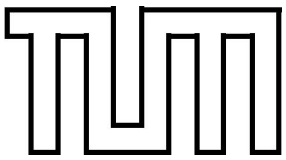
Suffix Trees and their Applications

Moritz Maaß

October 26, 1999



Technische Universität München
Fakultät für Informatik

Abstract

Suffix trees have many different applications and have been studied extensively. This paper gives an overview of suffix trees and their construction and studies two applications of suffix trees.

Contents

1 Introduction

Suffix trees are very useful for many string processing problems. They are data structures that are built up from strings and "really turn" the string "inside out" [GK97]. This way they provide answer to a lot of important questions in linear time (e.g. "Does text t contain a word ω?" in $O(n)$, the length of the word). There are many algorithms that can build a suffix tree in linear time.
The first algorithm was published by Weiner in 1973 [Wei73]. It reads a string t from right to left and successively inserts suffixes beginning with the shortest suffix, which leads to a very complex algorithm. Following Giegerich and Kurtz [GK97] the algorithm "has no practical value [...], but remains a true historic monument in the area of string processing." I will therefore not bother with a deeper study of this early algorithm.
Shortly after Weiner McCreight published his "algorithm M" in [McC76]. The algorithm is explained in detail in section 3.
A newer idea and a different intuition lead to an on-line algorithm by Ukkonen [Ukk95] that turns out to perform the same abstract operations as McCreight's algorithm but with a different control structure, which leads to the on-line ability but also to a slightly weaker performance [GK97]. The algorithm is explained in detail in section 4.
I will begin with the explanation of the involved data structures (2). I will mostly use the notation and definitions of [GK97]. The next sections will explain the two algorithms (3),(4) and make some remarks on the implementation (5).
At the end I will present two basic applications of suffix trees, string assembling (6), and data compression (7), and present some available alternatives to as well as some other applications for suffix trees (8).
I will finish with a short conclusion (9).

2 Data Structures

2.1 Notation

alphabet

Let Σ be a non empty set of characters, the *alphabet*. A (possibly empty) sequence of characters from Σ will be denoted like r, s, and t. t^{-1} will denote the reversed string of t. A single letter will be shown like $\mathtt{x}$, $\mathtt{y}$, or $\mathtt{z}$. ε is the empty string. The actual characters from Σ will be denoted as $\mathtt{a}$, $\mathtt{b}$, ...
For now the size of Σ will be assumed constant and independently bound of the length of any input string encountered.
Let $|t|$ denote the length of a string t. Let Σ^m be all strings with length m, $\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^i$, and $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$.

prefix

A *prefix* w of a string t is a string such that $wv = t$ for a (possibly empty) string v. I will write $w \sqsubset_p t$. The prefix is called proper if $|v|$ is not zero.

suffix

A *suffix* w of a string t is a string such that $vw = t$ for a (possibly empty) string v. I will write $w \sqsupset_s t$. The suffix is called proper if $|v|$ is not zero.

factor

Similar, a *factor* w of a string is a substring of string t such that $vwx = t$ for (possibly empty) strings v and x. I will write $w \sqsubset\!\sqsupset_f t$.

right-branching

A factor w of a string t is called *right-branching* if t can be decomposed as $uwxv$ and $u'wyv'$ for some strings u, v, u', and v', and where $\mathtt{x} \neq \mathtt{y}$. A *left-branching* factor is defined analogous.

left-branching

2.2 Σ^+-Trees and suffix trees.

Definition 1. *(Σ^+-tree) A Σ^+-tree $\mathbb{T}$ is a tree with a root and edge labels from Σ^+ . For each $\mathtt{a} \in \Sigma$, every node in $\mathbb{T}$ has at most one edge, whose label starts with $\mathtt{a}$. For an edge from t to s with label v we will write $t \xrightarrow{v} s$.*

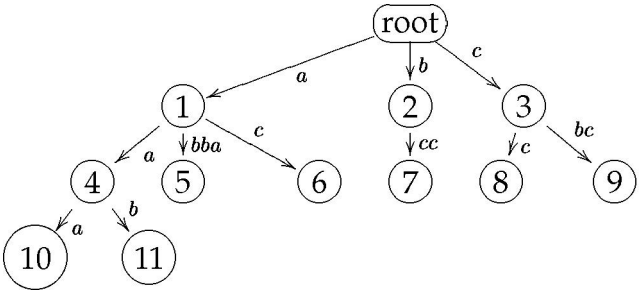


Figure 1: Example of a Σ^+-tree

location

If $\mathbb{T}$ is a Σ^+-tree with the node k then we let $path(k)$ be the string that is the concatenation of all edge labels from the *root* to k. In the example in figure 1 the paths of node 5 and node 7 are $path(5) = \mathtt{abba}$ and $path(7) = \mathtt{bcc}$. We will call $\overline{w}$ the *location* of w, that is $path(\overline{w}) = w$.

subtree

Since every branch is unique, if $path(t) = w$ we can denote a node t $\overline{w}$. In the example $\overline{\mathtt{aaa}}$ refers to node 10. We let the *subtree* at node $\overline{w}$ be $\mathbb{T}_{\overline{w}}$.

The words that are represented in a Σ^+-tree $\mathbb{T}$ are given by the set $words(\mathbb{T})$. A word w is in $words(\mathbb{T})$ if and only if there is a v such that $\overline{wv}$ is a node in $\mathbb{T}$.
If a string w is in $words(\mathbb{T})$ such that $w = uv$ and $\overline{u}$ is a node in $\mathbb{T}$ we will call $(\overline{u}, v)$ the *reference pair* of w with respect to $\mathbb{T}$. If u is the longest prefix such that $(\overline{u}, v)$ is a reference pair, we will call $(\overline{u}, v)$ the *canonical reference pair*. We will then write $\widehat{w} = (\overline{u}, v)$.
In our example $(\overline{\varepsilon}, \mathtt{ab})$ is a reference pair for $\mathtt{ab}$ and $\widehat{\mathtt{ab}} = (1, b)$ is the canonical reference pair.
A location $\widehat{w} = (\overline{u}, v)$ is called *explicit* if $|v| = 0$ and *implicit* otherwise.

reference pair
canonical reference pair
explicit
implicit

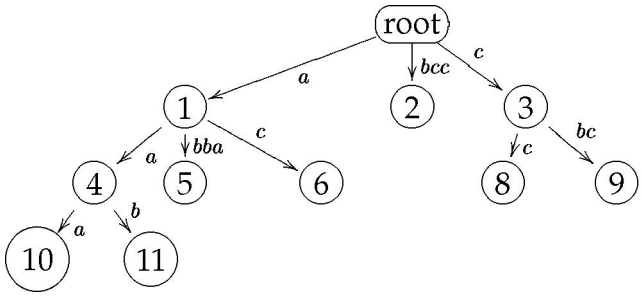


Figure 2: The compact Σ^+-tree for figure 1

Definition 2. *(Atomic and Compact Σ^+-tree) A Σ^+-tree $\mathbb{T}$ where every edge label consists only of a single character is called* atomic *(every location is explicit).*
A Σ^+-tree $\mathbb{T}$ where every node is either root*, a leaf or a branching node is called* compact.

atomic
compact

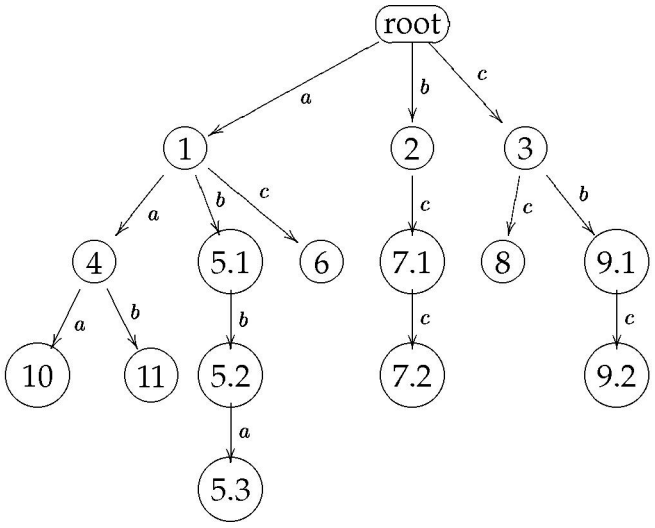


Figure 3: The atomic Σ^+-tree for figure 1

Atomic Σ^+-trees are also known as "tries" [Ukk95, UW93]. Atomic and compact Σ^+-trees are uniquely determined by the words they contain.

Figure 2 and figure 3 show the compact and the atomic Σ^+-tree for the example tree of figure 1.

suffix tree

Definition 3. *(Suffix Tree) A* suffix tree *for a string t is a Σ^+-tree $\mathbb{T}$ such that $words(\mathbb{T}) = \{w | w \text{ is a factor of } t\}$. For a string t the atomic suffix tree will be denoted by $ast(t)$, the compact suffix tree will be denoted by $cst(t)$.*

reverse prefix tree

The *reverse prefix tree* of a string t is the suffix tree of t^{-1}.

nested suffix

A *nested suffix* of t is a suffix that appears also somewhere else in t. The longest nested suffix is called the *active suffix* of t.

active suffix

Lemma 1. *(Explicit locations in the compact suffix tree) A location $\overline{w}$ is explicit in the compact suffix tree $cst(t)$ if and only if*

1. *w is a non nested suffix of t or*
2. *w is right branching.*

Proof. "$\Rightarrow$": If $\overline{w}$ is explicit, it can either be a leaf or a branching node or *root* (in which case $w = \varepsilon$ and w is a nested suffix of t).
If $\overline{w}$ is a leaf it is also a suffix of t. Then it must also be a non nested suffix, since if it were nested, it would appear somewhere else in t: $\exists v \sqsupset_s t : w \sqsubset_p v$. The node can then not be a leaf.
If $\overline{w}$ is a branching node then there must exist at least two outgoing edges from $\overline{w}$ with different labels. This means that there must exist two different suffixes u, v of t with $w \sqsubset_p u$ and $w \sqsubset_p v$, where $v = w\mathrm{x}s$ and $u = w\mathrm{x}'s'$ with $\mathrm{x} \neq \mathrm{x}'$. Hence w is right branching.
"$\Leftarrow$": If w is a non nested suffix of t, it must be a leaf. If w is right branching there are two suffixes u, v of t with $u = w\mathrm{x}s$ and $v = w\mathrm{x}'s'$, where $\mathrm{x} \neq \mathrm{x}'$, and w is a branching node. □

Now it is easy to see, why the decision whether a word w occurs in string t can be done in $O(|w|)$, since one just has to check if $\overline{w}$ is an (implicit) location in $cst(t)$.
The edge labels must be represented by pointers into the string to keep the size of the compact suffix tree to $O(n)$ (see section 2.4). The edge label (p, q) denotes the substring $t_p t_{p+1} \ldots t_q$ of t or the empty string ε if $p > q$.

open edges

Ukkonen [Ukk95] introduced so called *open edges* for leaf edges (edges that lead to a leaf). Since all leaves of a suffix tree extend to the end of the string, an open edge is denoted by (p, ∞) instead of $(p, |t|)$, where ∞ is always the maximal length, that is $|t|$. This way all leaves extend automatically to the right, when the string is extended.

Definition 4. *(Suffix Links) Let $\mathbb{T}$ be a Σ^+-tree. Let $\overline{\mathrm{a}w}$ be a node in $\mathbb{T}$ and let v be the longest suffix of w, such that $\overline{v}$ is a node in $\mathbb{T}$. An (unlabeled) edge from $\overline{\mathrm{a}w}$ to v is a* suffix link. *If $v = w$ it is called atomic.*

suffix link

Proposition 1. *In $ast(t)$ and $cst(t\$)$, where $\$ \notin t$, all suffix links are atomic.*

sentinel character

Proof. The character \$ is called a *sentinel character*. The first part follows from the definition, since all locations are explicit. To prove the second proposition we must show that for each node $\overline{aw}$, $\overline{w}$ is also a node in $cst(t)$. If $\overline{aw}$ is a node in $cst(t)$, it must be either a leaf or a branching node. If it is a leaf, w must be a non nested suffix of t. Because of the sentinel character, following lemma 1 all suffixes (including *root*, the empty suffix) are explicit, since only *root* is a nested suffix. Therefore $\overline{w}$ is also a leaf or *root*. If $\overline{aw}$ is a branching node, then aw is right branching and so is w. Hence $\overline{w}$ is explicit by lemma 1. □

As follows from this proof, the sentinel character guarantees the existence of leaves for all suffixes. With the sentinel there can be no nested suffixes but the empty suffix of t. If we drop the sentinel character some suffixes might be nested and their locations become implicit.

2.3 Suffix Links and Duality of Suffix Trees

The suffix links of a suffix tree $\mathbb{T}$ form a Σ^+-tree by themselves. We will denote this tree by $\mathbb{T}^{-1}$. It has a node $\overline{w^{-1}}$ for each node $\overline{w}$ in $\mathbb{T}$ and an edge from $\overline{w^{-1}}$ to $\overline{(vw)^{-1}}$ labeled v^{-1} for each suffix link from $\overline{vw}$ to $\overline{w}$ in $\mathbb{T}$.

Proposition 2. *$\mathbb{T}^{-1}$ is a Σ^+-tree.*

Proof. By contradiction: Suppose there is a node $\overline{w^{-1}}$ in the suffix link tree $\mathbb{T}^{-1}$ that has two a-edges. Then there must be two suffix links in the suffix tree $\mathbb{T}$ from $\overline{uaw}$ to $\overline{w}$ and from $\overline{vaw}$ to $\overline{w}$. Here $u \neq \varepsilon$ and $v \neq \varepsilon$ because if $\overline{aw}$ is explicit, the suffix links would point there instead of to $\overline{w}$.
If $\overline{uaw}$ or $\overline{vaw}$ are inner nodes, then uaw or vaw is right branching in t. But then aw must also be right branching and be an explicit location, which is a contradiction.
If $\overline{uaw}$ and $\overline{vaw}$ are leaves, uaw and vaw must be suffixes of t and so $uaw \sqsupset_s vaw$ or $vaw \sqsupset_s uaw$ in which case the suffix link from $\overline{vaw}$ must point to $\overline{uaw}$ or vice versa. □

Traversing the suffix link chain from $\overline{w}$ to *root* yields a path that is a factor of t^{-1}. Therefore the suffix link tree $\mathbb{T}^{-1}$ contains a subset of all words of the suffix tree of t^{-1}.

Proposition 3. $(ast(t))^{-1} = ast(t^{-1})$.

Proof. All suffix links of $ast(t)$ are atomic by proposition 1. Therefore we can simply deduce:
$\overline{w^{-1}} \xrightarrow{a} \overline{(aw)^{-1}}$ is an edge in $(ast(t))^{-1}$, iff $\overline{aw} \xrightarrow{a} \overline{w}$ is a suffix link in $ast(t)$, iff there are nodes $\overline{aw}$ and $\overline{w}$ in $ast(t)$, iff there are nodes $\overline{(aw)^{-1}}$ and $\overline{w^{-1}}$ in $ast(t^{-1})$, iff there is an edge $\overline{w^{-1}} \xrightarrow{a} \overline{(aw)^{-1}}$ in $ast(t^{-1})$. □

For the compact suffix tree there is the weaker duality that is proven in [GK97].

Proposition 4. $((cst(t))^{-1})^{-1} = cst(t)$.

Proof. See [GK97], proposition 2.12 (3) □

The further exploitation of this duality leads to *affix trees* that are studied in detail in [Sto95]. Unfortunately the aim of constructing and proving an $O(n)$ on-line algorithm for constructing an affix tree is not achieved by Stoye. affix trees

2.4 Space-Requirements of Suffix Trees

Proposition 5. *Compact suffix trees can be represented in $O(n)$ space.*

Proof. A suffix tree contains at most one leaf per suffix (exactly one with the sentinel character). Every internal node must be a branching node, hence every internal node has at least two children. Each branch increases the number of leaves by at least one, we therefore have at most n internal nodes and at most n leaves.
To represent the string labels of the edges we use indices into the original string as described above. Each node has at most one parent and so the total number of edges does not exceed $2n$.
Similar each node has at most one suffix link, so the total number of suffix links is also restricted by $2n$. □

As an example of a suffix tree with $2n - 1$ nodes consider the tree for $\mathtt{a}^n\$$.
The size of the atomic suffix tree for a string t is $O(n^2)$. (For an example see the atomic suffix tree for $\mathtt{a}^{n/2}\mathtt{b}^{n/2}$, which has $(n/2+1)^2$ nodes.)

3 McCreight's Algorithm

3.1 The Underlying Idea

I will first try to give the intuition and then describe McCreight's algorithm (*mcc*) in further detail.
mcc starts with an empty tree and inserts suffixes starting with the longest one. (Therefore *mcc* is not an on-line algorithm.)
mcc requires the existence of the sentinel character so that no suffix is the prefix of another suffix and there will always be a terminal node per suffix.
For the algorithm we will define suf_i to be the current suffix (in step i), and $head_i$ to be the longest prefix of suf_i, which is also a prefix of another suffix suf_j, where $j < i$. $tail_i$ is defined as $suf_i - head_i$.
The key idea to *mcc* is that, since we insert suffixes of decreasing lengths, there is a relation between $head_i$ and $head_{i-1}$ that can be used:

Lemma 2. *If $head_{i-1} = aw$ for letter a and (possibly empty) string w, then $w \sqsubset_p head_i$.*

Proof. Suppose $head_{i-1} = aw$. Then there is a $j - 1 < i - 1$ so that $aw \sqsubset_p suf_{i-1}$ and $aw \sqsubset_p suf_{j-1}$. Then $w \sqsubset_p suf_j$ and $w \sqsubset_p suf_i$, hence $w \sqsubset_p head_i$. □

We know the location of $head_{i-1} = aw$ and if we had the suffix links, we could easily move to the location of $w \sqsubset_p head_i$ without having to find our way from *root*. But $\overline{w}$ might not be explicit (if $\overline{head_{i-1}}$ wasn't explicit in the previous step) and the suffix link might not yet be set for $\overline{head_{i-1}}$.

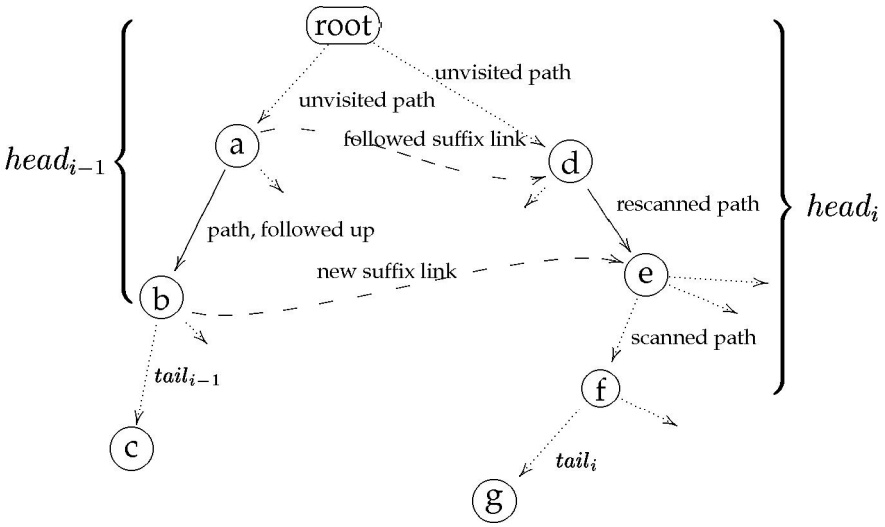


Figure 4: Suffix Link Usage by *mcc*

The solution found be McCreight is to find $\overline{head_i}$ by two steps of "rescanning" and "scanning". We just walk up the tree from $\overline{head_{i-1}}$ until we find a suffix link, follow it and then rescan the path we walked up back down to the location of w (which is

easy because we know the length of w and that its location exists, so we don't have to read the complete edge labels moving down the tree, but we can just check the start letter and the length).
Figure 4 shows this idea. Instead of trying to find the path from *root* to node f, the algorithm moves up to a, takes the suffix link to d, rescans its way to the (possibly implicit) location e, and only has to find the way to f letter by letter.

3.2 The Algorithm

The algorithm has three phases. First it determines the structure of the old header, finds the next available suffix link and follows it. Secondly it rescans the part of the previous header for which the length is known (called β). Thirdly it sets the suffix link for $head_{i-1}$, scans the rest of $head_i$ (called γ) and inserts a new leaf for $tail_i$. A branching node is constructed in the second phase of rescanning, if the location of w does not exist. In this case no scanning is needed, because if $head_i$ were longer than $\alpha\beta$, $\alpha\beta\gamma$ would be right branching, but because of lemma 2 $\alpha\beta$ is also right branching, so a node must already exist at $\overline{\alpha\beta}$. A node is constructed in the third phase, if the location $\overline{head_i}$ is not yet explicit.

Algorithm *mcc*
t is the given string.

1: $\mathbb{T} \longleftarrow emtpy\_tree$;
2: $head_0 \longleftarrow \varepsilon$;
3: $n \longleftarrow length(t)$;
4: **for** $i \longleftarrow 1$ to n **do**
5: find χ, α,β such that
a. $head_{i-1} = \chi\alpha\beta$,
b. if the parent of $\overline{head_{i-1}}$ is not root, it is $\overline{\chi\alpha}$, otherwise $|\alpha| = 0$,
c. $|\chi| \leq 1$ and $|\chi| = 0 \Leftrightarrow |head_{i-1}| = 0$.
6: **if** $|\alpha| > 0$ **then**
7: follow the suffix link from $\overline{\chi\alpha}$ to $\overline{\alpha}$;
8: **end if**
9: $\overline{\alpha\beta} \longleftarrow Rescan(\overline{\alpha}, \beta)$;
10: set suffix link from $\overline{head_{i-1}}$ to $\overline{\alpha\beta}$;
11: $(\overline{head_i}, tail_i) \longleftarrow Scan(\overline{\alpha\beta}, suf_i - \alpha\beta)$;
12: add leaf for $tail_i$;
13: **end for**

Note that if $|\alpha| = 0$ then $\overline{\alpha} = root$ and is known equally fast as by taking the suffix link in line 7.
Procedure *Rescan* finds the location of $\alpha\beta$. If $\overline{\alpha\beta}$ is not explicit yet, a new node is inserted. This case only occurs when the complete head is already scanned: If the head is longer (and the node already exists), $\alpha\beta$ must be the prefix of more than two suffixes and also left-branching in t. The $\overline{\alpha\beta}$ is only explicit, if it is a branching node already, and if $\alpha\beta$ were not left-branching then $head_{i-1}$ must have been longer because it would have met the longer prefix.

Procedure *Rescan*(n, β)
n is a node and β a string.
1: $l \longleftarrow 1$;
2: **while** $l \leq |\beta|$ **do**
3: find edge $e = n \xrightarrow{\omega} n'$ with $\omega_1 = \beta_l$;
4: **if** $l + |\omega| > |\beta| + 1$ **then**
5: $k \longleftarrow |\beta| - l + 1$;
6: split e with new node m and edges $n \xrightarrow{\omega_1 \ldots \omega_k} m$ and $m \xrightarrow{\omega_{k+1} \ldots \omega_{|\omega|}} n'$
7: return m;
8: **end if**
9: $l \longleftarrow l + |\omega|$;
10: $n \longleftarrow n'$;
11: **end while**
12: return n';

Procedure *Scan* searches deeper into the tree until it falls out and returns that position.

Procedure *Scan*(n, γ)
n is a node and γ a string.
1: $l \longleftarrow 1$;
2: **while** $\exists e = n \xrightarrow{\omega} n'$ with $\omega_1 = \gamma_l$ **do**
3: $k \longleftarrow 1$;
4: **while** $\omega_k = \gamma_l$ and $k \leq |\omega|$ **do**
5: $k \longleftarrow k + 1$;
6: $l \longleftarrow l + 1$;
7: **end while**
8: **if** $k > |\omega|$ **then**
9: $n \longleftarrow n'$
10: **else**
11: split e with new node m and edges $n \xrightarrow{\omega_1 \ldots \omega_{k-1}} m$ and $m \xrightarrow{\omega_k \ldots \omega_{|\omega|}} n'$
12: return $(m, \gamma_l \ldots \gamma_{|\gamma|})$
13: **end if**
14: **end while**
15: return $(n, \gamma_l \ldots \gamma_{|\gamma|})$

3.3 Complexity of *mcc*

Proposition 6. *mcc takes time* $O(n)$.

Proof. Let t be our underlying string and $n = |t|$. Algorithm *mcc*'s main loop executes n times, each step except for *Rescan* in line 8 and *Scan* in line 10 takes constant time.
Rescan takes time proportional to the number of nodes int_i it visits. The scanning takes place in a suffix res_i of the current suffix suf_i ($res_i = suf_i - \alpha$). Because every node encountered in rescanning already has a suffix link, there will be a non-empty

string ω (the edges label) that is in res_i but not in res_{i+1} for each node. Therefore $res_i \geq res_{i+1} + int_i$. Hence $\sum_{i=1}^{n} int_i \leq res_0 - res_1 + res_1 - res_2 \pm \ldots - res_n = res_0 - res_n = n$ and all calls to *Rescan* take time $O(n)$.
Scan cannot simply skip from node to node, but the characters in between are looked at one by one. Let $\gamma_{(i)} = head_i - \alpha_{(i)}\beta_{(i)}$. Then $|\gamma_{(i)}|$ is the number of characters scanned. By definition of α and β $|head_i| = |head_{i-1}| + |\gamma| - 1$. Hence the total number of characters scanned in all calls to *Scan* is $\sum_{i=1}^{n} |\gamma_{(i)}| = \sum_{i=1}^{n} (|head_i| - |head_{i-1}| + 1) = |head_n| - |head_0| + n = n$. □

4 Ukkonen's Algorithm

Ukkonen developed his $O(n)$ suffix tree algorithm from a different (and more intuitive) idea. He was working on string matching automata and his algorithm is derived from one that constructs the atomic suffix tree (sometimes referred to as "trie"). I will therefore shortly describe that algorithm and then derive Ukkonen's algorithm.

4.1 On-Line Construction of Atomic Suffix Trees

As shown in section 2.4 the atomic suffix tree has size $O(n^2)$. Therefore any algorithm needs at least that time to build the atomic suffix tree. The following algorithm lengthens the suffixes by one letter in each step. Every intermediate tree after steps i is the atomic suffix tree for $t_1 \dots t_i$. The algorithm starts inserting new leaves at the longest suffix, and follows the suffix links to the shorter suffixes until it reaches a node, where the corresponding edge already exists. To ensure termination of this loop an extra node, the "negative" position of all characters, $\perp$ is introduced. We have a suffix link from *root* to $\perp$ and a x labeled edge from $\perp$ to *root* for every $x \in \Sigma$.

Algorithm *ast*
t is the given string.

1: $T \longleftarrow emtpy\_tree$;
2: add *root* and $\perp$ to T.
3: **for all** $x \in \Sigma$ **do**
4: add edge $\perp \xrightarrow{x} root$
5: **end for**
6: `suffix_link`$(root) \longleftarrow \perp$;
7: $n \longleftarrow length(t)$;
8: $longest\_suffix \longleftarrow root$;
9: $previous\_node \longleftarrow root$;
10: **for** $i \longleftarrow 1$ to n **do**
11: $current\_node \longleftarrow longest\_suffix$;
12: **while** $\nexists e = current\_node \xrightarrow{t_i} temp\_node$ **do**
13: add edge $current\_node \xrightarrow{t_i} new\_node$ with new node new_node;
14: **if** $current\_node = longest\_suffix$ **then**
15: $longest\_suffix \longleftarrow new\_node$;
16: **else**
17: `suffix_link`$(previous\_node) \longleftarrow new\_node$;
18: **end if**
19: $previous\_node \longleftarrow new\_node$;
20: $current\_node \longleftarrow$ `suffix_link`$(current\_node)$
21: **end while**
22: `suffix_link`$(previous\_node) \longleftarrow temp\_node$;
23: **end for**

4.2 From *ast* to *ukk*: On-Line Construction of Compact Suffix Trees

The above given algorithm will now be transfered into an $O(n)$ algorithm for constructing compact suffix trees. In step $i+1$ *ast* adds a new node and a new edge to all nodes on the boundary path. The *boundary path* is the path from the longest suffix along the suffix links until a t_{i+1} labeled edge is found. Let s_j be the factor $t_j \dots t_i$ of t, and let $\overline{s_j}$ be its location in $ast(suf_i)$, with $\overline{s_{i+1}} = root$ and $\overline{s_{i+2}} = \perp$. Let l be the largest index, such that for all nodes $\overline{s_j}$, $j < l$, a node and an edge is inserted. Now let k be the largest index, such that for all nodes $\overline{s_j}$, $j < k$, $\overline{s_j}$ is a leaf. After step i the longest suffix $\overline{suf_i} = s_1$ is a leaf and $\perp = s_{i+2}$ always has a t_i labeled edge, so it is clear that $1 < k \leq l < i+2$ in step $i+1$.

boundary path

active point

endpoint

We call $\overline{s_k}$ the *active point* and $\overline{s_l}$ the *endpoint*. s_k is the active suffix of $t_1 \dots t_i$ (the longest nested one). *ast* inserts two different kinds of t_i-edges: Until the active point these are leaves that extend a branch, after that each edge starts a new branch. When applying this to the construction of compact suffix trees one Ukkonen's key idea was to introduce open edges as described on page 6. This way the edges would expand automatically with the string and needed not be added by hand.

Now we just have to describe how to update the active point (that is initially *root*) and how to add the branching nodes and new leaves from there along the boundary path to the endpoint. Let $\mathbb{T}^i$ be the compact suffix tree after step i

Lemma 3. *If reference pair $(s,(k,i-1))$ is the endpoint of $\mathbb{T}^{i-1}$ in step i, then $(s,(k,i))$ is the new active point of $\mathbb{T}^i$ in step $i+1$.*

Proof. Let $\overline{s_j} = (s,(k,i))$. $\overline{s_j}$ is the active point in $\mathbb{T}^i$, if s_j is the longest nested suffix of suf_i, if s_j is the longest factor of suf_{i-1}, such that $s_j \sqsupset_s suf_i$, if there is a t_i-edge from $\overline{s_j}$ in tree $\mathbb{T}^{i-1}$ and j is minimal, if $\overline{s_j}$ is the endpoint of $\mathbb{T}^{i-1}$. □

Since the current active point (and other later reached locations) can be implicit, *ukk* works with (canonized) reference pairs and inserts nodes to make locations explicit as necessary. When following the boundary path *ukk* follows the suffix links of the node of the current reference pair and then canonizes the new reference pair.

For each step the procedure *update* is called, that inserts all new branches for the new given character.

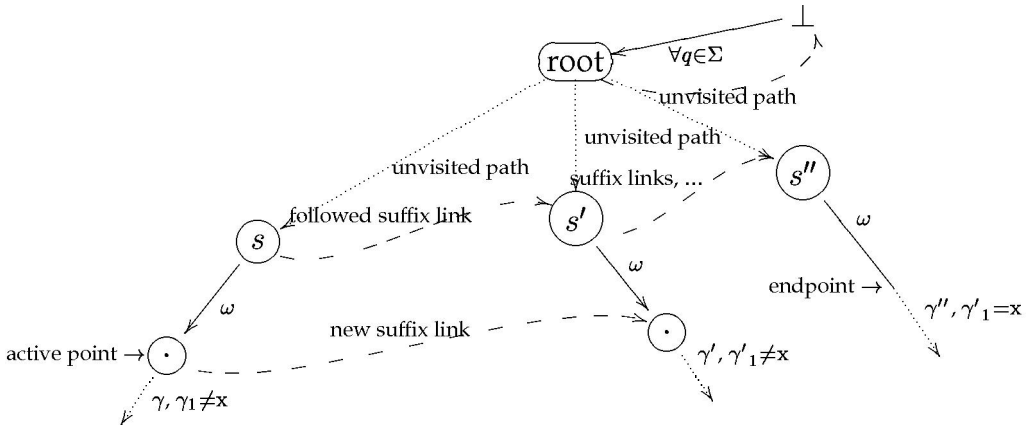


Figure 5: An Iteration of *ukk* with New Letter x and Initial Active Point (s, ω)

Procedure *update*$(s, (k, i))$
s is a node, $(s, (k, i))$ the reference pair for the current active point.

```
1:  oldr ⟵ root;
2:  (b_endpoint, r) ⟵ test_and_split(s, (k, i − 1), t_i);
3:  while ¬b_endpoint do
4:     add edge r --(i,∞)--> m with new node m;
5:     if oldr ≠ root then
6:        suffix_link(oldr) ⟵ r;
7:     end if
8:     oldr ⟵ r;
9:     (s, k) ⟵ canonize(suffix_link(s), (k, i − 1));
10:    (b_endpoint, r) ⟵ test_and_split(s, (k, i − 1), t_i);
11: end while
12: if oldr ≠ root then
13:    suffix_link(oldr) ⟵ s;
14: end if
15: return (s, k);
```

test_and_split tests whether the given (implicit or explicit) location is the endpoint and returns that in the first output parameter. If it is not the endpoint, then a node is inserted (and thus the location made explicit, if not already). The node is returned as second output parameter.

Procedure *test_and_split*$(s, (k,p), \mathrm{x})$
s is a node, $(s,(k,p))$ the canonical reference pair that is to be tested whether it is the endpoint, x is the letter that is tested as edge label.

1: **if** $|t_k \ldots t_p| > 0$ **then**
2: let $\omega = (k', p')$ be the edge label of edge $e = s \xrightarrow{\omega} s'$ with $\omega_1 = t_k$;
3: **if** $\mathrm{x} = \omega_{|t_k \ldots t_p|}$ **then**
4: return (**true**, s);
5: **else**
6: split e with new node m, and edges $s \xrightarrow{\omega_{k'} \ldots \omega_{k'+|t_k \ldots t_p|-1}} m$ and $m \xrightarrow{\omega_{k'+|t_k \ldots t_p|} \ldots \omega_{p'}} s'$;
7: return (**false**, m);
8: **end if**
9: **else**
10: **if** $\exists s \xrightarrow{\mathrm{x}\ldots} m$ **then**
11: return (**true**, s);
12: **else**
13: return (**false**, s);
14: **end if**
15: **end if**

test_and_split expects a canonical reference pair in order to work. It should be obvious that it takes constant time. To produce a canonical reference pair from a reference pair, we have the procedure *canonize*. It returns only a different node and a different start index for the edge label, since the end index must stay the same.

Procedure *canonize*$(s, (k,p))$
s is a node, $(s,(k,p))$ a reference pair that is to canonized.

1: **if** $|t_k \ldots t_p| = 0$ **then**
2: return (s, k);
3: **else**
4: find edge $e = s \xrightarrow{\omega} s'$ with $\omega_1 = t_k$;
5: **while** $|\omega| \leq |t_k \ldots t_p|$ **do**
6: $k \longleftarrow k + |\omega|$;
7: $s \longleftarrow s'$;
8: **if** $|t_k \ldots t_p| > 0$ **then**
9: find edge $e = s \xrightarrow{\omega} s'$ with $\omega_1 = t_k$;
10: **end if**
11: **end while**
12: return (s, k);
13: **end if**

The complete algorithm can now be written as follows:

Algorithm *ukk*
t is the given string.
1: $T \longleftarrow$ *emtpy_tree;*
2: add *root* and $\perp$ to T.
3: **for all** x $\in \Sigma$ **do**
4: add edge $\perp \xrightarrow{x} root$
5: **end for**
6: `suffix_link`$(root) \longleftarrow \perp$;
7: $n \longleftarrow length(t)$;
8: $s \longleftarrow root$;
9: $k \longleftarrow 1$;
10: **for** $i \longleftarrow 1$ to n **do**
11: $(s,k) \longleftarrow canonize(s,(k,i-1))$;
12: $(s,k) \longleftarrow update(s,(k,i))$;
13: **end for**

As said before, both *ukk* and *ast* are on-line. This property is reached if we replace the for loop in line 10 of the algorithms by a loop, that stops when no new input is given or an ending symbol is found.

4.3 Complexity of *ukk*

Proposition 7. *ukk takes time* $O(n)$.

Proof. We will analyze the time taken in procedure *canonize* and in the rest independently.
For each step of the algorithm *update* is called once. It has constant running time except for the execution of the while-loop (and *canonize* that will be dealt with later). Let $\overline{res_i}$ be the active point after step i. In the while-loop *ukk* moves along the boundary path from the active point of the current step to the new active point (minus one letter) as proved in lemma 3. Let $\omega = res_i$. For each move in the while-loop, a suffix link is taken and ω is decreased by one. At the end of step i (or the beginning of step $i+1$) ω is increased by the next letter to form res_{i+1}. Hence the number of steps in the while-loop is $|res_i| - |res_{i+1}| + 1$. This gives a complexity of $n + \sum_{i=1}^{n} |res_i| - |res_{i+1}| + 1 = 2n + |res_0| - |res_n| \leq 2n$. This is also the number of calls to *canonize*, so we will now only deal with the while-loop (lines 5 to 8) of *canonize*.
canonize is always called on string $t_k \ldots t_{i-1}$. The string is initially empty and enlarged in every iteration of *ukk*. Each iteration of the while-loop in *canonize* shortens the string at least by one character. Therefore the total number of iterations can only be $O(n)$.
Together all parts of the algorithm take time $O(n)$. □

4.4 Relationship between *ukk* and *mcc*

[GK97] shows that *mcc* and *ukk* actually perform the same abstract operations of splitting edges and adding leaves in the same order. They differ only in the control

structure and *ukk* can be transformed into *mcc*. I will not further describe this, but to make it plausible, one can notice that the first inserted leaf by *ukk* is also the longest suffix (it will become the longest suffix, when the string has grown to its final size), and that *ukk* follows suffix links in the same manner as *mcc* when inserting branching nodes.
[GK97] also state that this is an optimization of *ukk*'s control structure, which is also plausible when taking a look at the compact suffix tree *cst*(`ababc`). While *mcc* inserts a branch in every iteration, *ukk* doesn't: In steps 3 and 4, no branches are inserted, only the leaves enlarge automatically. Compare figures 9, 10, 11, 12, and 13 in appendix B to figures 14, 15, 16, 17, and 18.