ukkonen. We use the terminology of the most recent algorithm, Ukkonen's on-line construction, to explain its historic predecessors. This reveals relationships much closer than one would expect, since the three algorithms are based on rather different intuitive ideas. Moreover, it completely explains the differences between these algorithms in terms of simplicity, efficiency, and implementation complexity.

Key Words. Text processing, On-line string matching, Suffix trees, Linear-time algorithm, Program transformation.

1. Motivation and Overview. Suffix trees provide most efficient solutions to a "myriad" [4] of string processing problems. The suffix tree for a string t really turns t inside out, immediately exposing properties like longest or most frequent subwords. The fundamental question whether w occurs in t can be answered in $\mathcal{O}(|w|)$ steps—independent of the length of t—once the suffix tree for t is constructed. Thus it is of great importance that the suffix tree for t can be constructed and represented in linear time and space.

In spite of their basic role for string processing, elementary books on algorithms and data structures barely mention suffix trees, and never give efficient algorithms for their construction [3], [21], [11], [1], [16], [7]. Recent exceptions are [22] and [13]. The reason for this is historical: starting with the seminal paper by Weiner [26], suffix tree construction has built up a reputation of being overly complicated. The purpose of this paper is to correct this reputation—by working out what is essential about efficient suffix tree construction, and what is unnecessary complexity.

More precisely, we review the linear-time algorithms of Weiner [26], McCreight [18], and Ukkonen [25]. We call these algorithms *wrf*,[3] *mcc*, and *ukk*.

We use the terminology of the most recent algorithm, Ukkonen's on-line construction, to explain its predecessors. This reveals relationships much closer than one would expect, since the three algorithms are based on rather different intuitive ideas. Moreover, it completely explains the differences between these algorithms in terms of simplicity, efficiency, and implementation complexity.

[1] Work by the first author was supported by a grant from the International Computer Science Institute, Berkeley, CA, USA.

[2] Technische Fakultät, Universität Bielefeld, Postfach 100 131, D-33501 Bielefeld, Germany. {robert, kurtz}@techfak.uni-bielefeld.de.

[3] *wrf* stands for the historic name "Weiner's repetition finder" used in [19].

Received February 12, 1995; revised January 28, 1996. Communicated by K. Mehlhorn.

In Section 2 we take some time to establish carefully the terminology necessary for suffix tree construction. New aspects of this section are a more generalized definition of suffix links, and observations concerning their duality with reverse prefix trees.

Section 3 gives an exposition of Ukkonen's and McCreight's algorithm on a very abstract level, showing that their different intuitive ideas lead to the same sequence of tree constructing operations. The two following sections make this observation more precise. A derivation of Ukkonen's algorithm is given (Section 4), and then Ukkonen's algorithm is transformed into McCreight's algorithm. Section 5 explains the transformation steps.

Section 6 revisits Weiner's algorithm. In a sense that is made precise there, *wrf* is shown as a version of *ukk* working on the "wrong" tree. Section 7 concludes.

As you see from this overview, the purpose of this paper is purely academic—no new algorithms, no improvements of old ones. Just a few explanations about how the known ones relate. If you have ever been puzzled by the complexity of linear-time suffix tree construction, we hope you will enjoy just reading through Sections 2, 3, and 6. The more technical material in Sections 4 and 5 may be safely spared out for a later reading.

2. Suffix Trees and Their Duality Properties

2.1. *$\mathcal{A}^+$-Trees and Suffix Trees.* Let $\mathcal{A}$ be a finite set, the *alphabet*. The elements of $\mathcal{A}$ are *characters*. ε denotes the *empty string*, $\mathcal{A}^*$ denotes the set of *strings over* $\mathcal{A}$, and $\mathcal{A}^+ = \mathcal{A}^* \backslash \{\varepsilon\}$. We use a, c, d, e to denote characters, and $b, p, q, s, t, u, v, w, x, y, z$ to denote strings. The *reverse* of $t = t_1 \cdots t_n$ is $t_n \cdots t_1$, also denoted by t^{-1}. If $t = uvw$ for some (possibly empty) u, v, w, then u is a *prefix* of t, v is a t-*word*, and w is a *suffix* of t. A prefix or suffix of t is *proper*, if it is different from t. A suffix or prefix of t is *nested*, if it occurs elsewhere in t. Notice that s is a nested suffix of t, if and only if s^{-1} is a nested prefix of t^{-1}. We call a t-word w *right-branching* (resp. *left-branching*) in t, if there are different characters a and c, such that wa and wc (resp. aw and cw) are t-words. Of course, w is right-branching in t, if and only if w^{-1} is left-branching in t^{-1}.

DEFINITION 2.1 ($\mathcal{A}^+$-Tree). An *$\mathcal{A}^+$-tree* T is a rooted tree with edge labels from $\mathcal{A}^+$. For each $a \in \mathcal{A}$, every node k in T has at most one a-edge $k \xrightarrow{aw} k'$.

Suffix trees are introduced below as a special form of $\mathcal{A}^+$-trees. However, most of the terminology used with suffix tree construction applies to $\mathcal{A}^+$-trees as well, so we present it first.

Let T be an $\mathcal{A}^+$-tree. By $path(k)$ we denote the concatenation of the edge labels on the path from the *root* of T to the node k. Due to the requirement of unique a-edges at each node of T, path labels are also unique and we can denote k by $\bar{w}$, if and only if $path(k) = w$. Moreover, by $T_{\bar{w}}$ we denote the *subtree of* T at node $\bar{w}$.

DEFINITION 2.2 (Words Represented in an $\mathcal{A}^+$-Tree). A string w *occurs* in T if and only if T contains a node $\overline{wu}$, for some u. By $words(T)$ we denote the set of strings occurring in T. For all $s \in words(T)$ we call $(\bar{b}, u)$ the *reference pair* of s with respect to T, if $\bar{b}$ is a node in T and $s = bu$. If b is the longest such prefix of s, then $(\bar{b}, u)$ is the *canonical reference pair* of s with respect to T. In such a case we write $\hat{s} = (\bar{b}, u)$.

A canonical reference pair of the form $(\bar{b}, \varepsilon)$ is called an *explicit* node, since it denotes the node $\bar{b}$ in T. A canonical reference pair $(\bar{b}, aw)$ is called an *implicit* node, since the node $\overline{baw}$ does not exist in T, but can be seen "inside" the edge $\bar{b} \xrightarrow{awv} \overline{bawv}$.

DEFINITION 2.3 (Atomic and Compact $\mathcal{A}^+$-Trees). T is *atomic*, if every edge in T is marked by a single character. T is *compact*, if every node in T is either the *root*, a leaf, or a branching node.

Atomic $\mathcal{A}^+$-trees are also known under the name "trie" [2]. Both atomic and compact $\mathcal{A}^+$-trees are uniquely determined by the words occurring in them. In an atomic $\mathcal{A}^+$-tree every node is explicit. In a compact $\mathcal{A}^+$-tree, nodes with a single outgoing edge are implicit nodes.

DEFINITION 2.4 (Suffix Trees).

1. A *suffix tree* for t is an $\mathcal{A}^+$-tree T, s.t. $words(T) = \{w \mid w \text{ is a } t\text{-word}\}$.
2. The *atomic suffix tree* for t is denoted by $ast(t)$.
3. The *compact suffix tree* for t is denoted by $cst(t)$.
4. $ast(t^{-1})$ and $cst(t^{-1})$ are called the atomic and compact *reverse prefix tree* for t, respectively.

Figure 1 shows different suffix trees for the string $aeceaceae$.

The reverse prefix tree is, of course, nothing new, but just the suffix tree for t^{-1}. It plays an important role in explaining suffix tree constructions. We refine our notation by writing $\overleftarrow{w}$ instead of $\overline{w^{-1}}$ for a node in a reverse prefix tree.

To decide whether a word w occurs in T takes $\mathcal{O}(|w|)$ steps: check if there is a path in T labeled w. This efficient access to all subwords of t is the *raison d'etre* of suffix trees.

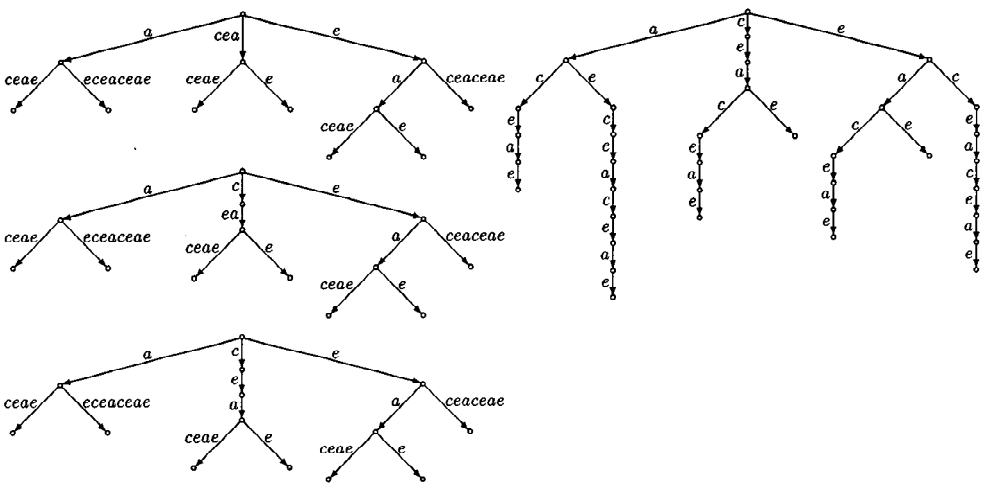

Fig. 1. Different suffix trees for the string $aeceaceae$.

The following is known about the space requirements for representing atomic and compact suffix trees (and holds for reverse prefix trees alike):

1. $ast(t)$ has $\mathcal{O}(n^2)$ nodes (take, e.g., $t = a^n c^n$ in Figure 4). However, isomorphic subtrees[4] can be shared [10]. Sharing brings the space requirements down to $\mathcal{O}(n)$ [8], [12]. However, subtree sharing may be impossible, when leaves are to be annotated with extra information.
2. $cst(t)$ has $\mathcal{O}(n)$ nodes, as all inner nodes are branching, and there are at most n leaves. The edge labels can be represented in constant space by a pair of indices into t. This is necessary to achieve a theoretical worst-case bound of $\mathcal{O}(n)$. In practice, this is quite a delicate choice of representation in a virtual memory environment. Traversing the tree and reading the edge labels will create random-like accesses into t, and can lead to performance problems with the memory subsystem.

2.2. *Open Edges*. A particularly convenient representation of edges which lead to a leaf node (leaf edges, for short) was introduced in [25]. The label of a leaf edge always extends to the end of the actually scanned prefix of t. We may as well represent an index pair $(i, |t|)$ by (i, ∞), with ∞ denoting $|t|$, whatever its value is. This means that if t is extended to the right, the label of the leaf edge grows implicitly, and the leaf continues to represent a complete suffix of (the extended) t. This representation is called "open edge." It plays a crucial part in the following sections. With a little speculation, we might even say: if Weiner had seen this idea in 1973, he would have designed Ukkonen's algorithm then (and it would be in all textbooks today). We return to this in Section 6.

2.3. *Active Suffixes and Prefixes*. The following notion plays a central part in all constructions:

DEFINITION 2.5 (Active Suffix and Prefix). The *active suffix* of t is its longest nested suffix, denoted $\alpha(t)$. The *active prefix* of t is its longest nested prefix, denoted $\alpha^{-1}(t)$.

With this notation, we have $\alpha(t^{-1}) = (\alpha^{-1}(t))^{-1}$.

The node $(\bar{u}, v)$ representing the active suffix of t in $cst(t)$ is the neuralgic point of the suffix tree. If t is to be extended to the right by another character, changes in the tree structure (if any) will start at this point. Correspondingly, the active prefix node will respond to extensions of t on the left. This behavior is proved and spelled out in detail in later sections.

McCreight uses functions *head* and *tail* that split a suffix s of t into an initial part that already occurs to the left, and the remainder. We can define them in the following way.

DEFINITION 2.6 (*head* and *tail*). Let $t = us$ for some strings u and s. $head(s)$ is the longest prefix x of s, such that x is a nested suffix of ux. $tail(s)$ is defined by $s = head(s)tail(s)$.

[4] Two $\mathcal{A}^+$-trees T and T' are *isomorphic*, if there is a bijection φ between the node sets of T and T', s.t. $\bar{w} \xrightarrow{u} \overline{wu}$ is an edge in T, if and only if $\varphi(\bar{w}) \xrightarrow{u} \varphi(\overline{wu})$ is an edge in T'.

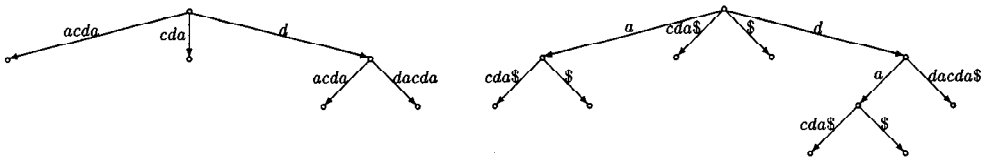

Fig. 2. The compact suffix trees for $ddacda$ and $ddacda\$$.

2.4. *The Role of the Sentinel Character.* If s is a nested suffix of t, then a suffix tree for t does not contain a leaf $\bar{s}$. It is often convenient to add to t a *sentinel character*, say \$, that does not occur in t. Then $t\$$ has no nested suffix, except for the empty string, i.e., each nonempty suffix of $t\$$ uniquely corresponds to a leaf in a suffix tree T for $t\$$. Considering a t-word w and the node $\bar{w}$ in T, the number of leaves of $T_{\bar{w}}$ is equal to the number of positions in t where w occurs:

DEFINITION 2.7 (Suffix-Rests). For a node $\bar{w}$ in a suffix tree T for t, let $\mathit{suffixRests}_T(\bar{w}) = \{s \mid ws \text{ is a suffix of } t\}$.

Clearly, $\mathit{suffixRests}_T(\bar{w})$ uniquely determines the shape of $T_{\bar{w}}$. However, can $\mathit{suffixRests}_T(\bar{w})$ be determined from the edge labels of $T_{\bar{w}}$? The answer is Yes, if there is the sentinel, since then there is the leaf $\overline{ws}$ for every $s \in \mathit{suffixRests}_T(\bar{w})$. The answer is No without the sentinel, as can be seen for $T_{\bar{d}}$ in $T = cst(ddacda)$ as shown in Figure 2.

It often simplifies proofs and constructions considerably to assume the presence of the sentinel character. Only in contexts where t may be expanded to the right (e.g., during on-line construction), does the requirement for a unique final character not make sense.

In the subsequent sections, the sentinel character is not assumed unless we explicitly say so.

2.5. *Suffix Links.* For construction and many applications of $\mathcal{A}^+$-trees it is convenient to augment $\mathcal{A}^+$-trees with auxiliary edges that connect nodes quite unrelated in the tree structure:

DEFINITION 2.8 (Suffix Links). Consider an $\mathcal{A}^+$-tree T. Let $\overline{aw}$ be a node in T, and let v be the longest suffix of w, such that $\bar{v}$ is also a node in T. An unlabeled edge $\overline{aw} \longrightarrow \bar{v}$ is a *suffix link* in T. A suffix link $\overline{aw} \longrightarrow \bar{w}$ is called *atomic*.

Notice that node $\bar{v}$ is well defined, since $\bar{\varepsilon}$ is a node and ε is a suffix of w.

When the $\mathcal{A}^+$-tree is a trie, suffix links are identical to the failure transitions of [2]. The name suffix link is due to McCreight [18]. Some authors also define a link for the *root*: $\bar{\varepsilon} \longrightarrow \bar{\varepsilon}$. We found that this obscures the algorithms as well as the observations in Section 2.6.

PROPOSITION 2.9.

1. *In the atomic suffix tree for t, all suffix links are atomic.*
2. *In the compact suffix tree for $t\$$, all suffix links are atomic.*

PROOF. 1. This follows from the definitions, since all nodes in $ast(t)$ are explicit.

2. We must show that for each node $\overline{aw}$, $\bar{w}$ is also a node in $cst(t\$)$. $\overline{aw}$ is either a branching node, or a leaf. Hence aw is right-branching or a nonnested suffix of $t\$$. However, then the same holds for w, and so $\bar{w}$ is a node in $cst(t)$. □

What if we drop the sentinel in the case of assertion 2? The suffix links for all inner nodes in $cst(t)$ are atomic. For a leaf $\overline{aw}$, w may be nested (due to the lack of \$) and not right-branching, so there is no (explicit) node $\bar{w}$. In this case, we have a nonatomic suffix link $\overline{aw} \longrightarrow \bar{v}$ for some proper suffix v of w. Note that this link is the only possible exception, with all other suffix links in $cst(t)$ being atomic.

Suffix links are the key to efficient sequential suffix tree construction, but there is more to them than this.

The atomic suffix tree of t, augmented by suffix links, can be seen as a two-head automaton. Denoting the two heads by [and], we can represent a configuration as $u[v]w$, where

- uv is the scanned part,
- v is the memorized part, and
- w is the unread part of the input string.

Now if $v = ay$ and $w = cx$, there are two possible transitions:

$$\begin{array}{ll} u[ay]cx \rightsquigarrow u[ayc]x & \text{by following the edge } \overline{ay} \xrightarrow{c} \overline{ayc}, \\ u[ay]cx \rightsquigarrow ua[y]cx & \text{by following the suffix link } \overline{ay} \longrightarrow \bar{y}. \end{array}$$

This view is taken from [19]. It nicely summarizes the additional power of suffix links that makes them useful in many contexts. For example, such an automaton can be used to compute the matching statistics in [9], the q-gram distance [24], or the shift-table for the Boyer–Moore algorithm [17].

2.6. *Dualities Between Suffix Trees and Suffix Links.* We now study the deeper relation between suffix trees and their suffix links. First we note that the suffix links form a tree themselves.

DEFINITION 2.10. The suffix link tree T^{-1} of an $\mathcal{A}^+$-tree T has a node $\overleftarrow{w}$ for each node $\bar{w}$ of T, and an edge $\overleftarrow{w} \xrightarrow{v^{-1}} \overleftarrow{vw}$ when $\overline{vw} \longrightarrow \bar{w}$ is a suffix link in T.

It is easy to confirm that T^{-1} is in fact a tree, since each node in T has exactly one suffix link, which designates its parent in T^{-1}. The notation T^{-1} will be justified by our subsequent results.

For an arbitrary $\mathcal{A}^+$-tree T, T^{-1} is generally not an $\mathcal{A}^+$-tree, as can be seen in Figure 3: node $\overleftarrow{e}$ has two d-edges. However, this changes when T is a suffix tree:

Fig. 3. An $\mathcal{A}^+$-tree and its suffix link tree.

PROPOSITION 2.11 (Duality for Atomic Suffix Trees). $(ast(t))^{-1} = ast(t^{-1})$. *In words: the suffix link tree of an atomic suffix tree is the reverse prefix tree.*

PROOF. There is an edge $\overleftarrow{w} \xrightarrow{a} \overleftarrow{aw}$ in $(ast(t))^{-1}$, iff there is a suffix link $\overline{aw} \longrightarrow \bar{w}$ in $ast(t)$, iff there are nodes $\bar{w}$ and $\overline{aw}$ in $ast(t)$, iff there are nodes $\overleftarrow{w}$ and $\overleftarrow{aw}$ in $ast(t^{-1})$, iff there is an edge $\overleftarrow{w} \xrightarrow{a} \overleftarrow{aw}$ in $ast(t^{-1})$. □

Figure 4 shows $ast(aaaccc)$ and $ast(aaaccc^{-1})$. Solid edges represent $ast(aaaccc)$, while dotted edges (without their labels) represent the suffix links. Vice versa for $ast(aaaccc^{-1})$.

The reason why this duality is not widely known is that when considering the compact suffix tree (our main object of interest), it is obscured by the fact that the explicit nodes of a compact suffix tree and the corresponding reverse prefix tree do not coincide. But a weaker form of duality still holds:

PROPOSITION 2.12 (Weak Duality for Compact Suffix Trees).

1. $(cst(t))^{-1}$ *is an* $\mathcal{A}^+$*-tree.*
2. $(cst(t))^{-1}$ *represents a subset of the words represented by* $cst(t^{-1})$.
3. $((cst(t))^{-1})^{-1} = cst(t)$.

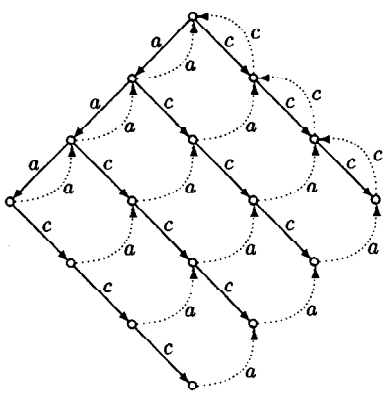

Fig. 4. The atomic suffix tree for $aaaccc$ and $aaaccc^{-1}$.

PROOF. 1. Assume there is a node $\bar{w}$ in the suffix link tree that has two a-edges. This means that in $cst(t)$ we have suffix links $\overline{uaw} \longrightarrow \bar{w}$ and $\overline{vaw} \longrightarrow \bar{w}$ with $u \neq \varepsilon$ and $v \neq \varepsilon$. $\overline{aw}$ is not a node, since otherwise, the links would point to $\overline{aw}$.

- Suppose $\overline{uaw}$ or $\overline{vaw}$ is an inner node. Then uaw or vaw is right-branching in t, and so must be aw. So $\overline{aw}$ must be a node, which is a contradiction.
- Suppose $\overline{uaw}$ and $\overline{vaw}$ are leaves. Without restriction to generality vaw is a suffix of uaw, and it is longer than w. Hence there can be no suffix link $\overline{uaw} \longrightarrow \bar{w}$.

2. The suffix link chain from $\bar{w}$ to $\bar{\varepsilon}$ in $cst(t)$ yields a path labeled w^{-1} in the suffix link tree. Of course, w^{-1} is a t^{-1}-word.

3. Because of Statement 1, $(cst(t))^{-1}$ is an $\mathcal{A}^+$-tree, so $((cst(t))^{-1})^{-1}$ is defined. The node set is unchanged under the $()^{-1}$ operation, except for reversal of node names. There is a suffix link $\overleftarrow{vw} \longrightarrow \overleftarrow{v}$ in $(cst(t))^{-1}$, iff there is no suffix r^{-1} of $(vw)^{-1}$, s.t. $\overleftarrow{r}$ is a node and $|(vw)^{-1}| > |r^{-1}| > |v^{-1}|$, iff there is no prefix r of vw, s.t. $\bar{r}$ is a node and $|vw| > |r| > |v|$, iff $cst(t)$ has an edge $\bar{w} \xrightarrow{v} \overline{wv}$. □

Statement 1 of Proposition 2.12 can be slightly generalized: if T is (any sort of) suffix tree of some string t, then T^{-1} is an $\mathcal{A}^+$-tree. The reverse of this statement does not hold. For example, let T be an $\mathcal{A}^+$-tree representing the words aa and bb. Then $T^{-1} = T$. Thus T^{-1} is an $\mathcal{A}^+$-tree, but T is not a suffix tree.

A t^{-1}-word w^{-1} is not represented in the suffix link tree, if w is neither right-branching in t nor a suffix of t. (Adding the sentinel does not change this situation.) This is also why $(cst(t))^{-1}$ is *not a subtree* of $cst(t^{-1})$: some nodes of $(cst(t))^{-1}$ are not nodes in $cst(t^{-1})$. However in the precise sense of Proposition 2.12, the suffix link tree approximates the reverse prefix tree. By duality, $cst(t)$ itself approximates the prefix links of $cst(t^{-1})$.

At this point, it seems natural to ask whether suffix/prefix trees can be subsumed by a more general data structure in $\mathcal{O}(n)$ space, which has the duality as an inherent property. In fact, the affix trees recently introduced by Stoye [23] are such a self-dual data structure. However, this is beyond the scope of the present paper.

We now turn to suffix tree constructions.

3. An Abstract Comparison of *ukk* and *mcc*.

ukk reads t from left to right, character by character, and incrementally constructs suffix trees for the prefixes of t seen so far. With *ukk*, labels of open edges grow implicitly as t is read, while some edges are split and new open edges are inserted explicitly. The intermediate trees when constructing $cst(adadc)$ using *ukk* are shown in the left column of Figure 5.

mcc inserts the suffixes of t into an initially empty tree. Starting with the longest suffix, the method is not on-line, and the intermediate trees are not suffix trees. For a suffix s of t let $T(s)$ denote the $\mathcal{A}^+$-tree representing the suffixes of t that are longer than or equal to s. The right column of Figure 5 shows the intermediate trees when constructing $T(c) = cst(adadc)$ using *mcc*.

We introduce two abstract tree construction operations:

- $split(\bar{u}, v)$ replaces an edge $\bar{u} \xrightarrow{vw} \overline{uvw}$ by two edges $\bar{u} \xrightarrow{v} \overline{uv} \xrightarrow{w} \overline{uvw}$.
- $add(\bar{u}, a\cdots)$ adds a new edge from node $\bar{u}$ to a leaf, labeled $a\cdots$.

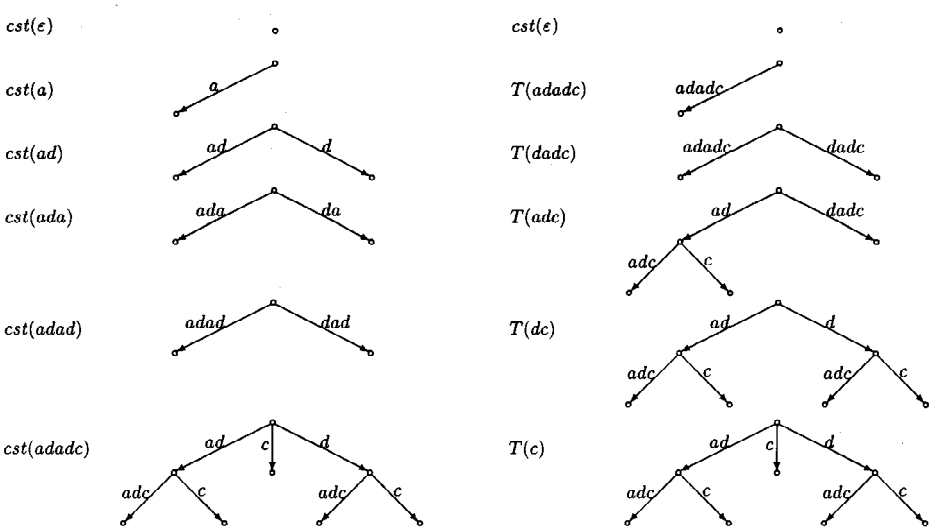

Fig. 5. Sequence of trees constructed by *ukk* and *mcc*.

Note that the *add*-operation abstracts from whether the edge label is entered fully or left open to grow later. The central observation of this section is the following: the intermediate trees of *ukk* and *mcc* are both constructed by the same sequence of abstract operations! However, these operations are applied to the intermediate trees in a different way. Both are shown in Table 1.

Analogies on an abstract level often break down when a more concrete level of presentation is used. In our case we have abstracted from a number of aspects which are essential in making both *ukk* and *mcc* linear-time algorithms. However, when we take these into account, our analogy still persists. We summarize what is shown in full in Sections 4 and 5:

1. *ukk* can be transformed into *mcc* by a modification of its control structure, leaving the sequence of tree constructing operations invariant.
2. This modification is a slight optimization. Under a fair implementation of the related data structures, it will give *mcc* a minor efficiency advantage over *ukk*, on every possible input.

Table 1. Operations to compute intermediate trees.

Operation	Applied by *ukk* to	Applied by *mcc* to
$add(root, a\cdots)$	$cst(\varepsilon)$	$cst(\varepsilon)$
$add(root, d\cdots)$	$cst(a)$	$T(adadc)$
$split(root, ad)$	$cst(adad)$	$T(dadc)$
$add(\overline{ad}, c\cdots)$	$cst(adad)$	$T(dadc)$
$split(\underline{root}, d)$	$cst(adad)$	$T(adc)$
$add(\underline{d}, c\cdots)$	$cst(adad)$	$T(adc)$
$add(root, c\cdots)$	$cst(adad)$	$T(dc)$

3. This transformation sacrifices the on-line property. *mcc* will always read ahead of *ukk* in t. This lookahead is quantified in Proposition 5.5.

Assertion 2 is confirmed by the measurements in [15]. In fact, this invariance of the relative efficiency of *ukk* and *mcc* made us first wonder about a deeper relationship between these two algorithms. We were incited further by a note in [25], where Ukkonen remarks that, on the technical level, the main difference between *ukk* and *mcc* lies in the way in which character reads and suffix link traversals are arranged over the loops of the program. Our study confirms, concretizes, and explains this observation.

4. Development of *ukk* and *mcc*

4.1. *A Short Derivation of ukk.* Space does not allow a complete derivation of *ukk* here. We only give a short explanation together with the concrete algorithm, and refer the reader to the development in [25] or [15].

On-line construction means generating a series of suffix trees for longer and longer prefixes of t. While $cst(\varepsilon)$ is trivial (just the *root* with no edges), we study the step from $cst(p)$ to $cst(pa)$, where p is a prefix of t and a is the next character in t to be read. To construct $cst(pa)$ we have to insert some suffixes of pa into $cst(p)$. Let sa be a suffix of pa. *ukk* is based on the following observations about suffixes:

- If $|sa| > |\alpha(p)a|$, then s is not a nested suffix of p and thus s corresponds to a leaf in $cst(p)$. In such a case sa will correspond to the same leaf in $cst(pa)$ by the implicit growing of the corresponding open edge.
- If $|\alpha(p)a| \geq |sa| > |\alpha(pa)|$, then sa is a relevant suffix of pa, and a new leaf $s\bar{a}$ must be introduced.
- If $|\alpha(pa)| \geq |sa|$, then no action is required, since sa already occurs in $cst(p)$.

In *ukk* a suffix s is represented by its canonical reference pair. To make reference pairs canonical we use a function *canonize*. When the relevant suffixes of pa are processed in their natural order, i.e., by decreasing length from $\alpha(p)a$ to (excluding) $\alpha(pa)$, the corresponding canonical reference pairs can be accessed via the suffix links.

With the prefix p of t globally given, *ukk* takes four arguments with each call:

- $T = cst(p)$.
- The set L of suffix links in T.
- The canonical reference pair $(\bar{b}, u)$ of $\alpha(p)$.
- The position i, such that $p = t_1 \cdots t_{i-1}$ and t_i is the next input character to be read.

For convenience we denote $\bar{b}'$ by $L(\bar{b})$, whenever $\bar{b} \longrightarrow \bar{b}' \in L$.

The access from one canonical reference pair to the next is accomplished by a function *link*, which is defined as follows:

$$link(T, L, (\bar{b}, \varepsilon)) = \begin{cases} (\bar{b}, \varepsilon), & \text{if } \bar{b} = root, \\ (L(\bar{b}), \varepsilon, & \text{otherwise.} \end{cases}$$

$$link(T, L, (\bar{b}, cw)) = \begin{cases} canonize(T, (\bar{b}, \underline{w})), & \text{if } \bar{b} = root, \\ canonize(T, (L(b), cw)), & \text{otherwise.} \end{cases}$$

Let $n = |t|$. *ukk* is simply an iteration of a function *update* that inserts the relevant suffixes.

$$ukk(T, L, (\bar{b}, u), i) = \begin{cases} T, & \text{if } i = n+1, \\ ukk(T', L', (\bar{b'}, u'), i+1), & \text{otherwise,} \end{cases}$$

where $(T', L', (\bar{b'}, u')) = update(T, L, (\bar{b}, u), i)$.

To construct $cst(t)$, the initial call of *ukk* is $ukk(\emptyset, \emptyset, (root, \varepsilon), 1)$. Now we define the function *update*. For each relevant suffix, *update* creates (if necessary) a new branching node by edge splitting, and sets its suffix link. It adds a new open edge for the new suffix, and advances $(\bar{b}, u)$ via the suffix link to the next suffix, until the canonical reference pair of $\alpha(pt_i)$ is reached. The function *canonize* is applied whenever the right component of a reference pair is extended by a new character. As indicated in Section 2, edge labels are now implemented as index pairs. The pair (l, r) denotes the label $t_l \cdots t_r$, while (i, ∞) denotes the suffix $t_i \cdots$.

$$update(T, L, (\bar{b}, \varepsilon), i) = \begin{cases} (T, L, canonize(T, (\bar{b}, t_i))), & \text{if } \bar{b} \text{ has a } t_i\text{-edge,} \\ (T \sqcup ((\bar{b}, \varepsilon), i), L, (\bar{b}, \varepsilon)), & \text{else if } \bar{b} = root, \\ update(T \sqcup ((\bar{b}, \varepsilon), i), & \\ \quad L, (L(\bar{b}), \varepsilon), i), & \text{otherwise,} \end{cases}$$

$$update(T, L, (\bar{b}, cw), i) = \begin{cases} (T, L, canonize(T, (\bar{b}, cwt_i))), & \text{if } t_{l+|cw|} = t_i, \\ update(T \sqcup ((\bar{b}, cw), i), & \\ \quad L', (\bar{b'}, u'), i), & \text{otherwise,} \end{cases}$$

where $\bar{b} \xrightarrow{(l,r)} \bar{v}$ is a c-edge, $(\bar{b'}, u') = link(T, L, (\bar{b}, cw))$, and $L' = L \cup \{(\overline{bcw}, \overline{b'u'})\}$.

The expression $T \sqcup ((\bar{b}, u), i)$ denotes the $\mathcal{A}^+$-tree that results from inserting the suffix $but_i \cdots$ into T. It is formally defined as follows:

$$\begin{aligned} T \sqcup ((\bar{b}, \varepsilon), i) &= T \cup \{\bar{b} \xrightarrow{(i,\infty)} \overline{bt_i}\}, \\ T \sqcup ((\bar{b}, cw), i) &= (T \setminus \{\bar{b} \xrightarrow{(l,r)} \bar{v}\}) \cup \{\bar{b} \xrightarrow{(l,k)} \overline{bcw} \xrightarrow{(k+1,r)} \bar{v}, \overline{bcw} \xrightarrow{(i,\infty)} \overline{bt_i}\}, \end{aligned}$$

where $\bar{b} \xrightarrow{(l,r)} \bar{v}$ is a c-edge and $k = l + |w|$.

The first equation for $\sqcup$ implements the abstract add-operation of Section 3. The second equation corresponds to a *split*/*add*-combination.

4.2. *A Short Description of mcc*. Before we embark on the derivation of *mcc* from *ukk*, we give a short intuitive description of our target. The complete algorithm is given in Section 5, at the end of our transformation series.

mcc constructs $cst(t)$ by successively inserting the suffixes of t into an initially empty tree, from longest to shortest. It produces a sequence

$$cst(\varepsilon), T(t_1 \cdots t_n), T(t_2 \cdots t_n), \ldots, T(t_{n-1} t_n), T(t_n) = cst(t)$$

of compact $\mathcal{A}^+$-trees, of which only the first and the last one is a suffix tree. The initial step of *mcc* is trivial: $T(t) = T(t_1 \cdots t_n)$ is obtained from $cst(\varepsilon)$ by inserting the longest suffix t. Thus, $T(t)$ is the compact $\mathcal{A}^+$-tree with only one edge $root \xrightarrow{t} \bar{t}$. Let as be a suffix

of t, and suppose $x = head(as)$. For the step from $T(as)$ to $T(s)$, *mcc* first determines $tail(s)$ and the canonical reference pair $\hat{y}$ of $y = head(s)$ in constant time from $\hat{x}$ and $tail(as)$. This is accomplished by following suffix links and scanning downward in the actual tree using a function *scan* (see Section 5.1). Then it constructs $T(s)$ from $T(as)$ by splitting for the node $\bar{y}$ (if necessary) and adding a leaf edge labeled $tail(s)$.

5. Transforming *ukk* into *mcc*

5.1. *A Series of Program Transformations from ukk to mcc.* *mcc* assumes that t ends with a sentinel. We assume the same in the rest of this section.

In Figure 5 we saw that *ukk* produces a sequence

$$cst(\varepsilon), cst(t_1), cst(t_1t_2), \ldots, cst(t),$$

which might contain a subsequence of suffix trees, in which only the leafs grow implicitly with the length of the input string. In the sequence of trees produced by *mcc* there are no such "nonessential" subsequences, i.e., every step produces a tree of a different structure. In the following we show that it is in fact the additional "nonessential" steps in *ukk* that make the difference between both algorithms. Technically, we transform *ukk* stepwise into equivalent functions ukk_1, ukk_2, and ukk_3, such that ukk_3 does only "essential" derivation steps. Equivalence means that for $k = 1, 2, 3$ we have $ukk(\emptyset, \emptyset, (root, \varepsilon), 1) = ukk_k(\emptyset, \emptyset, (root, \varepsilon), 1)$, and that linear-time complexity is preserved. From ukk_3 we synthesize a definition of *mcc*.

DEFINITION 5.1 (Essential Steps). A derivation step $ukk_k(T, L, (\bar{b}, w), i) \Rightarrow ukk_k(T', L', (\bar{h}, q), j)$, $k = 1, 2, 3$, is *essential*, if the set of edges in T' is different from the set of edges in T.

The first transformation step does not affect the essential steps. It simply eliminates the function *update* in *ukk*, yielding an equivalent function ukk_1:

$$ukk_1(T, L, (\bar{b}, \varepsilon), i) = \begin{cases} T, & \text{if } i = n+1, \quad (1)\\ ukk_1(T, L, canonize(T, (\bar{b}, t_i)), i+1), & \text{else if } \bar{b} \text{ has a } t_i\text{-edge}, \quad (2)\\ ukk_1(T \sqcup ((\bar{b}, \varepsilon), i), L, (\bar{b}, \varepsilon), i+1), & \text{else if } \bar{b} = root, \quad (3)\\ ukk_1(T \sqcup ((\bar{b}, \varepsilon), i), L, (L(\bar{b}), \varepsilon), i), & \text{otherwise}, \quad (4) \end{cases}$$

$$ukk_1(T, L, (\bar{b}, cw), i) = \begin{cases} T, & \text{if } i = n+1, \quad (5)\\ ukk_1(T, L, canonize(T, (\bar{b}, cwt_i)), i+1), & \text{else if } t_{l+|cw|} = t_i, \quad (6)\\ ukk_1(T \sqcup ((\bar{b}, cw), i), L', (\bar{b}', u'), i), & \text{otherwise}, \quad (7) \end{cases}$$

where $\bar{b} \xrightarrow{(l,r)} \bar{v} \in T$ is a c-edge, $(\bar{b}', u') = link(T, L, (\bar{b}, cw))$, and $L' = L \cup \{(\overline{bcw}, \overline{b'u'})\}$.

To develop ukk_2 we need the following lemmas.

LEMMA 5.2. *Let csa be a relevant suffix of pa, such that s is not a right-branching p-word. Then sa is a relevant suffix of pa.*

PROOF. By assumption, cs is a nested suffix of p. This implies that s is a nested suffix of p, i.e., $p = vcsdw$, for some strings v, w and some character d. Since csa is not a p-word, we have $d \neq a$. Suppose $p = v'sd'w'$ for some character d' and some strings v' and w'. Then $d = d'$, since otherwise s would be right-branching in p. Hence $d' \neq a$, i.e., sa is not a p-word. Thus sa is a relevant suffix of pa. □

LEMMA 5.3. *Consider a derivation*

$$(*) \qquad e_0 = ukk_1(\emptyset, \emptyset, (root, \varepsilon), 1) \Rightarrow e_1 \cdots \Rightarrow e_N = cst(t).$$

1. *$e_N = cst(t)$ is derived from e_{N-1} by an application of* (1).
2. *Assume that e_{k+1} is derived from e_k by an application of* (6). *Then we have $0 < k < N - 1$ and e_k is derived from e_{k-1} by an application of* (2) *or* (6).

PROOF. 1. Let $e_{N-1} = ukk_1(T, L, (\bar{h}, q), i + 1)$, such that $i = n$. Let $p = t_1 \cdots t_{i-1}$ and assume that $q \neq \varepsilon$. Then e_{N-1} is derived from $e_{N-2} = (T, L, (\bar{b}, w), i)$ by an application of (2) or (6). Hence bwt_i occurs in T, i.e., bwt_i is a p-word and thus the character t_i occurs in p. This is a contradiction, since t_i is the sentinel in t. Hence $q = \varepsilon$, i.e., e_N is derived from e_{N-1} by an application of (1).

2. We have $k > 0$, since (6) cannot be applied to e_0. $k < n - 1$ follows from Statement 1. e_k could not be derived from e_{k-1} by an application of (1), (3), (4), or (5), since this would lead to an expression, to which (6) is not applicable. We show that this is also true for (7). Assume that e_k is derived from e_{k-1} by an application of (7). Hence $e_{k-1} = ukk_1(T, L, (\bar{b}, cw), i)$ and $e_k = ukk_1(T \sqcup ((\bar{b}, cw), i), L', (\bar{b'}, u'), i)$, where $(\bar{b'}, u') = link(T, L, (\bar{b}, cw))$ and $L' = L \cup \{(\overline{bcw}, \overline{b'u'})\}$. By assumption, $u' \neq \varepsilon$. Let $p = t_1 \cdots t_{i-1}$ and $a = t_i$. Now observe that $bcwa$ is a relevant suffix of pa and that $b'u'$ is not right-branching in p. By Lemma 5.2, $b'u'a$ is a relevant suffix of pa, i.e., $b'u'a$ is not a p-word. Hence $b'u'a$ does not occur in $cst(p)$ and therefore not in $T \sqcup ((\bar{b}, cw), i)$. Thus $t_{l+|cs|} \neq t_i$ and (6) is not applicable to e_k, which is a contradiction. Hence e_k is derived from e_{k-1} by (2) or (6). □

Consider a maximal subderivation $e_k \Rightarrow \cdots \Rightarrow e_{k+m+1}$ of derivation $(*)$, in which only (2) or (6) is applied. By Statement 2 of Lemma 5.3 we can conclude that e_{k+1} is derived from e_k by an application of (2). If $e_k = (T, L, (\bar{b}, \varepsilon), i)$, then $e_{k+m+1} = (T, L, (\bar{h}, q), j)$ and $((\bar{h}, q), j)$ is the information we need to insert the suffix $bt_it_{i+1} \cdots$ into T. We have $hqt_jt_{j+1} \cdots = bt_it_{i+1} \cdots$, such that $(\bar{h}, q)$ is the canonical reference pair of the longest prefix of $bt_it_{i+1} \cdots$ that occurs in T. Thus to compute e_{k+m+1} from e_k we can start at node $\bar{b}$, scan a prefix $t_i \cdots t_{j-1}$ of $t_it_{i+1} \cdots$ until we "fall out of the tree"[5] and canonize the reference pair $(\bar{b}, t_i \cdots t_{j-1})$ to obtain $(\bar{h}, q)$. Instead of computing $((\bar{h}, q), j)$ by some nonessential steps using (2) or (6) we use a function *scan*:

$$scan(T, \bar{b}, i) = \begin{cases} ((\bar{b}, \varepsilon), i), & \text{if } \bar{b} \text{ has no } t_i\text{-edge}, \\ ((\bar{b}, p), i + |p|), & \text{else if } |p| < r - l + 1, \\ scan(T, \bar{v}, i + |p|), & \text{otherwise}, \end{cases}$$

where $\bar{b} \xrightarrow{(l,r)} \bar{v}$ is a t_i-edge, and p is the longest common prefix of $t_l \cdots t_r$ and $t_it_{i+1} \cdots$.

[5] The sentinel ensures that this must happen before $t_it_{i+1} \cdots$ is exhausted, since it cannot be a nested suffix.

If we use *scan* to compute e_{k+m+1} from e_k we do not need (2) and (6). Furthermore, from Statement 1 of Lemma 5.3 we learn that (5) is not necessary. Hence we can transform ukk_1 into the following equivalent function ukk_2:

$$ukk_2(T, L, (\bar{b}, \varepsilon), i) = \begin{cases} T, & \text{if } i = n+1, \quad (8)\\ ukk_2(T, L, (\bar{h}, q), j), & \text{else if } j > i, \quad (9)\\ ukk_2(T \sqcup ((\bar{h}, q), j), L, (\bar{h}, q), j+1), & \text{else if } (\bar{h}, q) = (root, \varepsilon), \quad (10)\\ ukk_2(T \sqcup ((\bar{h}, q), j), L, (L(\bar{h}), \varepsilon), j), & \text{otherwise}, \quad (11) \end{cases}$$

where $((\bar{h}, q), j) = scan(T, \bar{b}, i)$

$$ukk_2(T, L, (\bar{b}, cw), i) = ukk_2(T \sqcup ((\bar{b}, cw), i), L', (\bar{b'}, u'), i), \quad (12)$$

where $(\bar{b'}, u') = link(T, L, (\bar{b}, cw))$ and $L' = L \cup \{(\overline{bcw}, \overline{b'u'})\}$.

Notice that (10) and (11) result from substituting $(\bar{b}, \varepsilon)$ by $(\bar{h}, q)$ and i by j in (3) and (4). This is correct, since $(\bar{h}, q) = (\bar{b}, \varepsilon)$, whenever $i = j$. Obviously, the program transformation from ukk_1 to ukk_2 does not affect the linear-time complexity, since a sequence of m nonessential ukk_1-derivation steps with a single character comparison is transformed into a single nonessential ukk_2-derivation step with m character comparisons, that are done in the same order. However, by the use of *scan*, the index i starts to advance through the string without extra calls to ukk_1. This is where we give up the on-line property. At the same time, this is where we gain the slight speed advantage of *mcc* over *ukk* [15] by eliminating successive calls to ukk_1 and *canonize*.

The next step is to eliminate the single nonessential steps in the derivation of the form $e_0 = ukk_2(\emptyset, \emptyset, (root, \varepsilon), 1) \Rightarrow e_1 \cdots \Rightarrow e_N = cst(t)$. Let $0 < k < N$ and assume that $e_k = ukk_2(T, L, (\bar{h}, q), j)$ is derived from the expression $e_{k-1} = ukk_2(T, L, (\bar{b}, \varepsilon), i)$ by an application of (9), where $((\bar{h}, q), j) = scan(T, \bar{b}, i)$. Since t has a sentinel, $j \leq n$. Let $q = \varepsilon$. Then we can derive e_{k+1} from e_k, using (10) or (11). Since $((\bar{h}, q), j) = scan(T, \bar{h}, j)$ we find that e_{k+1} equals the right-hand side of (10) or (11). Let $q \neq \varepsilon$. Then only (12) can be applied to e_k deriving $e_{k+1} = ukk_2(T \sqcup ((\bar{h}, q), j), L', (\bar{b'}, u'), j)$, where $(\bar{b'}, u') = link(T, L, (\bar{h}, q))$ and $L' = L \cup \{(\overline{hq}, \overline{b'u'})\}$. Hence for $q \neq \varepsilon$ the nonessential step from e_{k-1} to e_k can be merged with the step from e_k to e_{k+1}, if we substitute (9), yielding the following equivalent function ukk_3:

$$ukk_3(T, L, (\bar{b}, \varepsilon), i) = \begin{cases} T, & \text{if } i = n+1, \quad (13)\\ ukk_3(T \sqcup ((\bar{h}, q), j), L, (\bar{h}, q), j+1), & \text{else if } (\bar{h}, q) = (root, \varepsilon), \quad (14)\\ ukk_3(T \sqcup ((\bar{h}, q), j), L, (L(\bar{h}), \varepsilon), j), & \text{else if } q = \varepsilon, \quad (15)\\ ukk_3(T \sqcup ((\bar{h}, q), j), L', (\bar{b'}, u'), j), & \text{otherwise}, \quad (16) \end{cases}$$

where $((\bar{h}, q), j) = scan(T, \bar{b}, i)$, $(\bar{b'}, u') = link(T, L, (\bar{h}, q))$, and $L' = L \cup \{(\overline{hq}, \overline{b'u'})\}$.

$$ukk_3(T, L, (\bar{b}, cw), i) = ukk_3(T \sqcup ((\bar{b}, cw), i), L', (\bar{b'}, u'), i), \quad (17)$$

where $(\bar{b'}, u') = link(T, L, (\bar{b}, cw))$ and $L' = L \cup \{(\overline{bcw}, \overline{b'u'})\}$.

Obviously, ukk_3-derivation steps are always essential. Furthermore, the transformation from ukk_2 to ukk_3 does not affect the linear-time complexity. As the next step we synthesize the definition of a function *mcc* with the following properties:

$$mcc(T, L, (\bar{b}, u), i) = \begin{cases} ukk_3(T, L, (\bar{b}, u), i+1), & \text{if } (\bar{b}, u) = (root, \varepsilon), \\ ukk_3(T, L, (L(\bar{b}), u), i), & \text{if } \bar{b} \neq root \text{ and } u = \varepsilon, \\ ukk_3(T, L', (\bar{b'}, u'), i) & \text{if } u \neq \varepsilon, \end{cases}$$

where $(\bar{b'}, u') = link(T, L, (\bar{b}, u))$ and $L' = L \cup \{(\overline{bu}, \overline{b'u'})\}$.

Consider the following cases:

Case 1: $(\bar{b}, u) = (root, \varepsilon)$. Then $mcc(T, L, (\bar{b}, u), i) = ukk_3(T, L, (\bar{b}, u), i+1)$. If $i = n$, then $ukk_3(T, L, (\bar{b}, u), i+1)$ reduces to T by an application of (13). If $i < n$, then let $((h, q), j) = scan(T, \bar{b}, i+1)$. Now $ukk_3(T, L, (\bar{b}, u), i+1)$ reduces to

$$\begin{array}{ll} ukk_3(T \sqcup ((\bar{h}, q), j), L, (\bar{h}, q), j+1), & \text{if } (\bar{h}, q) = (root, \varepsilon), \\ ukk_3(T \sqcup ((\bar{h}, q), j), L, (L(\bar{h}), \varepsilon), j), & \text{if } \bar{h} \neq root \text{ and } q = \varepsilon, \\ ukk_3(T \sqcup ((\bar{h}, q), j), L', (\bar{b'}, u'), j), & \text{if } q \neq \varepsilon, \end{array}$$

where $(\bar{b'}, u') = link(T, L, (\bar{h}, q))$ and $L' = L \cup \{(\overline{hq}, \overline{b'u'})\}$. By definition, the three expressions are equal to $mcc(T \sqcup ((\bar{h}, q), j), L, (\bar{h}, q), j)$.

Case 2: $\bar{b} \neq root$ *and* $u = \varepsilon$. Then $mcc(T, L, (\bar{b}, u), i) = ukk_3(T, L, (L(\bar{b}), \varepsilon), i)$. If we let $((h, q), j) = scan(T, L(\bar{b}), i)$, then $ukk_3(T, L, (L(\bar{b}), \varepsilon), i)$ reduces to

$$\begin{array}{ll} ukk_3(T \sqcup ((\bar{h}, q), j), L, (\bar{h}, q), j+1), & \text{if } (\bar{h}, q) = (root, \varepsilon) \\ ukk_3(T \sqcup ((\bar{h}, q), j), L, (L(\bar{h}), \varepsilon), j), & \text{if } \bar{h} \neq root \text{ and } q = \varepsilon, \\ ukk_3(T \sqcup ((\bar{h}, q), j), L', (\bar{b'}, u'), j), & \text{if } q \neq \varepsilon, \end{array}$$

where $(\bar{b'}, u') = link(T, L, (\bar{h}, q))$ and $L' = L \cup \{(\overline{hq}, \overline{b'u'})\}$. By definition, the three expressions are equal to $mcc(T \sqcup ((\bar{h}, q), j), L, (\bar{h}, q), j)$.

Case 3: $u \neq \varepsilon$. Then $mcc(T, L, (\bar{b}, u), i) = ukk_3(T, L', (\bar{b'}, u'), i)$ where $(\bar{b'}, u') = link(T, L, (\bar{b}, u))$ and $L' = L \cup \{(\overline{bu}, \overline{b'u'})\}$. Consider the following subcases:

- $u' = \varepsilon$. Let $((\bar{h}, q), j) = scan(T, \bar{b'}, i)$. Then $ukk_3(T, L', (\bar{b'}, u'), i)$ reduces to

  $$\begin{array}{ll} ukk_3(T \sqcup ((\bar{h}, q), j), L', (\bar{h}, q), j+1), & \text{if } (\bar{h}, q) = (root, \varepsilon), \\ ukk_3(T \sqcup ((\bar{h}, q), j), L', (L(\bar{h}), \varepsilon), j), & \text{if } \bar{h} \neq root \text{ and } q = \varepsilon, \\ ukk_3(T \sqcup ((\bar{h}, q), j), L'', (\bar{b''}, u''), j), & \text{if } q \neq \varepsilon, \end{array}$$

  where $(\bar{b''}, u'') = link(T, L', (\bar{h}, q))$ and $L'' = L' \cup \{(\overline{hq}, \overline{b''u''})\}$. By definition, the three expressions are equal to $mcc(T \sqcup ((\bar{h}, q), j), L', (\bar{h}, q), j)$.
- $u' \neq \varepsilon$. Let $(\bar{b''}, u'') = link(T, L', (\bar{b'}, u'))$ and $L'' = L' \cup \{(\overline{b'u'}, \overline{b''u''})\}$. Then $ukk_3(T, L', (\bar{b'}, u'), i)$ reduces to $ukk_3(T \sqcup ((\bar{b'}, u'), i), L'', (\bar{b''}, u''), i)$, which equals $mcc(T \sqcup ((\bar{b'}, u'), i), L', (\bar{b'}, u'), i)$.

Putting it all together we get the following definition of *mcc*:

$$mcc(T, L, (\bar{b}, u), i) = \begin{cases} T, & \text{if } i = n \text{ and } (\bar{b}, u) = (root, \varepsilon), \\ mcc(T \sqcup ((\bar{h}, q), j), L, (\bar{h}, q), j), & \text{else if } u = \varepsilon, \\ mcc(T \sqcup ((\bar{h}, q), j), L', (\bar{h}, q), j), & \text{else if } u' = \varepsilon, \\ mcc(T \sqcup ((\bar{b}', u'), i), L', (\bar{b}', u'), i), & \text{otherwise,} \end{cases}$$

where $(\bar{b}', u') = link(T, L, (\bar{b}, u))$, $L' = L \cup \{(\overline{bu}, \overline{b'u'})\}$, and

$$((\bar{h}, q), j) = \begin{cases} scan(T, \bar{b}, i+1), & \text{if } (\bar{b}, u) = (root, \varepsilon), \\ scan(T, L(\bar{b}), i), & \text{else if } u = \varepsilon, \\ scan(T, \bar{b}', i), & \text{else if } u' = \varepsilon. \end{cases}$$

This definition of *mcc* is equivalent to the one we have developed directly in [15]. From the specification of *mcc* it is easy to see that the only difference between ukk_3 and *mcc* is that the computation of some information is delayed one step in *mcc*. There is no difference in the order or number of computation steps.

PROPOSITION 5.4. *For $n = |t|$, $mcc(\{root \xrightarrow{(1,n)} \bar{t}\}, \emptyset, (root, \varepsilon), 1)$ returns $cst(t)$ in $\mathcal{O}(n)$ time.*

PROOF. By construction of *mcc*, we have

$$\begin{aligned} mcc(\{root \xrightarrow{(1,n)} \bar{t}\}, \emptyset, (root, \varepsilon), 1) &= ukk_3(\{root \xrightarrow{(1,n)} \bar{t}\}, \emptyset, (root, \varepsilon), 2) \\ &= ukk_3(\emptyset, \emptyset, (root, \varepsilon), 1) \\ &= ukk(\emptyset, \emptyset, (root, \varepsilon), 1) \\ &= cst(t). \end{aligned}$$

Since *mcc* is derived from *ukk* by eliminating nonessential derivation steps, without affecting the number or order of essential steps, *mcc* inherits the linear-time property. □

5.2. *Synchronization Points Between ukk and mcc.* We call "*point i*" the situation after

- *ukk* has constructed the suffix tree $cst(t_1 \cdots t_i)$,
- *mcc* has constructed the $\mathcal{A}^+$-tree $T(t_i \cdots t_n)$, i.e., the suffix $t_i \cdots t_n$ has just been inserted.

At this point, *ukk* has read no character of t beyond t_i. If t_i does not occur to the left in t, it behaves as a sentinel for $t_1 \cdots t_i$, and both *ukk* and *mcc* will have constructed $cst(t_1 \cdots t_i)$, and no character beyond t_i has been read. However, generally, *mcc* has scanned further in t. We call the additional characters read by *mcc* its *lookahead at point i*.

PROPOSITION 5.5. *Let $cw = tail(t_i \ldots t_n)$ for some character c and some string w. The lookahead of mcc at point i is*

(i) ε, *if* $head(t_i \cdots t_n) = \varepsilon$,
(ii) uc, *if* $head(t_i \cdots t_n) = t_i u$.

PROOF. Intuitively, it is clear that t need not be scanned beyond uc, in order to insert the new open edge $\overline{t_i u} \xrightarrow{c\cdots}$. Formally, this can be verified against the implementation of *mcc* given in Section 5.1. □

What does this mean with respect to practical matters? On-line construction is attractive when the suffix tree is intended to be used to search for *first* occurrences of words in t. If a word occurs in t, only the suffix tree for the prefix of t ending with the first occurrence must be constructed. Further queries may further expand the tree. Thus, suffix tree construction time is amortized over a series of queries. This is the practical advantage of *ukk* being an on-line algorithm.

With the complete input string available—say as a character file—it does not really matter whether the partial tree construction stops exactly after the first occurrence of the search key, or some characters beyond it. *mcc* may as well be interleaved with queries for first occurrences, and, in this sense, it shares the advantages of a truly on-line construction. On the other hand, when t is incrementally calculated by some other computation—say as a character stream—then the difference matters: *ukk* is more lazy than *mcc*, and the extra characters called for by *mcc* may induce an overhead of arbitrary dimension.

6. An Explanation of Weiner's Algorithm. In this section we go back to the roots and take a look at the "Algorithm of the Year 1973" (D. E. Knuth according to [19]).

Our explanation of *wrf* is quite different from the treatment by Chen and Seiferas [10]. They restate Weiner's algorithm in a less technical, even prosaic, form. Our approach is to relate *wrf* to *ukk*. We explain *wrf* using today's terminology,[6] thus revealing its close relation to the algorithms discussed in the previous sections.

6.1. *An Abstract Explanation.* Before we enter the detailed analysis, we first take a look at Weiner's algorithm in terms of our abstract tree construction operations of Section 3. *wrf* reads the input string t from right to left, and successively inserts suffixes, shortest first. Figure 6 shows how the suffix tree arises from a series of *add*/*split*-operations. As with *ukk* and *mcc*, the crux lies in the efficient way of moving from one insertion point to the next, e.g., from node $\bar{d}$ to node $\overline{ad}$. Having read through all the previous sections, you might say: well—just follow the suffix link $\overline{ad} \longrightarrow \bar{d}$ in reverse direction! This idea is not totally wrong, but the general case is not as simple, and, besides, reverse links are more expensive, and they usually exist only *after* we needed them ....

6.2. *Traversing a Tree That We Do Not Construct.* Assume it is 1973 and little is known about suffix trees. The first natural thing to think of is on-line construction,

[6] Weiner [26] calls the suffix tree prefix tree, and vice versa, and the overall treatment is very technical.

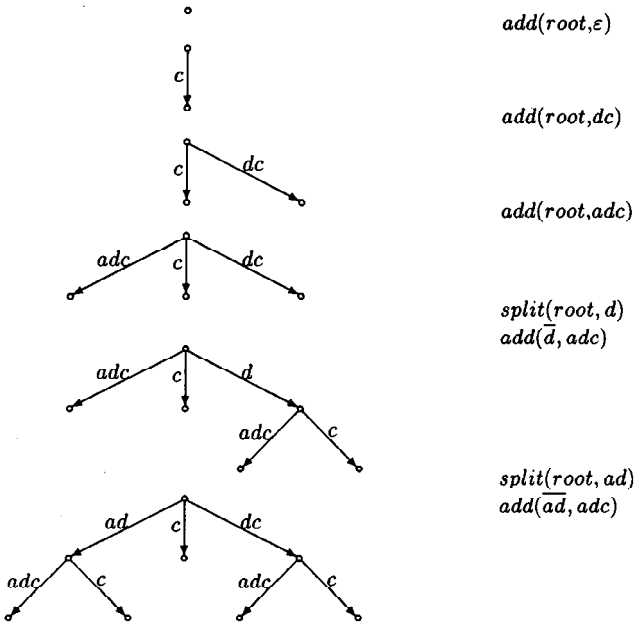

Fig. 6. Sequence of trees produced by *wrf* for $t = adadc$.

successively building the tree for longer and longer prefixes of t. However, a problem arises immediately: existing leaf edges will have to be extended for each new character, leading to an $\mathcal{O}(n^2)$-algorithm. Since this problem will only be solved by Ukkonen's open edges in 1992, we instead process t from right to left. This way, leaves will always represent a suffix and need to be changed less frequently. This decision is quite logical—but it will bring us into tremendous difficulties shortly. Anyway, we will be building the tree successively for longer and longer suffixes of t, so we have an on-line property in the reverse direction. We call this the anti-on-line property.

Suppose as is a suffix of t. To obtain $cst(as)$ from $cst(s)$, a naive anti-on-line algorithm determines the longest prefix u of as that is an s-word. This is accomplished by walking down the path for as in $cst(s)$ as far as possible. Let $uv = as$. One of the following cases will arise:

1. If $\bar{u}$ is a leaf in $cst(s)$, then the leaf edge $\bar{y} \xrightarrow{x} \bar{u}$ is replaced by the leaf edge $\bar{y} \xrightarrow{xv} \overline{uv}$.
2. If $\bar{u}$ is not a leaf in $cst(s)$, then the algorithm splits for $\bar{u}$ if necessary, and adds an edge $\bar{u} \xrightarrow{v} \overline{uv}$.

Later authors will suggest similar naive versions of other algorithms [18], [14], and it will be shown that their efficiency is $\mathcal{O}(n \log n)$ in the expected case [6]. The factor $\log n$ comes from walking the tree from the root to the point of insertion. For an $\mathcal{O}(n)$-algorithm we must access this point in $\mathcal{O}(1)$. As the string u above equals $\alpha^{-1}(as)$, our problem is solved, if for each iteration we can hop directly from one active prefix node to the next, and of course, we know exactly where it is:

PROPOSITION 6.1. *There is an edge* $\bar{u} \xrightarrow{vw} \bar{s}$ *in* $cst(s)$, *s.t.* $\alpha^{-1}(s) = uv$ *and* w *is nonempty.*

PROOF. Let u be the longest prefix of s that is right-branching in s. Then u is a nested prefix of s. Thus we can conclude $\alpha^{-1}(s) = uv$ for some string v. Moreover, there is a nonempty string w, s.t. $\alpha^{-1}(s)w = s$. Hence there is an edge $\bar{u} \xrightarrow{vw} \bar{s}$ in $cst(s)$. □

So the "old" active prefix is always at hand, just above the leaf inserted in the previous step, but how do we hop to the "new" one in $\mathcal{O}(1)$?

Glancing ahead into the future, we see Ukkonen's on-line algorithm swinging easily from active suffix to active suffix, using the suffix links. We are doing an anti-on-line construction, and are interested in active prefixes. If only we had $cst(s^{-1})$ available! Since $\alpha^{-1}(as)$ is a prefix of $a\alpha^{-1}(s)$ we could then determine $\alpha^{-1}(as)$ by following the (reverse) prefix links, shortening $\alpha^{-1}(s)$ from the right, until we find a prefix p, such that ap^{-1} occurs in $cst(s^{-1})$. If such a p exists, then $\alpha^{-1}(as) = ap^{-1}$. Otherwise $\alpha^{-1}(as) = \varepsilon$.

EXAMPLE 6.2. Let $s = bcdeabcbcdfbcde$. Figure 7 shows the relevant parts in $cst(s^{-1})$ (plus the new outgoing a-edge from node $\overleftarrow{bcd}$) and the way from $\alpha(s^{-1}) = edcb$ to $\alpha((as)^{-1}) = cba$. For the sake of comparison with Figure 8, reference pairs are written from right to left.

Of course, we cannot simply construct the reverse prefix tree, since this is the dual of the problem we started to solve. However, with some additional effort, we can use the suffix tree to simulate the above walk through the reverse prefix tree! This is the essential idea of Weiner's algorithm, and at the same time the reason for its extra complexity.

We consult Proposition 2.12. From Statements 1 and 2 of Proposition 2.12 we know that (and how) $(cst(s))^{-1}$ approximates the reverse prefix tree. From Statement 3 we learn that the reversed edges of $cst(s)$ are the suffix links of $(cst(s))^{-1}$, i.e., they approximate the prefix links! Thus if we make the edges of $cst(s)$ bidirectional, this will be sufficient to approach $\alpha^{-1}(as)$ from $\alpha^{-1}(s)$.

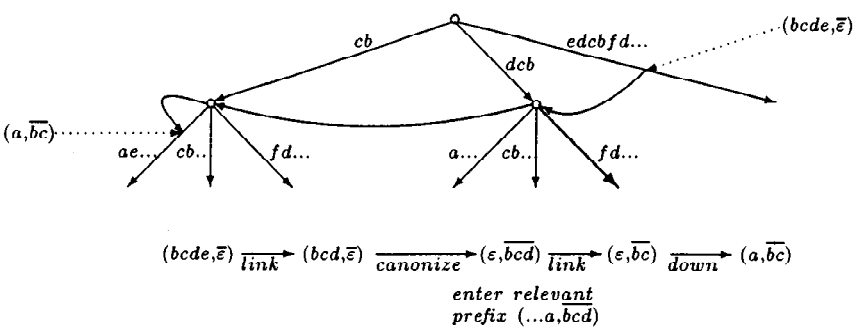

Fig. 7. The way from $\alpha(s^{-1}) = edcb$ to $\alpha((as)^{-1}) = cba$.

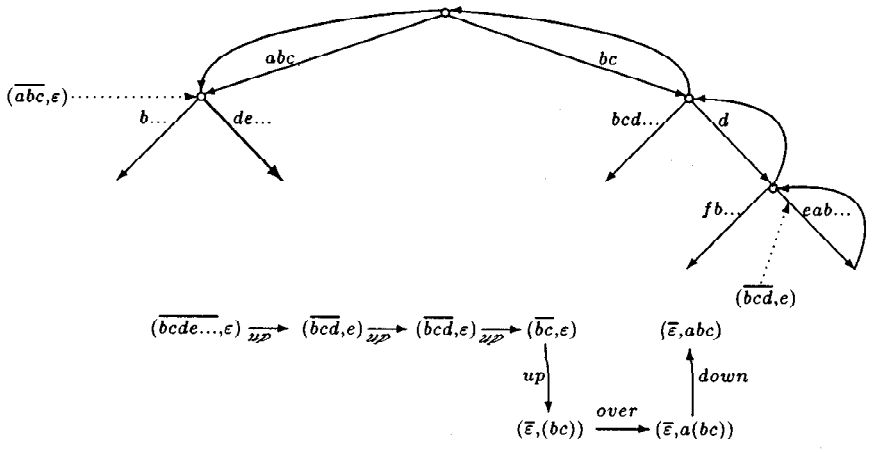

Fig. 8. Relevant parts of $cst(s)$.

A final problem remains, and its solution is less elegant and much more expensive. After all relevant suffixes are inserted, Ukkonen's algorithm follows an edge in the suffix tree downward. By analogy, we need to walk along a prefix edge once before $\alpha^{-1}(as)$ is reached. This means we must make an additional effort to record prefix edges between the nodes of $cst(s)$. Summing up, we need the following extra information:

(1) The edges of $cst(s)$ must be bidirectional, such that we can traverse them upward.
(2) For each node in $cst(s)$ and each $a \in \mathcal{A}$ we must indicate whether this node would have an a-edge in $cst(s^{-1})$. We call this a pending prefix edge.
(3) If the target node of this edge also happens to be a node in $cst(s)$, then we record this as a proper prefix edge.

This is how we now simulate the traversal of prefix links in $cst(s^{-1})$ by using $cst(s)$ and this auxiliary information: the traversal starts at the leaf below the active prefix, i.e., at $\bar{s}$, and moves upward until an a-prefix edge is indicated. If the edge is pending, we must take a detour higher up in the tree, recording its length (in characters), until we hit a node which has a proper a-prefix edge. We follow this edge, and then proceed downward in $cst(s)$ according to the recorded length of the detour.

EXAMPLE 6.3. Let $s = bcdeabcbcdfbcde$ as in Example 6.2. Figure 8 shows the relevant parts in $cst(s)$ and the way from $\alpha^{-1}(s) = bcde$ to $\alpha^{-1}(as) = abc$. Node $\overline{bc}$ is the one with the pending prefix edge, where the detour $(up, over, down)$ starts. The extra parentheses around bc indicate the characters which account for the length of the detour. It is not typical that this traversal passes the *root*, but an even more sophisticated example would be necessary to demonstrate this.

During this traversal, we must also create and update the extra information, and make sure that we can do all this in $\mathcal{O}(1)$ on the average (see Section 6.3).

Summing up, we may say that Weiner's algorithm has a touch of tragedy and heroism:

faced with the problem of growing leaf-edges, it turns to anti-on-line construction. This means having to traverse the reverse prefix tree while only the suffix tree is to be constructed. This adds an almost unsurmountable amount of difficulty—but *wrf* succeeds in handling it within the linear-time constraint.

6.3. *Extra Costs of wrf.* Here we detail the extra costs in time and space that result from *wrf*'s simulated traversal of the reverse prefix tree. Recall the extra information required by *wrf* (see items (1)–(3) preceding Example 6.3).

The extra pointer needed for (1) is equivalent to the effort of storing suffix links in *ukk* and *mcc*. It is the extra data structures for pending and proper prefix edges which make *wrf* more space consuming, and their maintainance makes it slower than the others.

Still, this extra information can be maintained with a fixed effort per node visited. When $\bar{u} \xrightarrow{vw} \overline{uvw}$ is split for $\overline{uv}$, this node inherits its proper and pending prefix edges from the leaf $\overline{uvw}$. The new leaf, representing the longest suffix, naturally has no prefix edges when created. Finally, prefix edges of a node on the path from $\alpha^{-1}(s)$ upward must be updated. They all have a pending a-prefix edge now, and if there is an explicit node $\overline{uv}$, we record the proper prefix edge $\overline{uv} \xrightarrow{a} \overline{auv}$.

At this point, we are left with one final question: while a traversal along prefix links can be easily shown to add up to $\mathcal{O}(n)$ node visits overall, it is not obvious that the same is true when we traverse the suffix tree instead. Our "detour" may take us up all the way to the root, and back down. This is also exemplified in Figure 8. In fact, if this happened at each iteration, we would essentially be back at the naive anti-on-line algorithm. How can we prove that this form of traversal does not visit more than $\mathcal{O}(n)$ nodes in total?

LEMMA 6.4. *As above, let $\bar{u}$ be the node encountered which has a proper a-prefix edge, and let $auv = \alpha^{-1}(s)$. Then there is no node between $\overline{au}$ and $\overline{auv}$.*

PROOF. The only possibility for a node between $\overline{au}$ and $\overline{auv}$ is when $v = xy$, $x \neq \varepsilon$, $y \neq \varepsilon$, and $\overline{aux}$ is an explicit node. It is an inner node, and the remark after Proposition 2.9 applies. So its suffix link points to the node $\overline{ux}$, which then has a proper a-prefix edge. This contradicts the definition of $\bar{u}$ being the first such node on the traversal. □

So from the "summit" $\bar{u}$ of the detour we descend at most one node. We now consider the depth of the nodes (from the root) visited: it is first decreased by the detour, and then increased by at most 1. The tree has $\mathcal{O}(n)$ nodes. Since the sum of all increases is bounded by n, the decreases cannot add up to more than $2n$. Hence the number of nodes visited over all detours is $\mathcal{O}(n)$.

7. Conclusion. We have reached the end of our investigation, and the conclusion is clear: the three suffix tree constructions considered—*wrf*, *mcc*, and *ukk*—are more closely related than is commonly assumed. While all three are $\mathcal{O}(n)$-algorithms, their relative virtues are different:

- *ukk* is on-line, the most elegant construction, and the clue to understanding the others.
- *mcc* is the most efficient construction, by a small margin over *ukk*.

- *wrf* has no practical virtue (it uses significantly more time and space), but remains a true historic monument in the area of string processing.

The notion of active suffixes, suffix links, and the duality between suffix link trees and prefix trees are the cardinal points of linear-time suffix tree construction. Although there is no truly formal way to express this, we conjecture that any sequential suffix tree construction not based on these concepts will fail to meet the $\mathcal{O}(n)$-criterion. This does not pertain to parallel constructions like [20].

Acknowledgments. Gene Lawler encouraged us to exploit our duality observation for explaining suffix tree construction. Dan Gusfield and Richard Karp directed our attention to the manuscript by Pratt [19]. Dan Gusfield also provided a carefully written exposition of Weiner's algorithm. Many discussions with Esko Ukkonen improved our understanding of suffix trees. The careful comments of the referees gave valuable hints to improve the exposition. All their contributions are truly appreciated.

References

[1] A. Aho. Algorithms for Finding Patterns in Strings. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science*, Volume A, pages 257–300. Elsevier, Amsterdam, 1990.
[2] A. Aho and M. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, **18**:333–340, 1975.
[3] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms*. Addision-Wesley, Reading, MA, 1982.
[4] A. Apostolico. The Myriad Virtues of Subword Trees. In [5], pages 85–96, 1985.
[5] A. Apostolico and Z. Galil. *Combinatorial Algorithms on Words*. Springer-Verlag, New York, 1985.
[6] A. Apostolico and W. Szpankowski. Self-Alignments in Words and Their Applications. *Journal of Algorithms*, **13**:446–467, 1992.
[7] R.A. Baeza-Yates. String Searching Algorithms. In W. Frakes and R.A. Baeza-Yates, editors, *Information Retrieval*: *Algorithms and Data Structures*, pages 219–240. Prentice-Hall, Englewood Cliffs, NJ, 1992.
[8] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M.T. Chen, and J. Seiferas. The Smallest Automaton Recognizing the Subwords of a Text. *Theoretical Computer Science*, **40**:31–55, 1985.
[9] W.I. Chang and E.L. Lawler. Sublinear Approximate String Matching and Biological Applications. *Algorithmica*, **12**(4/5):327–344, 1994.
[10] M.T. Chen and J.I. Seiferas. Efficient and Elegant Subword Tree Construction. In [5], pages 97–107, 1985.
[11] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.
[12] M. Crochemore. String Matching with Constraints. In *Proceedings of the* 1988 *International Symposium on Mathematical Foundations of Computer Science*, pages 44–58. Lecture Notes in Computer Science, Volume 324. Springer-Verlag, Berlin, 1988.
[13] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, Oxford, 1994.
[14] R. Giegerich and S. Kurtz. Suffix Trees in the Functional Programming Paradigm. In *Proceedings of the European Symposium on Programming* (*ESOP* '94), pages 225–240. Lecture Notes in Computer Science, Volume 788. Springer-Verlag, Berlin, 1994.
[15] R. Giegerich and S. Kurtz. A Comparison of Imperative and Purely Functional Suffix Tree Constructions. *Science of Computer Programming*, **25**(2-3):187–218, 1995.
[16] G.H. Gonnet and R.A. Baeza-Yates. *Handbook of Algorithms and Data Structures in Pascal and C*. Addison-Wesley, Reading, MA, 1991.

[17] S. Kurtz. Fundamental Algorithms for a Declarative Pattern Matching System. Dissertation, Technische Fakultät, Universität Bielefeld, available as Report 95-03, July 1995.
[18] E.M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, **23**(2):262–272, 1976.
[19] V.R. Pratt. Improvements and Applications of the Weiner Repetition Finder. Unpublished manuscript, Cambridge, MA, 1973.
[20] S.C. Sahinalp and U. Vishkin. Symmetry Breaking for Suffix Tree Construction. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pages 300–309, 1994.
[21] R. Sedgewick. *Algorithms*. Addision-Wesley, Reading, MA, 1988.
[22] G.A. Stephen. *String Searching Algorithms*. World Scientific, Singapore, 1994.
[23] J. Stoye. Affixbäume. Master's Thesis (in German), Technische Fakultät, Universität Bielefeld, 1995.
[24] E. Ukkonen. Constructing Suffix Trees On-line in Linear Time. *Algorithms, Software, Architecture*. In van Leeuven, J., editor, *Information Processing* 92, Volume I, pages 484–492, Elsevier, Amsterdam, 1992.
[25] E. Ukkonen. On-line Construction of Suffix-Trees (revised version of [24]). *Algorithmica*, **14**:249–260, 1995.
[26] P. Weiner. Linear Pattern Matching Algorithms. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.