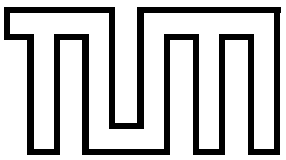
Suffix Trees and their Applications

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 **3**

2 Data Structures **4**
2.1 Notation 4
2.2 Σ^+-Trees and suffix trees. 4
2.3 Suffix Links and Duality of Suffix Trees 7
2.4 Space-Requirements of Suffix Trees 8

3 McCreight's Algorithm **9**
3.1 The Underlying Idea 9
3.2 The Algorithm 10
3.3 Complexity of *mcc* 11

4 Ukkonen's Algorithm **13**
4.1 On-Line Construction of Atomic Suffix Trees 13
4.2 From *ast* to *ukk*: On-Line Construction of Compact Suffix Trees 14
4.3 Complexity of *ukk* 17
4.4 Relationship between *ukk* and *mcc* 17

5 Problems in the Implementation of Suffix Trees **19**
5.1 Taking the Alphabet Size into Account 19
5.2 More Precise Storage Considerations 19

6 Suffix Trees and the Assembly of Strings **20**
6.1 The Superstring Problem 20
6.2 A GREEDY-Heuristic with Suffix Trees 20

7 Suffix Trees and Data Compression **23**
7.1 Simple Data Compression 23
7.2 Extensions 25

8 Other Applications and Alternatives **26**
8.1 Other Applications 26
8.2 Alternatives 26

9 Conclusion and Prospects **27**

A Data Compression Figures **28**

B Suffix Tree Construction Steps **31**

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 $\mathtt{a}, \mathtt{b}, \ldots$.
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 $w \sqsupset\sqsubset_f t$.

right-branching

left-branching

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

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 $t \xrightarrow{v} s$.*

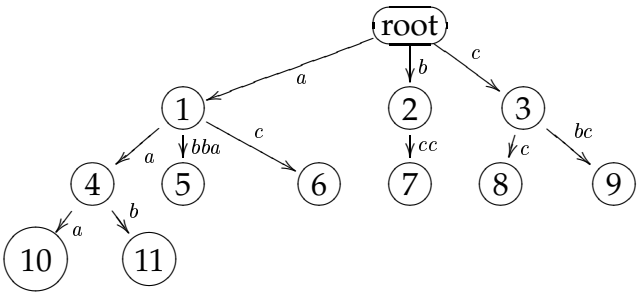


Figure 1: Example of a Σ^+-tree

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

location

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 $path(\overline{w}) = w$.
Since every branch is unique, if $path(t) = w$ we can denote a node t $\overline{w}$. In the

subtree

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 $(\overline{\varepsilon}, \texttt{ab})$ is a reference pair for ab and $\widehat{\texttt{ab}} = (1, b)$ 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

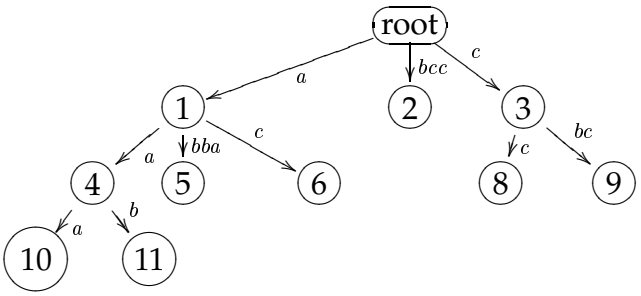


Figure 2: The compact Σ^+-tree for figure 1

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

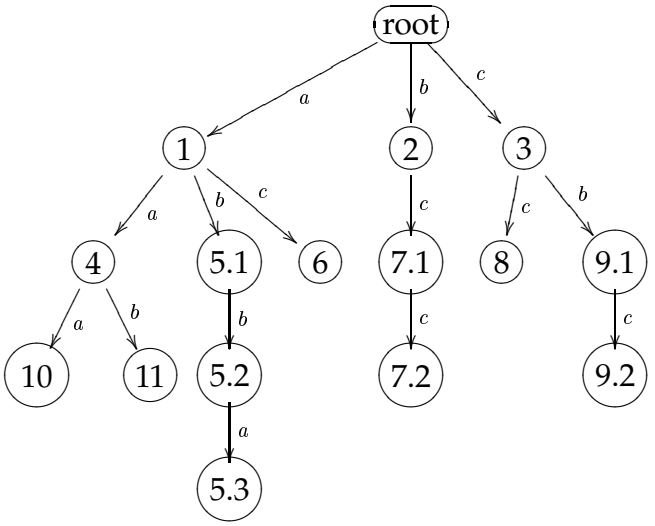


Figure 3: The atomic Σ^+-tree for figure 1

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 active 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.

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.

suffix link

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.*

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 $\overline{\mathrm{a}w}$, $\overline{w}$ is also a node in $cst(t)$. If $\overline{\mathrm{a}w}$ 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 $\overline{\mathrm{a}w}$ is a branching node, then $\mathrm{a}w$ 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 $\overline{u\mathrm{a}w}$ to $\overline{w}$ and from $\overline{v\mathrm{a}w}$ to $\overline{w}$. Here $u \neq \varepsilon$ and $v \neq \varepsilon$ because if $\overline{\mathrm{a}w}$ is explicit, the suffix links would point there instead of to $\overline{w}$.
If $\overline{u\mathrm{a}w}$ or $\overline{v\mathrm{a}w}$ are inner nodes, then $u\mathrm{a}w$ or $v\mathrm{a}w$ is right branching in t. But then $\mathrm{a}w$ must also be right branching and be an explicit location, which is a contradiction.
If $\overline{u\mathrm{a}w}$ and $\overline{v\mathrm{a}w}$ are leaves, $u\mathrm{a}w$ and $v\mathrm{a}w$ must be suffixes of t and so $u\mathrm{a}w \sqsupset_s v\mathrm{a}w$ or $v\mathrm{a}w \sqsupset_s u\mathrm{a}w$ in which case the suffix link from $\overline{v\mathrm{a}w}$ must point to $\overline{u\mathrm{a}w}$ 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:
$\overline{w^{-1}} \xrightarrow{\mathrm{a}} \overline{(\mathrm{a}w)^{-1}}$ is an edge in $(ast(t))^{-1}$, iff $\overline{\mathrm{a}w} \xrightarrow{\mathrm{a}} \overline{w}$ is a suffix link in $ast(t)$, iff there are nodes $\overline{\mathrm{a}w}$ and $\overline{w}$ in $ast(t)$, iff there are nodes $\overline{(\mathrm{a}w)^{-1}}$ and $\overline{w^{-1}}$ in $ast(t^{-1})$, iff there is an edge $\overline{w^{-1}} \xrightarrow{\mathrm{a}} \overline{(\mathrm{a}w)^{-1}}$ 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* $head_{i-1} = \mathrm{a}w$ *for letter* a *and (possibly empty) string* w*, then* $w \sqsubset_p head_i$*.*

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

We know the location of $head_{i-1} = \mathrm{a}w$ 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 $\overline{head_{i-1}}$.

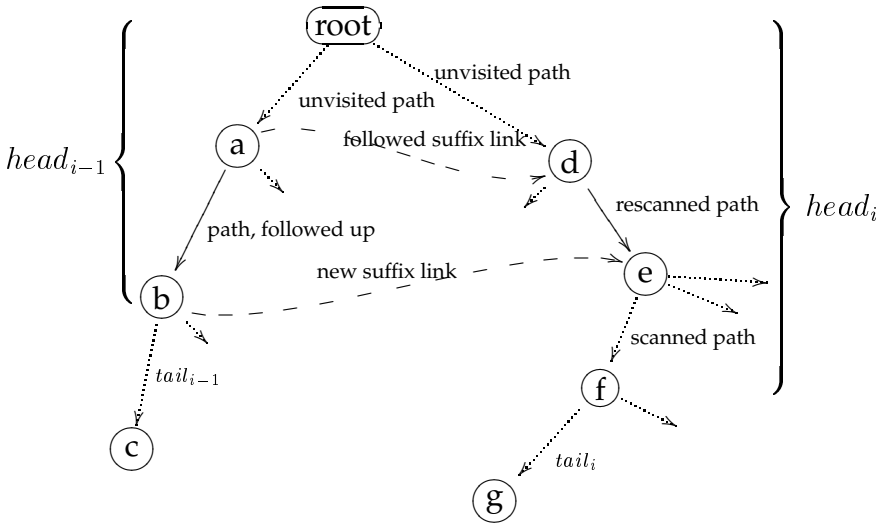


Figure 4: Suffix Link Usage by *mcc*

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 $Rescan(n, \beta)$
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 $Scan(n, \gamma)$
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

boundary path

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 $i+1$.

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.

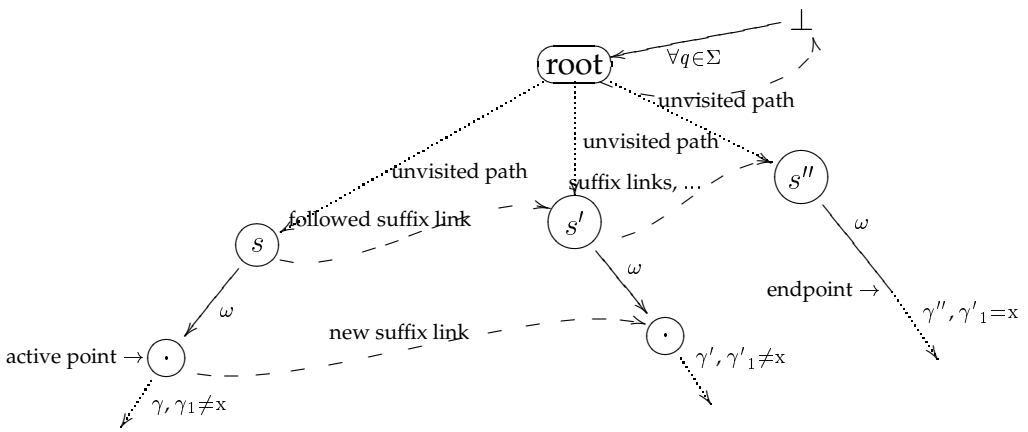


Figure 5: An Iteration of *ukk* with New Letter x and Initial Active Point (s, ω)

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.

if $|t_k \dots t_p| > 0$ **then**
  let $\omega = (k', p')$ be the edge label of edge $e = s \xrightarrow{\omega} s'$ with $\omega_1 = t_k$;
  if $\mathrm{x} = \omega_{|t_k \dots t_p|}$ **then**
    return $(\mathbf{true}, s)$;
  else
    split e with new node m, and edges $s \xrightarrow{\omega_{k'} \dots \omega_{k'+|t_k \dots t_p|-1}} m$ and $m \xrightarrow{\omega_{k'+|t_k \dots t_p|} \dots \omega_{p'}} s'$;
    return $(\mathbf{false}, m)$;
  end if
else
  if $\exists s \xrightarrow{\mathrm{x} \dots} m$ **then**
    return $(\mathbf{true}, s)$;
  else
    return $(\mathbf{false}, s)$;
  end if
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.

if $|t_k \dots t_p| = 0$ **then**
  return (s, k);
else
  find edge $e = s \xrightarrow{\omega} s'$ with $\omega_1 = t_k$;
  while $|\omega| \leq |t_k \dots t_p|$ **do**
    $k \longleftarrow k + |\omega|$;
    $s \longleftarrow s'$;
    if $|t_k \dots t_p| > 0$ **then**
      find edge $e = s \xrightarrow{\omega} s'$ with $\omega_1 = t_k$;
    end if
  end while
  return (s, k);
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 $t_k \dots t_{i-1}$. 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 $cst(\texttt{ababc})$. 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.

5 Problems in the Implementation of Suffix Trees

5.1 Taking the Alphabet Size into Account

As mentioned in 2.1 we have assumed until now that the alphabet is constant. When the size of the alphabet is an issue (for most practical applications), we are faced with another problem. To be able to find an outgoing edge of an inner node in constant time, we would need an array the size of the alphabet stored with each node that contained a pointer to a child for each x $\in \Sigma$. This would blow up the data structure size to $O(n|\Sigma|)$. If we want to keep the size small and use dynamic lists or balanced search trees at the nodes, we end up with complexity $O(n|\Sigma|)$ or $O(n \log |\Sigma|)$, respectively. (Consider the example $cst($`abcdefghijklmnopqrstuvxyz`$)$.)
A practical solution is to use hash tables. We use a global hash table that takes a node number and a character for a key. It returns the number of the node the edge starting with the given character from the given node leads to. Since the number of edges is limited to the number of nodes, we can assume an upper bound of $2n$ and size the hash table accordingly. When using chaining (and a uniform hash function), the average search time is $O(1 + \alpha)$, where α is the load factor of the table [CLR90]. When dimensioning the table after the size of the input string, α can be kept small enough (e.g. $m = n \Rightarrow \alpha \leq 1$).
For Ukkonen's on-line algorithm this does not work if the string length is not known before hand. But as long as the size of the table is proportional to the input size, we still have constant access on average.

5.2 More Precise Storage Considerations

We have proven in section 2.4 that a compact suffix tree takes $O(n)$ space. But we are now interested in the factor before the n ($= |t|$). Assume, we have $N = 2n$ nodes. Then for each node we have one edge to its parent, with a label of two indices pointing into our string t. We can store the edge labels with the nodes the edge points to. Using the hash table of size N, we store at each node a pointer to its parent, a suffix link and two indices. Then we also have the hash table of size N with N entries each consisting of two node indices and a character, and we must store the string t. If we use integers for indices and assume the same size for pointers as for integers, we have a total size of $4N + 4N = 16n$ integers and $2n$ characters (the string and the hash table keys).
Applying this to an n byte text on a machine with 2 byte integers our suffix tree has the final size of $34n$ bytes. This is of course a worst case upper bound. Empirical results from [MM93] show that the actual factor lies at about 20 (although they don't use the hash table set-up).

6 Suffix Trees and the Assembly of Strings

6.1 The Superstring Problem

super-string problem

One application of suffix trees is the assembly of strings, known as the *superstring problem*. This problem frequently occurs in the analysis of DNA strings, where a lot of small sequences of the whole string have been analyzed and read, and they need to be put together to form the complete sequence. According to [KD95], "there is a need to handle tens of thousands of strings whose total length is on the order of tens of millions."

Definition 5. *(The superstring problem) Given a set of strings* $\{s_1, s_2, \ldots, s_m\}$ *find a superstring* S *with* $\forall i : s_i \sqsubset\!\sqsubset_f S$.

[KD95] state that the superstring problem is MAX SNP-hard and GREEDY-heuristics achieve an approximation factor of 4.
The problem is solved by finding a prefix of s_i and a suffix of s_j, such that $\omega \sqsubset_p s_i$ and $\omega \sqsupset_s s_j, i \neq j$. We then can assemble s_i and s_j to a new string $s_{i,j}$ with length $|s_{i,j}| = |s_i| + |s_j| - |\omega|$. s_i and s_j are removed from the set and replaced by $s_{i,j}$. We call this operation *blending* s_i and s_j, where s_i will then be a prefix of the new string. We let $ov(s_i, s_j) = \omega$. The blending is done until the set contains only one string, which is the superstring $S = s_{1,\ldots,m}$.

blending

Unfortunately, local optimality does not imply global optimality, so that GREEDY-heuristics can only approximate the problem of finding a smallest S.

6.2 A GREEDY-Heuristic with Suffix Trees

I will present an algorithm from [KD95]. To solve the problem with suffix trees we need to modify the algorithm and extend the data structure. Let $n = |s_1| + \ldots + |s_m|$. The algorithm needs not build a suffix tree for each string but one that contains all strings. This can be done with a simple modification. We can build a suffix tree of the string $s_1\$_1 s_2\$_2 \ldots s_m\$_m$, where $\$_i$ are different sentinel characters. When inserting leaves, we omit everything from the next sentinel character on. The sentinel character is also omitted so that we might end up with an internal node as a "leaf". (With *mcc* it is easy to insert shorter leaves when $tail_i > 1$, and to just insert and label an internal node when $tail_i = 1$.) Each node is then labeled with the string and suffix number $(i, p), 1 \leq i \leq m, 1 \leq p \leq |s_i|$, if it represents the location p-th suffix of the i-th string.
We actually don't need any strings, that are factors of other strings, so that we can omit the insertion of such strings. (This can also be done nicely with *mcc*: the longest suffix, which is equal to the string, is inserted first. When we find that $tail_i = 1$, where we would just split the edge and have an internal node as "leaf", we stop and jump to the next string. This way all factors of other strings aren't even inserted.)
Each node u is associated with two sets P_u and S_u. P_u are the prefixes available at node u, S_u the suffixes:
$P_u = \{i | (i, 1)$ is a label at $u\}$ and $S_u = \{i | (i, d)$ is a label at u with $d > 1\}$.

We also keep track of the string depth $depth(u)$ of each node u (the length of the concatenated edge labels from $root$ to u). This is also easily incorporated into *mcc*. And we keep an array $nodesdepths$ of the nodes with a given depth. Slot i contains a set of nodes with depth i. The maximal depth can be found in one scan of all strings at the beginning of the algorithm (and thus the array's size).
We add three arrays $chain$, $chaindepth$, and $wrap$ to our data structure that keep track of eliminated and blended strings. Suppose strings $s_{i_1}, s_{i_2}, \ldots, s_{i_k}$ are blended to $s_{i_1,i_2,\ldots,i_k}$ in that order, then $chain[i_j] = i_{j+1}, \forall j = 1, \ldots, k-1$, $chain[i_k] = 0$, $wrap[i_1] = i_k$, $wrap[i_k] = i_1$, and $chaindepth[i_j] = |ov(s_{i_j}, s_{i_{j+1}})|, \forall j = 1, \ldots, k-1$. The size of each array is m.

Lemma 4. *Initialization of the data structures takes $O(n)$.*

Proof. The suffix tree can be built easily with the modified *mcc* in $O(n)$. The initialization of the sets can be done on insertion of the nodes. The array initialization takes time $O(m) = O(n)$. □

To assemble the strings we repeatedly search pairs of strings s_i, s_j with $|ov(s_i, s_j)|$ maximal and blend them together until only one string is left. Observe that $|ov(s_i, s_j)|$ corresponds to the $depth(u)$, where $i \in S_u$ and $j \in P_u$.
The whole algorithm looks as follows (assuming that no string is a factor of another):

Algorithm *gsa*

$\{s_1, s_2, \ldots, s_m\}$ is the given string set.

```
 1: initialize the data structures;
 2: str_num ⟵ m;
 3: curr_depth ⟵ sizeof(nodedepths);
 4: while str_num > 1 do
 5:    while nodedepths[curr_depth] = ∅ do
 6:       curr_depth ⟵ curr_depth − 1
 7:    end while
 8:    let u ∈ nodedepths[curr_depth]
 9:    find i ∈ S_u and j ∈ P_u, such that chain[i] = 0 and wrap[i] ≠ j, discard all i
       from S_u where chain[i] ≠ 0;
10:    if a pair (i, j) is found then
11:       remove i from S_u and j from P_u;
12:       chain[i] ⟵ j;
13:       chaindepth[i] ⟵ curr_depth;
14:       wrap[wrap[i]] ⟵ wrap[j];
15:       wrap[wrap[j]] ⟵ wrap[i];
16:       str_num ⟵ str_num − 1;
17:    else
18:       let v be the parent of u;
19:       union P_u to P_v;
20:       discard u from nodedepths[curr_depth]
21:    end if
22: end while
23: generate the superstring S from chain and chaindepth;
```

The algorithm uses lazy evaluation in that blended strings are not removed from the corresponding suffix sets S_u when blended together, but rather when they are encountered (hence line 9). The P_u sets are always up-to-date since prefixes are only propagated upwards if they are not used at a greater depth.

Proposition 8. *The algorithm gsa (*GREEDY *String Assembling) takes time $O(n)$.*

Proof. The initialization takes time $O(n)$ as shown in lemma 4.
Each iteration of the while-loop lines 4–22 either discards a node or a string, so the total number of iterations is $O(n) + m = O(n)$.
The while-loop lines 5–7 executes exactly $\texttt{sizeof}(nodedepths) = O(n)$ times.
Everything else except for line 9 in the body of the outer while-loop takes constant time.
In line 9, if an i with $chain[i] = 0$ is found and a j from P_u doesn't match ($wrap[i] \neq j$), then we can pick another j' from P_u and we must have a match. If P_u contains only j, we pick another i' from S_u and we must have a match.
The number of elements discarded in line 9 from S_u for all u is bound by $O(n)$ (the number of suffixes in the suffix tree). □

In actual DNA string assembly there are two further complications. First DNA strings can match reverse (that is $(\texttt{ataatgc})^{-1} = \texttt{gcattat}$). We can solve this by adding the reverse of each string to the suffix tree, too, and modifying the algorithm. The second problem is harder. Experimental errors lead to faulty strings, so that we actually need to do approximate matching. [KD95] also presents a solution to this problem that will not be included here since it involves preprocessing the suffix tree for nearest common ancestor queries.

7 Suffix Trees and Data Compression

7.1 Simple Data Compression

Data compression always trades time against space. The goal is to compress a given text, so that its compressed representation is less then the original representation. There are many means of compression. I will here present one that solely relies on suffix trees without the use of any other methods like alphabet reduction (e.g. the 8th bit in pure ASCII texts) or any probabilistic analyses.

Like many methods the algorithm relies on the idea to represent sequences of characters that have already been seen with their earlier position in the text. If the sequences are large enough this saves some space. The algorithm relies on sufficiently large repeated sequences, so the results presented here are purely academic.

For real compression algorithms many more aspects need to be treated (see [BK98, Apo85, Szp93]), but a lot of them are also based on (at least) suffix tree like data structures. The key idea here is to keep the suffix tree of either the complete parsed text so far or a portion of it, and use it to search for repeated sequences of characters.

While the algorithm in section 6 uses *mcc*, this task is best performed by *ukk*, since it uses its on-line capability.

Algorithm *stc*

S is the character source and T the character target.

```
1: 𝕋 ⟵ empty_tree;
2: t ⟵ ε;
3: x = S.readChar();
4: while S not empty do
5:    u ⟵ ε;
6:    while S not empty and ux ∈ 𝕋 do
7:       u ⟵ ux;
8:       x = S.readChar();
9:    end while
10:   if |u| "is large enough" then
11:      if |u| ≠ 0 then
12:         writecharacters(T, t);
13:         𝕋.insert(t);
14:         t ⟵ ε;
15:      end if
16:      writepos(T, u, 𝕋);
17:   else
18:      t ⟵ tu;
19:      u ⟵ ε;
20:   end if
21: end while
22: if |u| ≠ 0 then
23:   writecharacters(T, t);
24:   t ⟵ ε;
25: end if
```

Procedures $writecharacters$ and $writepos$ write the given string or the position, where the string can be found earlier in the character sequence, to T. Here an integer is assumed 4 Byte, a character 1 byte.

Procedure $writecharacters(T, t)$

T is the target, t the string to be written.

```
1: while |t| > 127 do
2:    let u ⊏_p t with |u| = 127;
3:    T.writeByte(127);
4:    T.writeChars(u);
5:    t ⟶ t − u
6: end while
7: if |t| ≠ 0 then
8:    T.writeByte(|t|);
9:    T.writeChars(t);
10: end if
```

The sign bit of the first byte indicates whether we write a string or a position, the absolute value of the byte gives the number of characters.

Procedure $writepos(T, t, \mathbb{T})$
T is the target, t the string to be written, $\mathbb{T}$ the suffix tree.

```
1: p ⟵ 𝕋.start_position_of(t);
2: while |t| > 127 do
3:    T.writeByte(−127);
4:    T.writeInt(p);
5:    t ⟶ t_128 ... t_|t|;
6:    p ⟶ p + 127;
7: end while
8: if |t| ≠ 0 then
9:    T.writeByte(−|t|);
10:   T.writeInt(p);
11: end if
```

Experiments have shown that a compression only takes place, when the number of sequences written by positions is more than 45 % of the total number of sequences written.

Appendix A shows results for texts of sizes 5000 and 10000 letters build out of 1 to 30 different words with random sizes between 5 and 30. It shows how the algorithm compares to gzip and bzip, and that suffix trees on their own are not enough for good compression.

7.2 Extensions

When compressing large amounts of data the suffix tree might grow too large. In this case a reduction of the represented string might be wanted. Fortunately suffix trees can be reduced easily from left to right.

The way this is done, is to always keep track of the leaf that was inserted earliest (which represents the largest suffix in the tree), and keep suffix links for leaves. A leaf can than be removed from the tree and the next longest leaf can be reached via the suffix link.

We can always keep track of the last inserted leaf, which is also the next longest suffix, and insert the appropriate suffix link. *ukk* inserts suffix link leaves in order of total length (that follows immediately from 4.4). The only thing that can happen, is that a leaf has no suffix link. This happens exactly when there have been insertions into the suffix tree that didn't enlarge it by any nodes, else the suffix link for that leaf must be $root$. If there is no suffix link, the leaf must have been inserted last and it is easy to verify, that if we reach this state we must be left with a tree containing only one leaf, which can be adjusted to a new size easily.

With these modifications the suffix tree can always be kept under a given size, establishing a "window" in the input stream.

Further details on incremental editing of compact suffix trees can be found in [McC76], although they are probably of little use in practical applications.

8 Other Applications and Alternatives

8.1 Other Applications

There are "myriads" [Apo85] of applications for suffix trees. Besides the one already shown (fast answer to the question whether a string is contained in another, string assembly, and data compression) suffix trees can be used in many other applications.

two headed automaton

The atomic suffix tree $ast(\omega)$ can be viewed as a *two headed automaton* scanning a text for the words contained in the tree with one head at the beginning of a factor and one head at the end of the factor of $t = \alpha \mathrm{x} \beta \mathrm{y} \gamma$, where $\mathrm{x}\beta \sqsupset\!\sqsubset_f \omega$:

$$\alpha[\mathrm{x}\beta]\mathrm{y}\gamma \vdash \begin{cases} \alpha[\mathrm{x}\beta\mathrm{y}]\gamma & \text{if } \mathrm{x}\beta\mathrm{y} \sqsupset\!\sqsubset_f \omega, \\ \alpha\mathrm{x}[\beta]\mathrm{y}\gamma & \text{otherwise} \end{cases}$$

The suffix links can thus be interpreted as failure transitions. With some further modifications and additional data structures this can be easily used for approximate string matching (matching strings with a given edit distance k) in time $O(kn)$ and preprocessing time $O(m^2 + |\Sigma|)$ (see [UW93]).

first occurrence

With the generated suffix tree it is also easy to search for the *first occurrence* of a substring ω in $O(|\omega|)$. We just need to scan into the tree until the string is found and output the position that can be calculated from the edge labels. If the tree was built with a sentinel character, *all other occurrences* can be found by inspecting the subtree's leaves in $O(|\omega| + output)$ (output is the number of positions).

all other occurrences substring identifier

The *substring identifier* is the shortest prefix of a suffix of t that is needed to uniquely identify a substring. It can also be found easily because it is $head_i(tail_i)_1$ for suf_i.

number of occurrences longest repeated factor

The *number of occurrences* of a factor in t can also be easily found, by adding a weight field to each node, weighing each leaf with one and each inner node with the sum of weights of its children.

longest common substring

The *longest repeated factor* of string t can also be found easily. It is $max_i\{|head_i|\}$. Similarly the *longest common substring* of two strings u and v can be found from the suffix tree $cst(u\$_1 v\$_2)$.

Even more applications can be found in [Apo85].

8.2 Alternatives

For the string matching problem other algorithms are ready at hand [CLR90]. The Rabin-Karp-Algorithm can find a pattern in worst-case time $O((n - m + 1)m)$ and expected time can be as low as $O(n + m)$. Finite automata can be constructed that take search time $O(n)$ and preprocessing time $O(m|\Sigma|)$, which is quite similar to suffix trees. The Knuth-Morris-Pratt algorithm outperforms suffix trees, it needs only $O(n + m)$. For long patterns and large alphabets the Boyer-Moore algorithm performs quite well in practice (and the shift table can be calculated using suffix trees [Kur95]).

suffix arrays

If suffixes are needed to find instances of a substring ω in a larger text t (as for data compression), then *suffix arrays* as introduced in [MM93] might be a good alternatives, especially if the size of the text, the alphabet, and the resulting data structure

matter. Suffix arrays are basicly a sorted list of all suffixes with an auxiliary structure that gives the length of the longest common prefix of two adjacent suffixes. With this structure string searches can be answered in $O(m + \log n)$ time. In this search only few accesses to the text are needed ($O(m + \log n - 1)$). The data structure has size $O(n)$ with a constant factor of about 6, and the construction takes worst-case time $O(n \log n)$ and expected time $O(n)$. The time and space used is independent of $|\Sigma|$, which avoids all problems raised in section 5.1. Experiments show that in practice suffix trees can be constructed much faster, but the search time often exceeds that of suffix array. Especially the size needed is significantly larger for suffix trees. Details can be found in [MM93].

9 Conclusion and Prospects

Since the first introduction of a linear time construction algorithm, suffix trees have been studied extensively. Unfortunately the algorithms by Weiner and McCreight are rather complicated, which have inhibited a wider use of the data structures.
Applications are found enough for suffix trees and more may come up. With the easy, intuitive, and much more flexible algorithm of Ukkonen suffix trees might as well get rid of their "overly-complicated"-image and find their way into more text books about algorithms [GK97].
Affix trees combine the new view on the data structure and have not been studied extensively, so there is still room for further research.

A Data Compression Figures

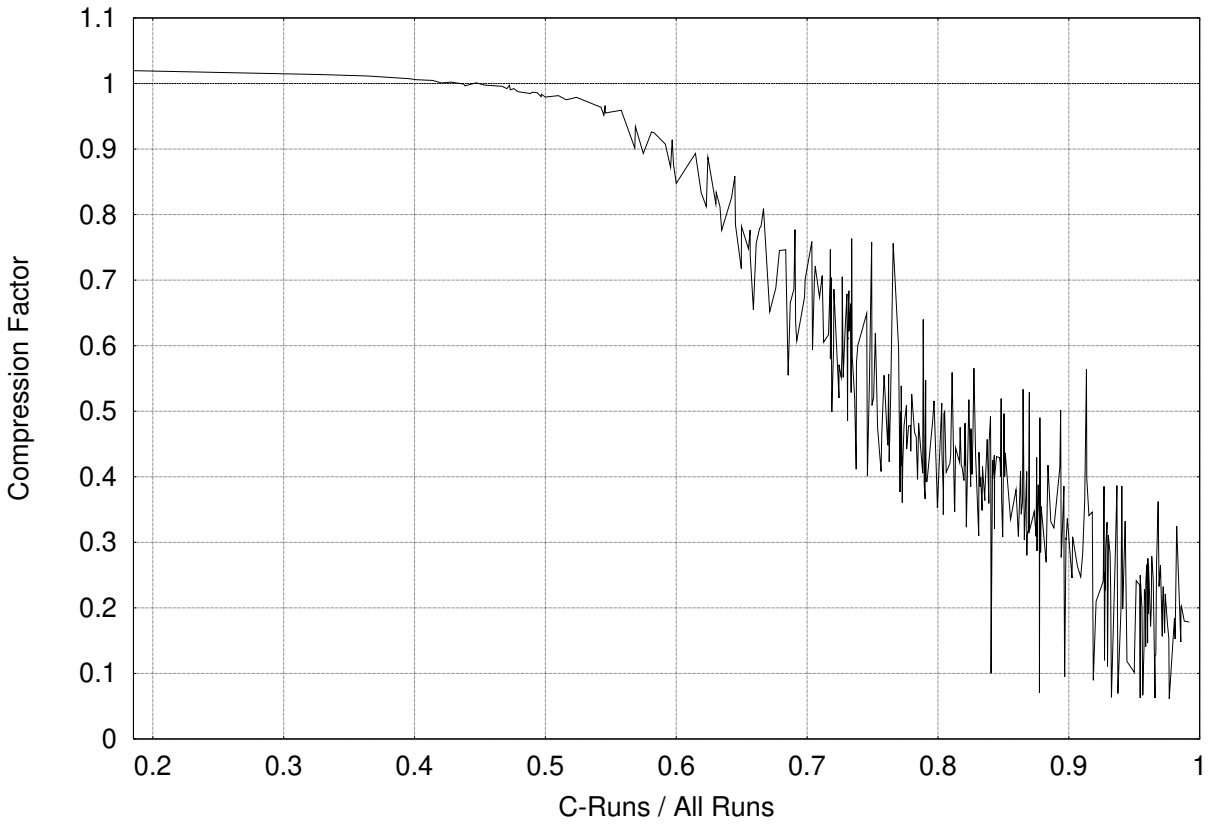


Figure 6: Compression Rate in relation to the percantage of compressed runs in a suffix tree compression

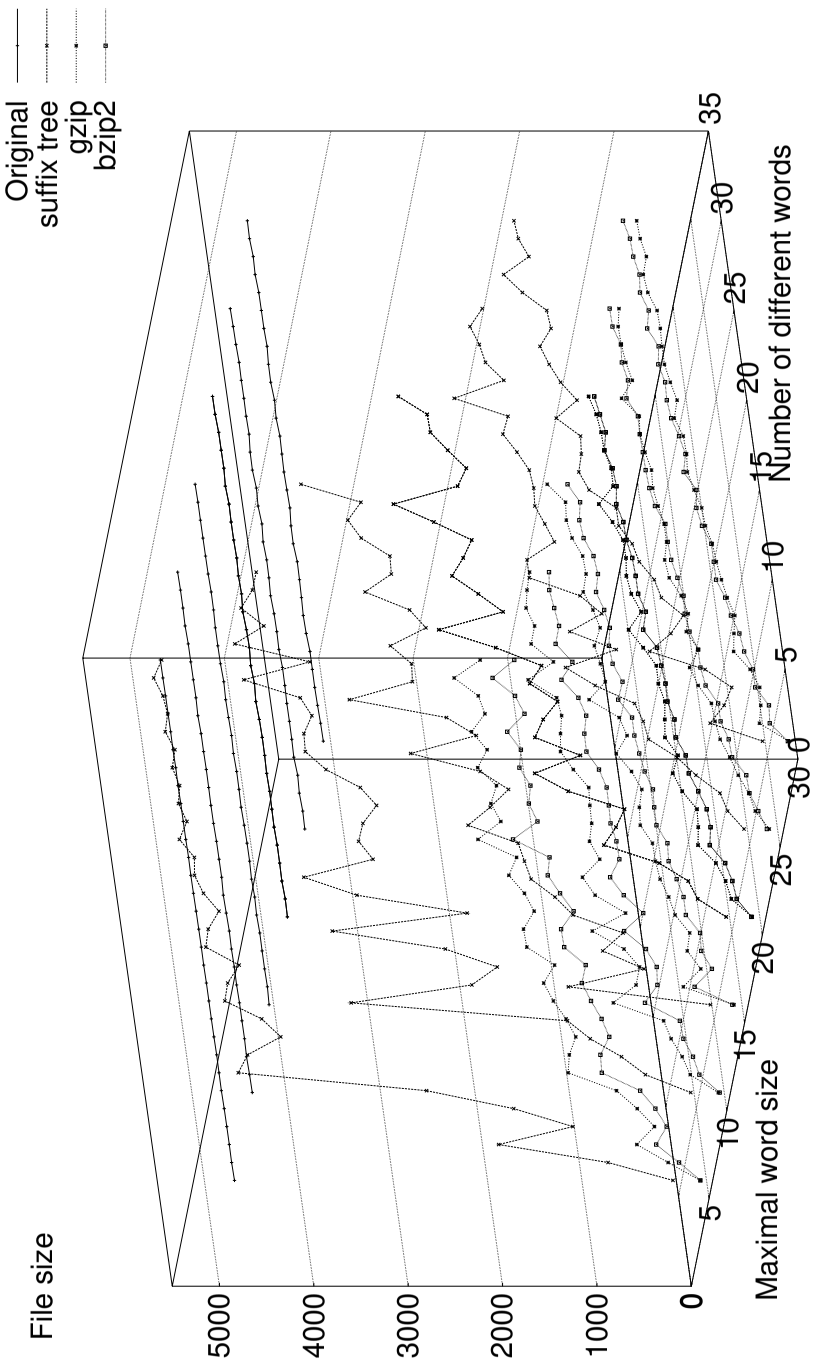


Figure 7: Compression of a text with length 5000 build out of different random word sizes and numbers

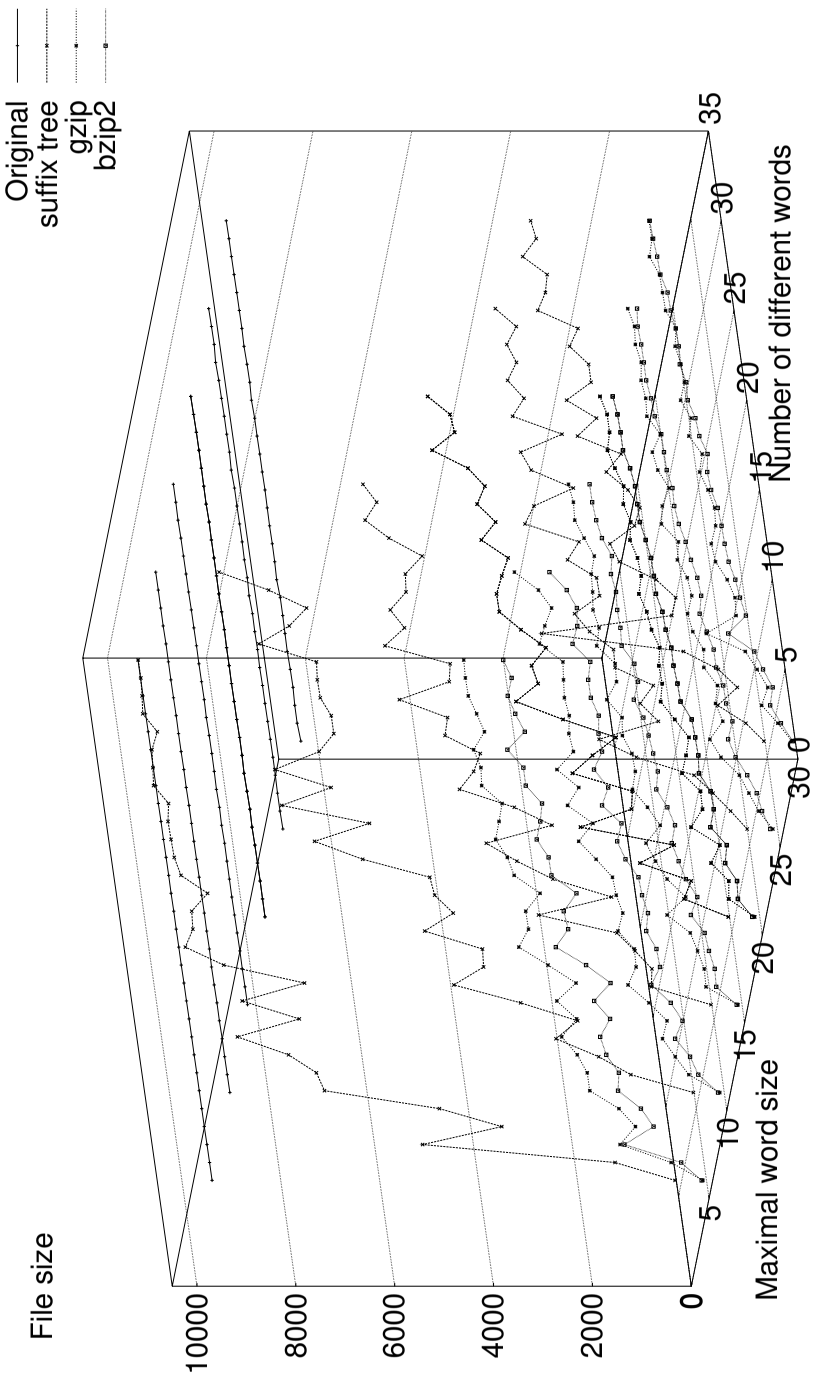


Figure 8: Compression of a text with length 10000 build out of different random word sizes and numbers

B Suffix Tree Construction Steps

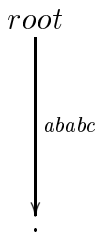


Figure 9: Step 1 of the *mcc* in the Construction of the Suffix Tree for `ababc`

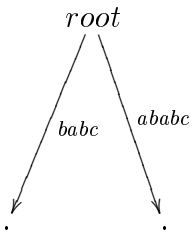


Figure 10: Step 2 of the *mcc* in the Construction of the Suffix Tree for `ababc`

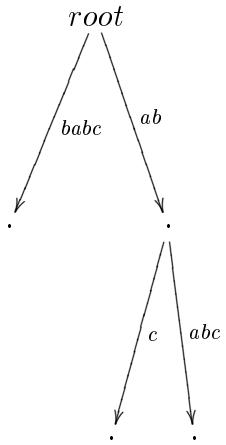


Figure 11: Step 3 of the *mcc* in the Construction of the Suffix Tree for `ababc`

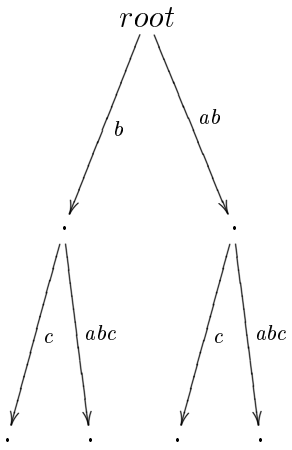


Figure 12: Step 4 of the *mcc* in the Construction of the Suffix Tree for `ababc`

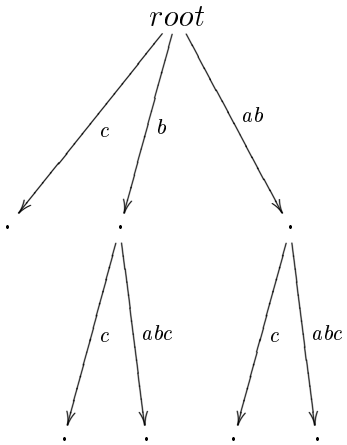


Figure 13: Step 5 of the *mcc* in the Construction of the Suffix Tree for `ababc`

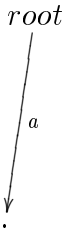


Figure 14: Step 1 of the *ukk* in the Construction of the Suffix Tree for `ababc`

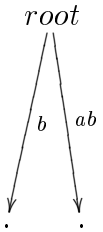


Figure 15: Step 2 of the *ukk* in the Construction of the Suffix Tree for `ababc`

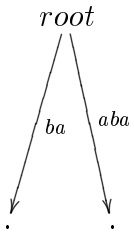


Figure 16: Step 3 of the *ukk* in the Construction of the Suffix Tree for `ababc`

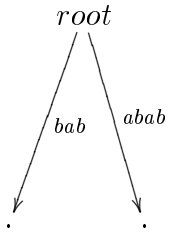


Figure 17: Step 4 of the *ukk* in the Construction of the Suffix Tree for `ababc`

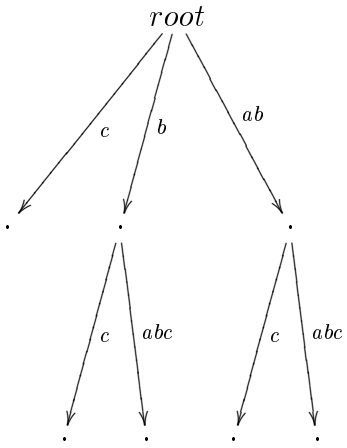


Figure 18: Step 5 of the *ukk* in the Construction of the Suffix Tree for `ababc`

References

[Apo85] Alberto Apostolico. The Myriad Virtues of Subword Trees. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume F12 of *NATO ASI*, pages 85–96. Springer, 1985.

[BK98] Bernhard Balkenhol and Stefan Kurtz. Universal Data Compression Based on the Burrows and Wheeler-Transformation: Theory and Practice. From the Internet, Universität Bielefeld, sep 1998.

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

[GK97] R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction. *Algorithmica*, 19:331–353, 1997.

[KD95] S. Rao Kosaraju and Arthur L. Delcher. Large-Scale Assembly of DNA Strings and Space-Efficient Construction of Suffix Trees. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing*, pages 169–177. ACM, 1995.

[Kur95] S. Kurtz. *Fundamental Algorithms for a Declarative Pattern Matching System*. PhD thesis, Universität Bielefeld, jul 1995.

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

[MM93] Udi Manber and Gene Myers. Suffix Arrays: A New Method for On-line String Searches. *SIAM J. COMPUT.*, 22(5):935–948, oct 1993.

[Sto95] J. Stoye. Affixbäume. Master's thesis, Universität Bielefeld, may 1995.

[Szp93] Wojciech Szpankowski. Asymptotic Properties of Data Compression and Suffix Trees. *IEEE Transactions on Information Theory*, 39(5):1647–1659, sep 1993.

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

[UW93] Esko Ukkonen and Derick Wood. Approximate String Matching with Suffix Automata. *Algorithmica*, 10:353–364, 1993.

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