A Comparison of Imperative and Purely Functional Suffix Tree Constructions*

Robert Giegerich *Stefan Kurtz*

Universität Bielefeld, Technische Fakultät
Postfach 100 131, 33501 Bielefeld, Germany
e-mail: {robert, kurtz}@techfak.uni-bielefeld.de

Abstract

We explore the design space of implementing suffix tree algorithms in the functional paradigm. We review the linear time and space algorithms of McCreight and Ukkonen. Based on a new terminology of nested suffixes and nested prefixes, we give a simpler and more declarative explanation of these algorithms than was previously known. We design two "naive" versions of these algorithms which are not linear time, but use simpler data structures, and can be implemented in a purely functional style. Furthermore, we present a new, "lazy" suffix tree construction which is even simpler. We evaluate both imperative and functional implementations of these algorithms. Our results show that the naive algorithms perform very favourably, and in particular, the lazy construction compares very well to all the others.

1 Introduction

Suffix trees are the method of choice when a large sequence of symbols, the "text", is to be searched frequently for occurrences of short sequences, the "patterns". Given that the text t is known and does not change (think of a famous novel or genetic data), while the patterns are not known in advance, one has to invest a certain effort to construct t's representation as a suffix tree. Given this suffix tree, all occurrences of a pattern p can be located in $\mathcal{O}(|p|)$ steps, independent of the length of t. This efficient access to all subwords of t has made suffix trees a ubiquitous data structure in a "myriad" of applications [Apo85].

Since suffix tree construction is the price to be pre-paid, it is fortunate that suffix trees can be built in $\mathcal{O}(n)$ time and represented in $\mathcal{O}(n)$ space, where n is the length of t. Suffix tree construction algorithms have a long history, starting with [Wei73]. The construction in that paper is given in a somewhat obscure terminology. Later authors [McC76, MR80, CS85] have developed more transparent constructions, sometimes tailored to specific additional requirements. The endpoint of the development is currently marked by [Ukk93], presenting a simpler construction in $\mathcal{O}(n)$ space and time, which additionally is online. It processes the text from left to right, and hence, in this sense it is incremental. What more can one ask for?

*A part of this work, concentrating on the functional implementations, has occurred as [GK94b].

Our interest in suffix trees is motivated by our work on a flexible pattern-matching system for biosequence analysis [Gie92]. Besides for locating subwords, suffix trees are useful for finding repetitions and palindromes, deriving q-gram profiles [Ukk92a], and calculating the so-called matching statistics as a prerequisite for fast approximate matching [CL90]. Further typical problems are searching the text in reverse, searching its genetic complement, or searching in an abstraction of the original text (such as the purine/pyrimidine abstraction of the nucleic acid alphabet, or the hydrophobicity abstraction of the amino acid alphabet).

Our system design follows a language-oriented rather than a tool-box approach. The user is provided a declarative language for describing pattern-matching problems. Sophisticated algorithms for "standard" problems are embedded in this language; suffix tree construction is one of these. Building on this machinery, complex matching problems are solved via backtracking. This context leads to the following requirements: Our suffix tree construction should be embedded in a declarative language, polymorphic with respect to the underlying alphabet, and extensible with respect to application-specific annotation. The tree implementation should be as simple as possible, since this data structure will be visible to the user. A final desirable feature is incrementality, which has different and competing aspects. One aspect is incrementality with respect to the input text, i.e. online construction. The other aspect is that the suffix tree itself should be constructed incrementally as it is traversed, leaving incomplete those subtrees that are never actually needed, i.e. "lazy" construction.

Heretofore, suffix tree constructions have always been given in an imperative style. The best known algorithms heavily depend on local updates to the tree data structure, and hence violate the principle of statelessness. In this paper, we

- present a new, "lazy" construction for suffix trees, probably the simplest construction that has ever been given,
- review Ukkonen's and McCreight's $\mathcal{O}(n)$-time suffix tree constructions[1] and derive simpler, but less efficient versions that can be implemented in a purely functional way,
- evaluate their efficiency in imperative and functional implementations,
- conclude with some methodological observations about the benefits of studying the same algorithm in both the functional and the imperative paradigms.

2 Basic Notions

Let $\mathcal{A}$ be a finite set, the alphabet. The elements of $\mathcal{A}$ are letters. ε denotes the empty string, $\mathcal{A}^*$ denotes the set of strings over $\mathcal{A}$, and $\mathcal{A}^+ = \mathcal{A}^* \setminus \{\varepsilon\}$. We use a, c, d to denote letters, and $b, p, s, t, u, v, w, x, y, z$ to denote strings. When $t = uvw$ for some (possibly empty) u, v, w, then u is a prefix of t, v is a t-word and w is a suffix of t. We call a t-word v branching, if there are different letters a and c, such that va and vc are t-words. A suffix of t is nested, if it occurs elsewhere in t. Let s be a suffix of t. A prefix p of s is nested, if $p = \varepsilon$ or there is a suffix s' of t, such that $|s'| > |s|$ and p is a prefix of s'. In other words, a nested prefix is empty or has another occurrence as a prefix of a longer suffix of t.[2]

[1] both are faster than the offline constructions of [Wei73] and [CS85], which are also $\mathcal{O}(n)$.

[2] Note that this definition is not symmetric, as it refers to suffixes of t, and prefixes of suffixes of t.

3 The Suffix Tree Family

We give a rather liberal definition of suffix trees, and then three more restricted instances of it, called the atomic suffix tree, the position suffix tree, and the compact suffix tree. We want to study all three of them, since sometimes a construction with inferior theoretical worst case or average case space bounds may be superior in practice, due to a smaller constant factor during construction, or better speed of traversal. The three variants are related by a transformation called edge contraction. Its inverse (edge splitting) will later prove to be useful for suffix tree construction.

An $\mathcal{A}^+$-tree is a rooted tree with edge labels from $\mathcal{A}^+$. For each $a \in \mathcal{A}$, a node k has at most one a-edge $k \xrightarrow{aw} k'$. By $path(k)$ we denote the concatenation of the edge labels on the path from the *root* to the node k. Due to the requirement of unique a-edges at each node, paths are also unique and we can denote k by $\overline{w}$, if and only if $path(k) = w$.[3] We say that a string u occurs in the tree, if and only if there is a node $\overline{uv}$, for some string v.

Definition 3.1 A suffix tree $\mathcal{S}_t$ for a string t is an $\mathcal{A}^+$-tree such that w occurs in $\mathcal{S}_t$, if and only if w is a t-word. □

The name suffix tree is justified by the observation that all non-nested suffixes of t correspond to leaves of $\mathcal{S}_t$. See Figure 1 for examples.

Definition 3.2 Let $\uplus$ denote the disjoint union. Edge contraction is a relation on $\mathcal{A}^+$-trees, denoted here by their edge sets:

$(E \uplus \{\overline{s} \xrightarrow{u} \overline{w}, \overline{w} \xrightarrow{u'} \overline{v}\}) \overset{\overline{w}}{\Longrightarrow} (E \uplus \{\overline{s} \xrightarrow{uu'} \overline{v}\})$, if $\overline{w} \xrightarrow{u'} \overline{v}$ is the only outgoing edge of $\overline{w}$.

If $E \overset{\overline{w}}{\Longrightarrow} E'$, we say that $\overline{w}$ is eliminated by edge contraction, or reading from right to left, $\overline{w}$ is introduced by edge splitting. Omitting the superscripts, we use $\Longrightarrow^*$ to denote the reflexive-transitive closure of $\Longrightarrow$. □

Lemma 3.3 Let E and E' be $\mathcal{A}^+$-trees, such that $E \Longrightarrow E'$. Then a word occurs in E, if and only if it occurs in E'. □

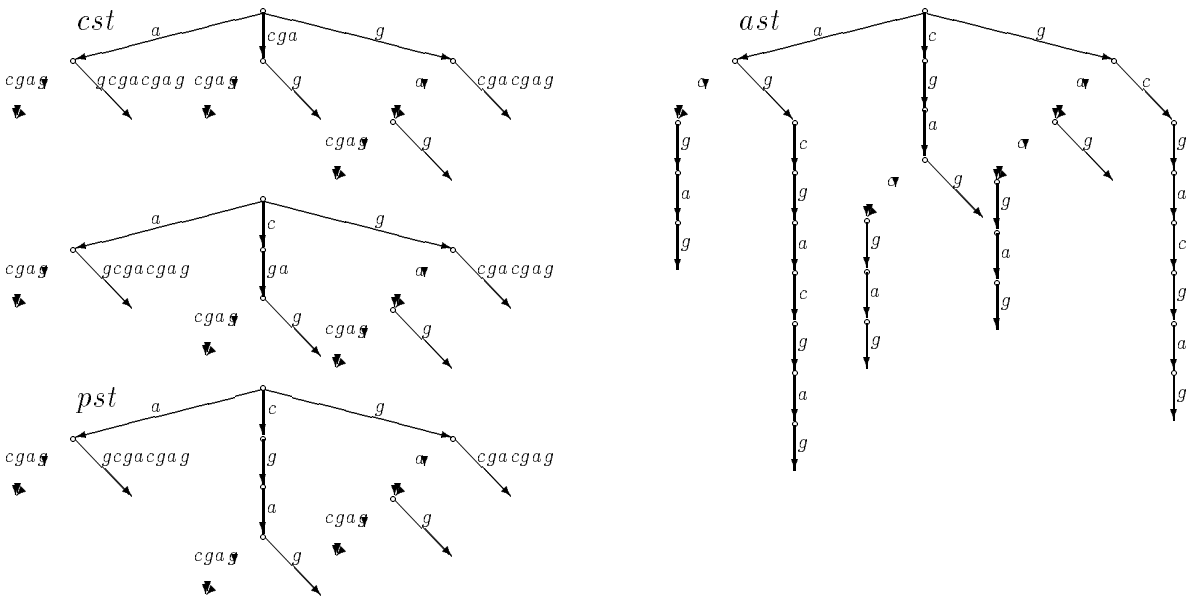
Lemma 3.4 Let E be an $\mathcal{A}^+$-tree, and E' be a suffix tree for t. Then E is a suffix tree for t, if and only if $E \Longleftrightarrow E'$, where $\Longleftrightarrow = (\Longrightarrow \cup \Longrightarrow^{-1})^*$. □

Definition 3.5 Let $\mathcal{S}_t$ be a suffix tree for t, given by its edge set E.

1. If $|E|$ is maximal, $\mathcal{S}_t$ is called atomic suffix tree for t and denoted by $ast(t)$.
2. If for all $\overline{w} \xrightarrow{au} \overline{v} \in E$ either wa is not unique in t and $u = \varepsilon$ or wa is unique in t and $\overline{v}$ is a leaf, then $\mathcal{S}_t$ is called position suffix tree for t and denoted by $pst(t)$.
3. If $|E|$ is minimal, $\mathcal{S}_t$ is called compact suffix tree for t and denoted by $cst(t)$. □

[3] This is a most elegant, but also deceptive convention, taken from [Ukk93]. It is easy to express relationships between tree nodes, e.g. $\overline{cw}$ and $\overline{w}$, that are quite unrelated in the tree structure.

Figure 1: Different suffix trees for the string *agcgacgag*.



It is easy to see that $ast(t)$ is the normal form of all suffix trees for t under edge splitting, while $cst(t)$ is their normal form under edge contraction. We can obtain $pst(t)$ from $ast(t)$ by applying edge contraction as long as one of the involved nodes is a leaf. Thus the position suffix tree is a form of an intermediate level of compactness. It is named after the "position tree" considered in [KBG87, MR80], where a sentinel \$ is used to create a unique leaf for each suffix of t. In "position trees" these leaves are usually labelled with the start position of the corresponding suffix in t.

The following is known about the space requirements for representing these trees:

1. $ast(t)$ has $\mathcal{O}(n^2)$ nodes (e.g. $t = a^n b^n$). However, isomorphic subtrees[4] can be shared [CS85]. We shall elaborate on this in section 6. Sharing brings the space requirement down to $\mathcal{O}(n)$. However, subtree sharing may be impossible, when leaves are to be annotated with extra information.

2. $pst(t)$ has $\mathcal{O}(n^2)$ nodes in the worst case (e.g. $t = a^n b^n a^n b^n$) [AHU74]. Under realistic assumptions, the expected number of nodes is $\mathcal{O}(n)$ [AS92].

3. $cst(t)$ has $\mathcal{O}(n)$ nodes, as all inner nodes are branching, and there are at most n leaves. The edge labels can be represented in constant space by a pair of indices into t. This is necessary to achieve a theoretical worst case bound of $\mathcal{O}(n)$. In practice, this is quite a delicate choice of representation in a virtual memory environment. Traversing the tree and reading the edge labels will create random-like accesses into the text, and can lead to paging problems. This phenomenon will be discussed in some detail in section 5.5.

[4] Two $\mathcal{A}^+$-trees are isomorphic, if they can be obtained from each other by renaming the nodes.

4 Functional Suffix Tree Algorithms

In this section, we present functional suffix tree algorithms. The first one, called *lazyTree*, is new. The other two, called *naiveOnline* and *naiveInsertion* are simplified and less efficient versions of Ukkonen's and McCreight's algorithms. The simplification as well as the loss of efficiency result from the need to avoid local updates of the tree during its construction.

Let us be more precise about what is new with *lazyTree*: There are two simple intuitive approaches to suffix tree construction. One is by successively inserting the suffixes of t into an initially empty tree. This view is used as a starting point for the derivation of Weiner's, McCreight's and also Ukkonen's method. The construction is driven by iteration over the input text, from which the suffixes are taken in right to left or left to right order. It is an imperative idea by nature, as it uses successive tree updates.

The alternative approach centers on the result data structure, the suffix tree. It first determines the outgoing edges of the *root*, and then constructs their subtrees recursively in a top-down manner. No updates to the tree are necessary, and so this approach is declarative by nature. To our knowledge, this approach has not been studied before[5], probably because the known algorithms based on the imperative approach seemed to leave no room for improvement. We shall see that this is only partly true.

Suffix trees imply a lexicographic ordering of all suffixes of a text. So it is easy to read the suffix array of [MM93] from the tree. In this sense *lazyTree* constructs suffix arrays in a top-down (and left to right) fashion.

4.1 The Lazy Suffix Tree Construction

We call a suffix tree construction (potentially) lazy when it constructs the suffix tree for the complete text from the *root* towards the leaves. This has the advantage that the construction may be interleaved with tree traversal — paths of the suffix tree need to be constructed (only) when being traversed for the first time. This kind of incrementality is achieved for free when implementing the lazy construction in a lazy language. It can be simulated in an eager language by explicit synchronization between construction and (all) traversal routines.

We start with an explanation: Write down the *root* with a sorted list l of all non-nested[6] suffixes of t. Let $l_a = \{s \mid as \in l\}$, for each $a \in \mathcal{A}$. Then $pst(t)$ emerges by creating, for each nonempty l_a, an a-edge leading to the subtree recursively constructed for l_a. The recursion terminates with a leaf edge when l_a becomes unitary.

Using a *sorted* list of suffixes only helps when doing this on paper — all the process needs is to group the suffixes according to their first letter, and then choose a common prefix of each group for the edge label. This construction is reflected literally in the functional implementation shown below, except that nested suffixes are not eliminated initially. Rather, they are eliminated when they become empty. Figure 2 exemplifies the process of construction.

Figure 3 declares the data types for suffix trees. We use the lazy functional language Haskell [FHPW92]. The alphabet is a type parameter to our program, denoted *alf*. The text is a

[5] Closest to this is the $\mathcal{O}(\log n)$ parallel algorithm using n processors of [AIL+88], which uses a top-down phase followed by a bottom-up phase that updates the tree.

[6] Nested suffixes do not create nodes.

Figure 2: Phases of the lazy construction of the position suffix tree for *agcgacgag*.

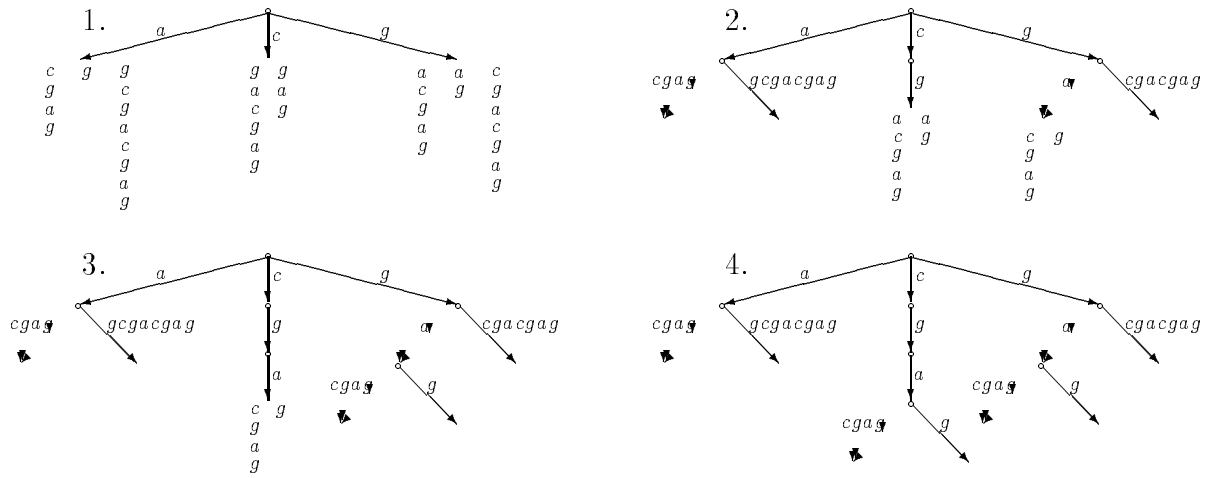


Figure 3: Suffix Tree Data Types

```
data STree alf = Leaf | Branch [(Label alf,STree alf)] deriving Eq
type Label alf = ([alf],Int)
type EdgeFunction alf = [[alf]]->(Int,[[alf]])
```

string (list) of letters from *alf*. Edge labels are represented as pairs (s, l), where s is a suffix of the text that contains the edge label as a prefix of length l. (We have experimented with a variety of edge label representations, but this one is the most convenient and space efficient.) A suffix tree is either a *Leaf*, or a *Branch* node with a list of $(edgelabel, subtree)$-pairs. These data structures are also used in all later functional algorithms.

An *edgeFunction* takes a list of suffixes and splits off a common prefix. Different edge functions are supplied for *ast*, *pst*, and *cst*.

The function *lazyTree* in Figure 4 constructs *ast*, *pst*, or *cst*, depending on the edge function supplied. It takes the list of all non-empty suffixes of the text, including the nested suffixes. It groups them by the first letter, applies the edge function, and constructs subtrees recursively.

Figure 5 gives three edge functions: *edge_ast* is trivial, since the first (only) letter of the edge label has already been split off. *edge_pst* proceeds similarly, but takes the whole suffix as an edge label once a suffix list has become unitary. This requires elimination of nested suffixes when they become empty. *edge_cst* extracts as the edge label the longest common prefix of its suffixes.

Figure 4: The *lazyTree* Construction

```
lazyTree::(Eq alf)=>(EdgeFunction alf)->[alf]->[alf]->STree alf
lazyTree edge alpha t
  = sTr (suffixes t)
    where sTr [[]] = Leaf
          sTr ss = Branch [((a:sa,1+cpl),sTr ssr) | a<-alpha,
                                                    sa:ssa<-[select ss a],
                                                    (cpl,ssr)<-[edge (sa:ssa)]]

select::(Eq alf)=>[[alf]]->alf->[[alf]]   -- select suffixes starting with a
select ss a = [u | c:u<-ss, a == c]

suffixes::[alf]->[[alf]]                -- returns all non-empty suffixes
suffixes aw@(_:w) = aw:suffixes w
suffixes [] = []

lazy_ast::(Eq alf)=>[alf]->[alf]->STree alf
lazy_ast = lazyTree edge_ast

lazy_pst::(Eq alf)=>[alf]->[alf]->STree alf
lazy_pst = lazyTree edge_pst

lazy_cst::(Eq alf)=>[alf]->[alf]->STree alf
lazy_cst = lazyTree edge_cst
```

Figure 5: Three Edge Functions

```
edge_ast::EdgeFunction alf
edge_ast ss = (0,ss)

edge_pst::(Eq alf)=>EdgeFunction alf
edge_pst = g.elimNested
           where g [s] = (length s,[[]])
                 g ss = (0,ss)

elimNested::(Eq alf)=>[[alf]]->[[alf]]
elimNested [s] = [s]
elimNested awss@((a:w):ss) | [] == [0 | c:_<-ss, a /= c] = [a:s | s<-rss]
                           | otherwise                   = awss
                             where rss = elimNested (w:[u | _:u<-ss])

edge_cst::(Eq alf)=>EdgeFunction alf
edge_cst [s] = (length s,[[]])
edge_cst awss@((a:w):ss) | [] == [0 | c:_<-ss, a /= c]  = (1+cpl,rss)
                         | otherwise                    = (0,awss)
                           where (cpl,rss) = edge_cst (w:[u | _:u<-ss])
```

4.2 Ukkonen's online Suffix Tree Construction, Functional Version

In this section, we review Ukkonen's linear-time online suffix tree construction. The differences in our treatment compared to [Ukk93] are the following:

- While Ukkonen derives his construction in an operational style using the atomic suffix tree as an intermediate step, we give a more direct and declarative presentation based on properties of suffixes.
- This approach leads to a more transparent construction, eases correctness arguments and also leads to some minor simplifications.
- It reveals the point where an implementation in a declarative language must proceed differently from Ukkonen's construction, which uses local updates to global data structures and, hence, is inherently imperative [PMN88].

Online construction means generating a series of suffix trees for longer and longer prefixes of the text. While $cst(\varepsilon)$ is trivial (just the *root* with no edges), we study the step from $cst(t)$ to $cst(ta)$, where t is a string and a a letter. Since $cst(ta)$ must represent all ta-words, we consider all new ta-words, i.e. all ta-words, which do not occur in $cst(t)$. Every new ta-word is obviously a nonempty suffix of ta. Let sa be a new ta-word. Then $\overline{sa}$ has to be a leaf in $cst(ta)$, since otherwise sa would be a t-word and hence occur in $cst(t)$.

If $\overline{s}$ is a leaf in $cst(t)$, then a leaf-edge $\overline{b} \xrightarrow{w} \overline{s}$ of $cst(t)$ gives rise to a leaf edge $\overline{b} \xrightarrow{wa} \overline{sa}$ in $cst(ta)$. This observation led Ukkonen to the idea of representing such leaf edges by "open" edges of the form $\overline{b} \xrightarrow{(i,\infty)} \overline{L}$, where $\overline{L}$ denotes a leaf and (i, ∞) denotes the suffix of t starting at position i, whatever the current length of t is. Hence the suffix tree produced by the online construction will be represented as a data structure with three components:

- the global text t,
- the global value $length = |t|$,
- the tree structure itself, with edges of the form $\overline{b} \xrightarrow{(l,r)} \overline{v}$, where the index pair (l, r) represents the edge label $t_l \ldots t_{\min\{length, r\}}$.

Note that the edge label $t_l \ldots t_r$ may occur several times in t, in which case the choice of (l, r) is arbitrary. The global value $length$ and the special right index value ∞ are introduced for the sake of online construction. While $length$ grows implicitly with the text, so do labels of leaf edges. To enter a new suffix sa into the tree, nothing must be done when $\overline{s}$ is a leaf. Hence we only have to consider the case that $\overline{s}$ is not a leaf in $cst(t)$, or equivalently s is a nested suffix of t.

Definition 4.1 A suffix sa of ta is relevant, if s is a nested suffix of t and sa is not a t-word. □

Now we can give an informal description of how to construct $cst(ta)$ from $cst(t)$:

(∗) Insert all relevant suffixes sa of ta into $cst(t)$.

Before we describe how to insert a relevant suffix of ta into $cst(t)$, we show that the relevant suffixes of ta form a contiguous segment of the list of all suffixes of ta, whose bounds are marked by "active suffixes":

Definition 4.2 The active suffix of t, denoted by $\alpha(t)$, is the longest nested suffix of t.[7] □

Example 4.3 Consider the string $agcgacgag$ and a list of columns, where each column contains the list of all suffixes of a prefix of this string. The relevant suffixes in each column are marked by the symbol ↓ and the active suffix is printed in bold face.

$\boldsymbol{\varepsilon}$	$\downarrow a$	ag	agc	$agcg$	$agcga$	$agcgac$	$agcgacg$	$agcgacga$	$agcgacgag$
	$\boldsymbol{\varepsilon}$	$\downarrow g$	gc	gcg	$gcga$	$gcgac$	$gcgacg$	$gcgacga$	$gcgacgag$
		$\boldsymbol{\varepsilon}$	$\downarrow c$	cg	cga	$cgac$	$cgacg$	$cgacga$	$cgacgag$
			$\boldsymbol{\varepsilon}$	$\boldsymbol{g}$	$\downarrow ga$	gac	$gacg$	$gacga$	$gacgag$
				ε	$\boldsymbol{a}$	$\downarrow ac$	acg	$acga$	$acgag$
					ε	$\boldsymbol{c}$	$\boldsymbol{cg}$	$\boldsymbol{cga}$	$\downarrow cgag$
						ε	g	ga	$\downarrow gag$
							ε	a	$\boldsymbol{ag}$
								ε	g
									ε

Lemma 4.4 For all $a \in \mathcal{A}$ and all suffixes s of t we have: sa is a relevant suffix of ta if and only if $|\alpha(t)a| \geq |sa| > |\alpha(ta)|$.

Proof sa is a relevant suffix of $ta \iff s$ is a nested suffix of t and sa is not a t-word $\iff$ $|\alpha(t)| \geq |s|$ and sa is not a nested suffix of $ta \iff |\alpha(t)a| \geq |sa|$ and $|sa| > |\alpha(ta)| \iff$ $|\alpha(t)a| \geq |sa| > |\alpha(ta)|$. □

Lemma 4.5 $\alpha(ta)$ is a suffix of $\alpha(t)a$.

Proof Since both $\alpha(ta)$ and $\alpha(t)a$ are suffixes of ta, it suffices to show $|\alpha(t)a| \geq |\alpha(ta)|$. If $\alpha(ta) = \varepsilon$ this is obviously true. Let $\alpha(ta) = wa$. Since wa is a nested suffix of ta, we have $uwav = t$ for some strings u, v. Hence w is a nested suffix of t. Since $\alpha(t)$ is the longest nested suffix of t, we have $|\alpha(t)| \geq |w|$ and hence $|\alpha(t)a| \geq |wa| = |\alpha(ta)|$. □

By Lemma 4.4 we know that the relevant suffixes of ta are "between" $\alpha(t)a$ and $\alpha(ta)$. Hence by Lemma 4.5 $\alpha(ta)$ is the longest suffix of $\alpha(t)a$ that is a t-word. Based on this fact we can refine algorithm (∗) as follows:

(∗∗) Take the suffixes of $\alpha(t)a$ one after the other by decreasing length and insert them into $cst(t)$ until a suffix is found which is a t-word, and therefore equals $\alpha(ta)$.

After having explained how to find the relevant suffixes of ta, we make precise how we insert them.

[7] The canonical reference pair (see Definition 4.6) of an active suffix corresponds to the notion "active point" introduced in [Ukk93].

Definition 4.6 Let E be an $\mathcal{A}^+$-tree and s be a string that occurs in E. We call $(\overline{b}, u)$ reference pair of s with respect to E, if $\overline{b}$ is the *root* or a branching node in E and $s = bu$. If b is the longest such prefix of s, then $(\overline{b}, u)$ is called canonical reference pair of s with respect to E. In such a case we write $\widehat{s} = (\overline{b}, u)$. □

Let sa be a relevant suffix of ta and E be the $\mathcal{A}^+$-tree in which sa has to be inserted. Let $\widehat{s} = (\overline{b}, u)$ and consider the following cases:

1. If $\overline{s}$ is a node in $cst(t)$ then $u = \varepsilon$ and $\overline{s} = \overline{b}$ has no a-edge, since otherwise sa would be a t-word. Thus we only add a new open a-edge $\overline{b} \xrightarrow{(i,\infty)} \overline{L}$, where $i = length = |ta|$.

2. If $\overline{s}$ is not a node in $cst(t)$, then $u = cw$ for some letter c and some string w. Let $\overline{b} \xrightarrow{(l,r)} \overline{v}$ be a c-edge in E and let $k = l + |w|$. Then we introduce $\overline{s}$ by splitting $\overline{b} \xrightarrow{(l,r)} \overline{v}$ into $\overline{b} \xrightarrow{(l,k)} \overline{s} \xrightarrow{(k+1,r)} \overline{v}$ and add a new open a-edge $\overline{s} \xrightarrow{(i,\infty)} \overline{L}$, where $i = length = |ta|$.

The trees resulting from 1. respectively 2. above will be denoted by $E \sqcup (\widehat{s}, i)$. Putting it altogether, we can describe algorithm $(**)$ by specifying a function *update* which inserts the relevant suffixes of ta into $cst(t)$ and computes $\alpha(ta)$:

$$\begin{array}{lcll} update(E, sa) & = & (E, sa), & \textbf{if } sa \text{ is a } t\text{-word} \\ & = & (E \sqcup (\widehat{s}, i), \varepsilon), & \textbf{else if } s = \varepsilon \\ & = & update(E \sqcup (\widehat{s}, i), drop(1, sa)), & \textbf{otherwise} \end{array}$$

Lemma 4.7 $update(cst(t), \alpha(t)a)$ returns the pair $(cst(ta), \alpha(ta))$. □

Here $drop(k, w)$ denotes the string w with the first k symbols removed. There are two critical operations in this algorithm: Checking whether sa is a t-word, and splitting some edge $\overline{b} \xrightarrow{(l,r)} \overline{v}$ to introduce $\overline{s}$, if necessary. Both are trivial once we have computed the canonical reference pair $(\overline{b}, u)$ of s: If $u = \varepsilon$ then sa is a t-word, if and only if $\overline{b}$ has an a-edge. If $u = cw$ for a letter c and a string w, then there is a c-edge $\overline{b} \xrightarrow{(l,r)} \overline{v}$ and sa is a t-word, if and only if $t_{l+|u|} = a$.

The easiest way to determine $(\overline{b}, u)$ is to follow the path for s down from the *root*, anew for each suffix s. This leads to a non-linear construction, as the length of this path can be $\mathcal{O}(n)$ in the worst case. *naiveOnline*, our functional version of Ukkonen's algorithm, uses this approach, since implementing this algorithm without local updates adds no extra overhead: Along the path from the *root* to $\overline{s}$, the tree may be de- and reconstructed in constant time for each node visited. Thus the local update is turned into a global one with no effect on asymptotic efficiency.

The Haskell program for *naiveOnline* is given in Figure 6[8]. *naiveOnline* iterates *update*, as explained in the text. Its first argument, the active suffix, is represented in the same way as edge labels. *insRelSuff* traverses (and reconstructs) the tree for each relevant suffix to be inserted. *isTword* is used to terminate the insertion of (relevant) suffixes when one is found that is a t-word already.

[8]The code is a little obscured by the use of as-patterns. `cusn@(cus@(cu@(c:_),culen),node)` gives names to all subpatterns of `(((c:_),culen),node)`. This has a beneficial effect on runtime and storage consumption, and we decided to show the programs exactly as measured.

Figure 6: The *naiveOnline* Algorithm

```
isTword::(Eq alf)=>(Label alf)->(STree alf)->Bool
isTword (a:w,0) (Branch es) = [] /= [0 | ((c:_,_),_)<-es, a == c]
isTword (a:w,wlen+1) (Branch es)
  | Leaf == node || wlen < ulen  = w!!wlen == u!!wlen
  | otherwise                    = isTword (drop ulen w,wlen-ulen) node
    where (u,ulen,node) = head [(u,culen-1,node) | ((c:u,culen),node)<-es, a == c]

update::(Ord alf)=>(STree alf, Label alf) -> (STree alf, Label alf)
update (root,(s,slen))
  | isTword (s,slen) root  = (root,(s,slen+1))
  | 0 == slen              = (root',(tail s,0))
  | otherwise              = update (root',(tail s,slen-1))
    where root' = insRelSuff (s,slen) root

insRelSuff::(Ord alf)=>(Label alf)->(STree alf)->(STree alf)
insRelSuff (aw@(a:w),0) (Branch es)
  = Branch (g es)
    where g [] = [((aw,length aw),Leaf)]
          g (cusn@((c:u,culen),node):es')
            | a > c     = cusn:g es'
            | otherwise = ((aw,length aw),Leaf):cusn:es'
insRelSuff (aw@(a:w),slen) (Branch es)
  = Branch (g es)
    where g (cusn@(cus@(cu@(c:_),culen),node):es')
            | a /= c                        = cusn:g es'
            | Leaf /= node && slen >= culen = (cus,node'):es'
            | head x < head y               = ((cu,slen),Branch [ex,ey]):es'
            | otherwise                     = ((cu,slen),Branch [ey,ex]):es'
              where node' = insRelSuff (drop culen aw,slen-culen) node
                    x = drop slen cu
                    y = drop slen aw
                    ex | Leaf == node = ((x,length x),Leaf)
                       | otherwise    = ((x,culen-slen),node)
                    ey = ((y,length y),Leaf)

naiveOnline::(Ord alf)=>[alf]->STree alf
naiveOnline t = fst (until stop update (Branch [],(t,0)))
                where stop (_,(s,slen)) = [] == drop slen s
```

4.3 McCreight's Suffix Tree Construction, Functional Version

In this section (and in section 5.3) we consider a text $t = t_1 \ldots t_n$, $n \geq 2$ in which the final letter appears nowhere else in t. Let s be a suffix of t. $head_t(s)$ denotes the longest nested prefix of s, whenever $t \neq s$. Furthermore let $head_t(t) = \varepsilon$. If $s = head_t(s)u$ then we denote u by $tail_t(s)$. $T_t(s)$ denotes the $\mathcal{A}^+$-tree, such that w occurs in $T_t(s)$, if and only if there is a suffix s' of t, such that $|s'| \geq |s|$ and w is a prefix of s'.

The general structure of McCreight's algorithm [McC76] is to compute $cst(t)$ by successively inserting the suffixes s of t into the tree. Notice that the intermediate trees are not suffix trees. More precisely, given $T_t(as)$, where as is a suffix of t, the algorithm computes the canonical reference pair $(\overline{h}, q)$ of $head_t(s)$ and the starting position j of $tail_t(s)$ and returns $T_t(s) = T_t(as) \sqcup ((\overline{h}, q), j)$. The easiest way to determine $(\overline{h}, q)$ and j is to follow the path for s in $T_t(as)$ down from the *root*, until one "falls out of the tree" (as guaranteed by the uniqueness of the final symbol in t). This process can be described by the function *scan*:

$$
\begin{array}{lcll}
scan(E, \overline{b}, i) & = & ((\overline{b}, \varepsilon), i), & \textbf{if } \overline{b} \text{ has no } t_i\text{-edge in } E \\
& = & ((\overline{b}, p), i + |p|), & \textbf{else if } |p| < r - l + 1 \\
& = & scan(E, \overline{v}, i + |p|), & \textbf{otherwise} \\
& & \multicolumn{2}{l}{\textbf{where } \overline{b} \xrightarrow{(l,r)} \overline{v} \text{ is a } t_i\text{-edge in } E} \\
& & \multicolumn{2}{l}{\quad\quad p \text{ is the longest common prefix of } t_l \ldots t_r \text{ and } t_i \ldots t_n}
\end{array}
$$

Using *scan*, it is easy to describe how to insert a suffix of t that starts at position $i \leq n$:

$$
\begin{array}{lcl}
insertSuffix(E, i) & = & E \sqcup ((\overline{h}, q), j) \\
& & \textbf{where } ((\overline{h}, q), j) = scan(E, root, i)
\end{array}
$$

naiveInsertion, our functional version of McCreight's algorithm, is based on iterated use of *insertSuffix*. Again, the tree is de- and reconstructed during the scan from the *root*, turning the local updates into global ones without extra overhead.

The Haskell program for *naiveInsertion* is shown in Figure 7. It looks simpler than *naiveOnline*, basically since it does not use nested iteration.

4.4 Asymptotic and Empirical Efficiency of the Functional Algorithms

In all three algorithms, when a leaf edge is constructed, an operation $length(s)$ is used to determine the length of the edge label. For the worst case analyses it does not make a difference whether we consider the efficiency of $length(s)$ to be $\mathcal{O}(length(s))$ or $\mathcal{O}(1)$. For the expected case, it does. We shall consider it an $\mathcal{O}(1)$-operation for the following reasons:

- In principle, it is possible to avoid this operation by initially pairing all suffixes with their lengths, and decreasing these lengths as suffixes are shortened. This makes the programs less readable (and actually slows them down by a factor of 2).
- If t is represented as an array rather than a list, this operation becomes $\mathcal{O}(1)$ anyway.
- Even with the program as shown, the length of an edge label is — due to laziness — not calculated until the edge is actually traversed. Then, the letters of s are read anyway, and this amortizes the cost of calculating $length(s)$.

Figure 7: The *naiveInsertion* Algorithm

```
insertSuffix::(Ord alf)=>(STree alf)->[alf]->STree alf
insertSuffix (Branch es) aw@(a:w)
  = Branch (g es)
    where g [] = [((aw,length aw),Leaf)]
          g cusnes@(cusn@(cus@(cu@(c:u),culen),node):es')
            | a > c                      = cusn:g es'
            | a < c                      = ((aw,length aw),Leaf):cusnes
            | Leaf /= node && xlen == 0 = (cus,insertSuffix node y):es'
            | head x < head y            = ((cu,cpl),Branch [ex,ey]):es'
            | otherwise                  = ((cu,cpl),Branch [ey,ex]):es'
              where cpl | Leaf == node = lcp cu aw
                        | otherwise    = lcp (take culen cu) aw
                    x = drop cpl cu
                    xlen = culen-cpl
                    y = drop cpl aw
                    ex = ((x,xlen),node)
                    ey = ((y,length y),Leaf)

lcp::(Eq alf)=>[alf]->[alf]->Int
lcp u w = length (takeWhile (True ==) [c == a|(c,a)<-zip u w])

naiveInsertion::(Ord alf)=>[alf]->STree alf
naiveInsertion t = foldl insertSuffix (Branch []) (suffixes t)
```

Let $|t| = n$, and $|\mathcal{A}| = k$. The asymptotic efficiency of *naiveOnline* and *naiveInsertion* is as follows: There are $\mathcal{O}(n)$ nodes created. The path length to access each node is $\mathcal{O}(n)$ in the worst and $\mathcal{O}(\log n)$ in the expected case [AS92]. Selecting the suitable branch at each node introduces a factor of $\mathcal{O}(k)$. This gives a worst case of $\mathcal{O}(kn^2)$ and an expected case of $\mathcal{O}(kn \log n)$. The alphabet factor k has in fact a strong influence for large alphabets, since nodes close to the *root* will have close to k outgoing edges. This is partly compensated for by the tree becoming flatter for larger alphabets.

The asymptotic efficiency of *lazyTree* is determined by considering the number of letters read from all suffixes, and the number of operations per letter read. The sum of suffix lengths is $n(n+1)/2$. For $t = a^{n-1}\$$, all suffixes except for the longest are read to the last letter. Since the functional *lazyTree* uses iteration over $\mathcal{A}$[9] to group suffixes according to their first letter, each letter is inspected k times. This yields a tight worst case of $\mathcal{O}(kn^2)$, achieved for $a^{n-1}\$$.

The expected length of the longest repeated subword is $\mathcal{O}(\log n)$ according to [AS92]. Since no suffix is read beyond the point where it becomes unique, we obtain an average case efficiency of $\mathcal{O}(kn \log n)$.

Note that while *lazyTree*'s factor of k stems from the iteration over the alphabet used for grouping suffixes, for *naiveInsertion* and *naiveOnline* this factor arises from checking if an a-edge occurs in a list of $\mathcal{O}(k)$ edges.

[9] This could be avoided if we had updateable arrays available, cf. the imperative implementation of *lazyTree*.

In this analysis we have abstracted from some minor difference between *naiveOnline* and *naiveInsertion*. While traversing the tree towards $\overline{s}$ in *naiveOnline*, we need to compare only the first letter of the edge label to the first letter of the current suffix of s. The subsequent letters of the edge label must coincide with those of s, since s is a nested suffix of t and therefore occurs in $cst(t)$. With this consideration, an edge can be traversed in constant time. *naiveInsertion* iterates suffix insertion by traversing the tree with the given suffix s until it "falls out of the tree". In contrast to *naiveOnline*, s does not (generally) occur in the tree, and all letters along the traversed edge labels must be compared to those of s. This could account for a small speed advantage of *naiveOnline*.

We present some empirical results with the functional implementations shown above. We used the Chalmers Haskell compiler [Aug93]. Note that there may be Haskell compilers that produce better code [HL93]. All algorithms were measured on random texts (Bernoulli-distribution) over alphabets with various sizes ($k = 4, 20, 50, 90$), running on a *SPARCstation* 10/41 with 32 MB. We also confirmed our measurements with "natural" data from the yeast genome. Measurements were done with the unix tool *rusage* and averaged over 10 runs.

From Figure 10–13 we obtain the following results:

- all algorithms show close to linear behaviour.
- independent of the alphabet size, *lazyTree* is the fastest of the three algorithms, with the advantage decreasing for larger alphabets. This effect lies within the constant factors. All have an alphabetic factor of $\mathcal{O}(k)$. For *lazyTree*, this factor is truly k, due to iteration over the alphabet. For *naiveInsertion* and *naiveOnline*, its expected value is $\leq k/2$, due to searching through sorted subtree lists of length $\leq k$. This becomes visible with increasing k.
- *naiveInsertion* is always somewhat faster than *naiveOnline*. The difference in reading edge labels, as explained above, does not pay off for *naiveOnline*, probably because the expected length of inner edge labels is close to 1.

5 Imperative Suffix Tree Algorithms

Although it was not in the original intent of this work, it turned out to be very instructive to further refine the functional algorithms into imperative programs, and redo the analysis of section 4.4. Again, the imperative implementation of *lazyTree* is new, while the imperative versions of *naiveOnline* and *naiveInsertion* are simplifications of the well-known linear-time algorithms *ukk* and *mcc*.

5.1 lazyTree, Imperative Version

A careful imperative implementation of *lazyTree* is an interesting topic of its own right, and we have not yet fully explored all alternatives. Our current version retains the basic recursion structure, uses counting sort [CLR90] for grouping suffixes according to first letters, and a naive function to determine longest common prefixes of those suffixes starting with the same letter.

5.2 Ukkonen's online Suffix Tree Construction, Imperative Version

We now return to the development in section 4.2 and further refine *naiveOnline* to the linear-time construction *ukk*. For achieving linear behaviour, we represent the suffix s by its canonical reference pair $(\overline{b}, u)$ and implement a function *link* which provides direct access from $(\overline{b}, u)$ to the representation of the longest proper suffix of s. The idea is to implement the function *link* using "suffix links" between the branching nodes.

Definition 5.1 Let E be an $\mathcal{A}^+$-tree, and B be the set of its branching nodes. We define a function $f : B \setminus \{root\} \to B$ by $f(\overline{cw}) = \overline{w}$, if and only if $(\overline{cw}, \overline{w}) \in B \setminus \{root\} \times B$. An element $(\overline{cw}, \overline{w}) \in f$ is called suffix link [McC76] and f is called suffix link function for E. □

If $E = cst(t)$ for some string t, then $f(\overline{b})$ is well-defined for all $\overline{b} \in B \setminus \{root\}$, due to the fact that if cw is a branching t-word, this is also true for w.

Suppose that cw is a nested suffix of t. If $\widehat{cw} = (\overline{b}, u)$ and $\overline{b} \neq root$ then $(f(\overline{b}), u)$ is a reference pair for w. It need not be canonical, which can be seen in the following example.

Example 5.2 Consider the compact suffix tree for $t = agcgacgag$ in Figure 1. Obviously ag is a nested suffix of t. We have $\widehat{ag} = (\overline{a}, g)$ and $(f(\overline{a}), g) = (root, g) \neq \widehat{g} = (\overline{g}, \varepsilon)$, i.e. $(f(\overline{a}), g)$ is not canonical. □

Thus it is necessary to make reference pairs canonical. This can be done using the function *canonize*:

$$
\begin{array}{lcll}
canonize(E, (\overline{b}, \varepsilon)) & = & (\overline{b}, \varepsilon) & \\
canonize(E, (\overline{b}, cw)) & = & (\overline{b}, cw), & \textbf{if } |w| < r - l \\
 & = & canonize(E, (\overline{v}, drop(r-l, w))), & \textbf{otherwise} \\
 & & \textbf{where } \overline{b} \xrightarrow{(l,r)} \overline{v} \in E \text{ is a } c\text{-edge} &
\end{array}
$$

Given the suffix link function f, *link* can be defined in the following way:

$$
\begin{array}{lcll}
link(E, f, (\overline{b}, \varepsilon)) & = & (\overline{b}, \varepsilon), & \textbf{if } \overline{b} = root \\
 & = & (f(\overline{b}), \varepsilon), & \textbf{otherwise} \\
link(E, f, (\overline{b}, cw)) & = & canonize(E, (\overline{b}, w)), & \textbf{if } \overline{b} = root \\
 & = & canonize(E, (f(\overline{b}), cw)), & \textbf{otherwise}
\end{array}
$$

Now we can refine the function *update*, yielding a function *update'*, in which a suffix s is represented by $\widehat{s}$. Notice that for every new node we have to extend the suffix link function f by a new suffix link.

$$
\begin{array}{lcll}
update'(E, f, (\overline{b}, \varepsilon), i) & = & (E, f, canonize(E, (\overline{b}, t_i))), & \textbf{if } \overline{b} \text{ has a } t_i\text{-edge} \\
 & = & (E \sqcup ((\overline{b}, \varepsilon), i), f, (\overline{b}, \varepsilon)), & \textbf{else if } \overline{b} = root \\
 & = & update'(E \sqcup ((\overline{b}, \varepsilon), i), f, (f(\overline{b}), \varepsilon), i), & \textbf{otherwise}
\end{array}
$$

$$
\begin{array}{lcll}
update'(E, f, (\overline{b}, cw), i) & = & (E, f, canonize(E, (\overline{b}, cwt_i))), & \textbf{if } t_{l+|cw|} = t_i \\
 & = & update'(E \sqcup ((\overline{b}, cw), i), f', (\overline{b'}, u'), i), & \textbf{otherwise} \\
 & & \textbf{where } \overline{b} \xrightarrow{(l,r)} \overline{v} \in E \text{ is a } c\text{-edge} & \\
 & & (\overline{b'}, u') = link(E, f, (\overline{b}, cw)) & \\
 & & f' = f \cup \{(\overline{bcw}, \overline{b'u'})\} &
\end{array}
$$

Lemma 5.3 Let $p = t_1 \ldots t_{i-1}$ and $a = t_i$. $update'(cst(p), f, \widehat{\alpha(p)}, i)$ returns the triple $(cst(pa), f', \widehat{\alpha(pa)})$, where f' is the suffix link function for $cst(pa)$. □

Notice that if $u \neq \varepsilon$ and we extend f by the new suffix link $(\overline{s}, \overline{b'u})$, the node $\overline{b'u}$ does not exist at that moment. But since s is a branching pa-word and $b'u$ is a suffix of s, $b'u$ is a branching pa-word, too. Hence $\overline{b'u}$ must be a branching node in $cst(pa)$ and it will be created by the next call of $update'$. Thus the setting of the suffix link $(\overline{s}, \overline{b'u})$ has to be delayed only to the next call of $update'$.

Ukkonen's algorithm is simply an iteration of the function $update'$:

$$\begin{array}{lcll} ukk(E, f, (\overline{b}, u), i) & = & E, & \textbf{if } i = n+1 \\ & = & ukk(E', f', (\overline{b'}, u'), i+1), & \textbf{otherwise} \\ & & \textbf{where } (E', f', (\overline{b'}, u')) = update'(E, f, (\overline{b}, u), i) & \end{array}$$

Theorem 5.4 If $|t| = n$ then $ukk(\emptyset, \emptyset, (root, \varepsilon), 1)$ returns $cst(t)$ in $\mathcal{O}(n)$ time.

Proof The correctness of ukk is obvious. The complexity proof carries over from [Ukk93]. □

5.3 McCreight's Suffix Tree Construction, Imperative Version

We return to the development in section 4.3 and further refine *naiveInsertion* to the linear-time construction *mcc*. To get a linear-time algorithm, $(\overline{h}, q)$ (the canonical reference pair of $head_t(s)$) and j (the starting position of $tail_t(s)$) must be computed in constant time (averaged over all steps). McCreight's algorithm does this by exploiting the following relationships:[10]

Lemma 5.5 Let $head_t(as) = aw$ for some string w. Then

1. $\overline{aw}$ is a branching node in $T_t(as)$,
2. w is a prefix of $head_t(s)$,
3. $w = head_t(s)$, if there is no branching node $\overline{w}$ in $T_t(as)$.

Proof

1. Since aw is a nested prefix of as there is a letter d and a string u such that $as = awdu$. Furthermore there is a suffix as' of t such that $|as'| > |as|$ and $as' = awcv$ for some $c \in \mathcal{A}$ and some $v \in \mathcal{A}^*$. Obviously $d \neq c$, since otherwise awd would be a nested prefix of as. Since awd and awc occur in $T_t(as)$, $\overline{aw}$ is a branching node in $T_t(as)$.
2. Since aw is a nested prefix of as there is a suffix as' of t such that $|as'| > |as|$ and aw is a prefix of as'. Hence w is a prefix both of s and s', which implies that w is a nested prefix of s. Since $head_t(s)$ is the longest nested prefix of s, w is a prefix of $head_t(s)$.

[10] McCreight explicitly uses only the second relationship, see Lemma 1 in [McC76].

3. Since $\overline{aw}$ is a branching node in $T_t(as)$ there are different suffixes as' and as'' of t, s.t. $|as'| > |as''| \geq |as|$, $as' = awcv$ and $as'' = awdu$ for different letters c, d and some strings u, v. This implies $s' = wcv$ and $s'' = wdu$. Hence w occurs in $T_t(as)$ and $\overline{w}$ is not a leaf in $T_t(as)$. Since by assumption $\overline{w}$ is not a branching node in $T_t(as)$ there is a letter $c' \in \mathcal{A}$, s.t. for all suffixes s''' of t with the property $|s'''| \geq |as|$, either w is not a prefix of s''' or $wc'r = s'''$ for some $r \in \mathcal{A}^*$. Since s' is a suffix of t, s.t. $|s'| \geq |as| > |s|$ and w is a prefix of s', we can conclude $wc'r = s' = wcv$. Hence we have $wdu = s'' = s$ because $d \neq c$. Thus wd does not occur in $T_t(as)$, which implies $w = head_t(s)$. □

To exploit these relationships McCreight's algorithm uses a representation of the "head" and the "tail" of the previous suffix and suffix links as auxiliary information.

Definition 5.6 The triple $(\emptyset, (root, \varepsilon), 1)$ is valid for t. Let as be a suffix of t. The triple $(f', (\overline{h}, q), j)$ is valid for s, if and only if the following is true:

1. $f' = \{(\overline{cw}, \overline{w}) \mid \overline{cw} \text{ is a branching node in } T_t(as)\}$,
2. $(\overline{h}, q)$ is the canonical reference pair of $head_t(s)$ with respect to $T_t(as)$,
3. $tail_t(s) = t_j \ldots t_n$. □

f' is the suffix link function for $T_t(s)$ restricted to the branching nodes in $T_t(as)$. It is well-defined, since for all branching nodes $\overline{cw}$ in $T_t(as)$ there is a branching node $\overline{w}$ in $T_t(s)$.

Let $(f, (\overline{b}, u), i)$ be the valid triple for as. We now explain the central idea of McCreight's algorithm, i.e. how to efficiently compute the valid triple $(f', (\overline{h}, q), j)$ for s from $T_t(as)$ and $(f, (\overline{b}, u), i)$. Therefore let us consider the following cases:

1. $(\overline{b}, u) = (root, \varepsilon)$. Then $head_t(as) = \varepsilon$ and $tail_t(as) = as = t_i \ldots t_n$. To compute $(\overline{h}, q)$ and j we have to scan s. Therefore let $((\overline{h}, q), j) = scan(T_t(as), \overline{b}, i+1)$, where $scan$ is as in section 4.3. Notice that $t_{i+1} \ldots t_n = s$. Since in $T_t(as)$ no new branching node was created, f is the suffix link function for $T_t(s)$, restricted to the branching nodes in $T_t(as)$. Thus $(f, (\overline{h}, q), j)$ is valid for s.
2. $(\overline{b}, u) \neq (root, \varepsilon)$. Then $head_t(as) = aw$ for some string w. By Lemma 5.5, Statement 2, w is a prefix of $head_t(s)$. Consider the following subcases:
   - $u = \varepsilon$. Since $\overline{b}$ is not the $root$ but a branching node in $T_t(as)$, $f(\overline{b})$ is well-defined. Let $((\overline{h}, q), j) = scan(T_t(as), f(\overline{b}), i)$. Then $(f, (\overline{h}, q), j)$ is valid for s.
   - $u \neq \varepsilon$. Let $(\overline{b'}, u') = link(T_t(as), f, (\overline{b}, u))$. Since $\overline{b}$ is the $root$ or a branching node in $T_t(as)$, $(\overline{b'}, u')$ is well-defined. Let $f' = f \cup \{(\overline{bu}, \overline{b'u'})\}$. If $u' = \varepsilon$, then $\overline{b'}$ exists in $T_t(as)$. To proceed we have to scan a prefix of $tail_t(as) = t_i \ldots t_n$. Therefore let $((\overline{h}, q), j) = scan(T_t(as), \overline{b'}, i)$. Then $(f', (\overline{h}, q), j)$ is valid for s. If $u' \neq \varepsilon$, $\overline{b'u'}$ is not a branching node in $T_t(as)$. By Lemma 5.5, Statement 3, we have $w = head_t(s)$. Since $b'u' = w = head_t(s)$ and $tail_t(s) = tail_t(as) = t_i \ldots t_n$ the triple $(f', (\overline{b'}, u'), i)$ is valid for s.

Putting it altogether we get the following refinement of *insertSuffix*:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{insertSuffix}'(E, f, (\overline{b}, u), i)} \\
\quad = & (E \sqcup ((\overline{h}, q), j), f, (\overline{h}, q), j), & \textbf{if } u = \varepsilon \\
\quad = & (E \sqcup ((\overline{h}, q), j), f', (\overline{h}, q), j), & \textbf{if } u' = \varepsilon \\
\quad = & (E \sqcup ((\overline{b'}, u'), i), f', (\overline{b'}, u'), i), & \textbf{otherwise} \\
\multicolumn{3}{l}{\quad \textbf{where } (\overline{b'}, u') = \mathit{link}(E, f, (\overline{b}, u))} \\
\multicolumn{3}{l}{\qquad\qquad f' = f \cup \{(\overline{bu}, \overline{b'u'})\}} \\
\multicolumn{3}{l}{\qquad\qquad ((\overline{h}, q), j) \;=\; \mathit{scan}(E, \overline{b}, i+1), \quad \textbf{if } (\overline{b}, u) = (\mathit{root}, \varepsilon)} \\
\multicolumn{3}{l}{\qquad\qquad\qquad\qquad =\; \mathit{scan}(E, f(\overline{b}), i), \quad \textbf{else if } u = \varepsilon} \\
\multicolumn{3}{l}{\qquad\qquad\qquad\qquad =\; \mathit{scan}(E, \overline{b'}, i), \quad \textbf{else if } u' = \varepsilon}
\end{array}
$$

McCreight's algorithm is simply an iteration of the function *insertSuffix*′:

$$
\begin{array}{lll}
\mathit{mcc}(E, f, (\overline{b}, u), i) & = & E, \qquad\qquad \textbf{if } i = n \;\&\; (\overline{b}, u) = (\mathit{root}, \varepsilon) \\
& = & \mathit{mcc}(\mathit{insertSuffix}'(E, f, (\overline{b}, u), i)), \quad \textbf{otherwise}
\end{array}
$$

Theorem 5.7 If $|t| = n$ then $\mathit{mcc}(\{\mathit{root} \xrightarrow{(1,n)} \overline{L}\}, \emptyset, (\mathit{root}, \varepsilon), 1)$ returns $\mathit{cst}(t)$ in $\mathcal{O}(n)$ time.

Proof The correctness of *mcc* is due to the fact that if $(f, (\overline{b}, u), i)$ is valid for as, then $\mathit{insertSuffix}'(T_t(as), f, (\overline{b}, u), i)$ returns the quadruple $(T_t(s), f', (\overline{h}, q), j)$, such that $(f', (\overline{h}, q), j)$ is valid for s. The complexity proof carries over from [McC76]. □

5.4 Asymptotic and Empirical Efficiency of the Imperative Versions

Both *ukk* and *mcc* are linear in the size of the text. Repeated traversal of lists of subtrees when a new one is added leads to a factor of k. This factor could be reduced e.g. to $\log_2 k$ by implementing subtree lists as balanced trees, or by the use of hashing techniques [McC76]. With our simple tree data structure, the asymptotic efficiency is $\mathcal{O}(kn)$ for worst and expected case.

Our imperative version of *lazyTree* uses counting sort [CLR90] for grouping suffixes, avoiding iteration over the letters of the alphabet. Since the number of letters inspected is the same as in the functional implementation, we obtain a worst case of $\mathcal{O}(n^2)$ and an expected case of $\mathcal{O}(n \log_k n)$. Note that there is no alphabetic factor k. This property is shared by the suffix array algorithm of [MM93], whereas all other known suffix tree constructions must use more complicated data structures to reduce the alphabet factor.

Imperative versions of *lazyTree*, *naiveOnline*, *ukk*, *naiveInsertion* and *mcc* were implemented in C. To avoid inefficiencies by dynamic storage allocation, all C-programs represent tree nodes as elements of a statically allocated array. Measurements are shown in Figure 14–17 from $n = 10.000$ to $n = 100.000$.

From these we draw the following conclusions:

- Up to $n = 100.000$, all implementations show close to linear behaviour, irrespective of their asymptotic efficiency,
- *ukk* is always better than its naive version *naiveOnline*,

- between *mcc* and *naiveInsertion*, the same relation holds,
- *mcc* is faster than *ukk*, but the difference is not significant,
- with larger alphabets, the advantage of *ukk* and *mcc* over their naive versions decreases. This is due to the fact that the overhead of navigating through the tree is related to $\log_k n$.

The most interesting finding — to our own surprise — is the behaviour of *lazyTree*:

- *lazyTree*'s running time is practically linear,
- *lazyTree* is comparable to *naiveInsertion* (and half as fast as *mcc* and *ukk*) for $k = 4$,
- *lazyTree* beats all other algorithms for the larger alphabets, showing about five times the speed of the second best (*mcc*) for $k = 90$ and $n = 100.000$.

Since *lazyTree* has no alphabetic factor, k enters only as the logarithm base in $\mathcal{O}(n \log_k n)$, and it becomes faster for larger alphabets.

Let us relate *lazyTree* and *mcc* in more detail. We mentioned earlier that the alphabetic factor can be reduced for *mcc* (and *ukk*) by using a more sophisticated data structure for the tree. By representing subtree lists as binary search trees, the factor for accessing a subtree reduces to $\log_2 k$. Now the expected execution time ratio is

$$\frac{lazyTree}{mcc} = \frac{n \log_k n}{n \log_2 k} = \frac{\log_2 n}{(\log_2 k)^2}.$$

For $n = 100.000$ and $k = 20$ this ratio is 0.89. The $(\log_2 k)^2$-term in the dominator means that the imperative *lazyTree* is hard to beat with a more sophisticated tree data structure, unless the alphabet is very small, n is *very* large, or the data are far from random.

Another interesting observation concerned the relationship between *ukk* and *mcc*. We noted that on all kinds and lengths of data, and for all alphabet sizes, *mcc* was ahead of *ukk* by a constant percentage of execution time. This is remarkable, since the two algorithms are based on rather different ideas. It turned out that they are much more closely related than one would expect. This is explicated in [GK94a].

5.5 Locality Effects

When we benchmark our programs on the abstraction level of asymptotic analysis, e.g. by inserting counters for characteristic steps, our runtime statistics precisely reproduce the theoretic results. *ukk* and *mcc* show perfectly linear graphs of their step counts, while the curve of lazyTree bends upward slightly. However, a close look at our "real time" measurements in Figures 14–17 suggests the opposite:

Anomaly A *ukk* and *mcc* look slightly superlinear, even the worse for increasing k.

Anomaly B On the contrary, *lazyTree* looks closer to linear, and even the better for increasing k.

It is well known [DTM94] that on today's pipelined processors with multi-level caching, the performance of the memory subsystem can significantly affect a program's execution time. The size ratio between the resident page set and the on-chip cache determines the chance of cache hits; as this ratio increases, more frequent cache misses interrupt the pipelined execution and slow down the program. With algorithms like the ones studied here, both running time (in terms of instructions executed) and storage requirements grow linearly with n, and so the time integral over the resident page set size is in $\mathcal{O}(n^2)$. This can in fact be measured and is shown in Figures 18 and 19. Like a magnifying lense, these data show the non-linear contributions of the memory subsystem to the running time of our programs.

The exact way in which the resident page set is related to n is difficult to describe analytically. The operating system determines which percentage of its address space a process needs to execute efficiently. In our case, the size of the data structure is the same for all variants, and the differences in the sizes of the resident page set are solely determined by the locality properties of the algorithms.

5.5.1 Locality of lazyTree

lazyTree has optimal locality on the tree data structure. Once a subtree is completed, it is not accessed again. In principle, more than a "current path" in the tree need not be in memory. With respect to text access, *lazyTree* is also very well-behaved: For each subtree, only the corresponding suffix-rests are accessed. At a certain tree level, the number of suffixes considered will be smaller than the number of available cache entries. As these suffixes are read sequentially, practically no further cache misses will occur. This point is reached earlier when the branching degree of the tree nodes is higher, since the suffixes split up more quickly. Hence, the locality of *lazyTree* improves for larger values of k. This explains Anomaly B.

5.5.2 Locality of ukk and mcc

ukk and *mcc* have identical locality behaviour. (For the deeper reason, see [GK94a].) Let us consider *ukk*. The active suffix creeps through the text like a caterpillar. At the same time, the corresponding active node swings through the tree like a butterfly. "Older" parts of the tree are re-accessed many times via suffix links. Moreover, reading edge labels means accessing the text at positions anywhere left of the current input. Even testing the presence of an a-edge creates an extra text access, and the number of these tests is directly related to the alphabet size k. This explains Anomaly A.

These observations suggest the following pragmatic approach: Modify *ukk* and *mcc* to construct the position suffix tree $pst(t)$ rather than the compact suffix tree $cst(t)$ (cf. Definition 3.5 and Figure 1). First, this simplifies the algorithms, since canonical reference pairs are no longer needed. All inner edges in $pst(t)$ are labelled by single characters, which are now stored in the tree. This avoids random-like accesses into the text, except when a leaf edge must be split. This variant may be a very attractive solution in practice. However, theory has the last word: For the worst case, the size of the position suffix tree can be $\mathcal{O}(n^2)$.

Figure 8: Adding Suffix Links

```
data STree' alf = Leaf' | Branch' [(Label alf,STree' alf)] (STree' alf) | UndefLink

down::(Eq alf)=>[alf]->STree' alf->STree' alf  -- walk down a path to a branching node
down [] node = node
down (a:w) (Branch' es l)
  = down (drop ulen w) node
    where (ulen,node) = head [(culen-1,node) | ((c:_,culen),node)<-es, a == c]

addLinks::(Eq a)=>STree a->STree' a
addLinks Leaf = Leaf'
addLinks (Branch es)
  = root'
    where root' = Branch' [ (cu,lk (down (take (culen-1) u) root') node) |
                            (cu@(c:u,culen),node)<-es] UndefLink
          lk l Leaf = Leaf'
          lk l (Branch es) = Branch' [(lab,lk (down (take slen s) l) node) |
                                      (lab@(s,slen),node)<-es] l
```

6 LazyTree and Suffix Links

6.1 Computing Suffix Links Purely Functionally

For the sake of a fair comparison between the functional and the imperative algorithms, another word on suffix links seems appropriate. *lazyTree*, *naiveOnline* and *naiveInsertion* do not require suffix links, and do not calculate them. Suffix links are necessary to achieve linear time. However, suffix links are also required for some applications, e.g. for deriving q-gram profiles [Ukk92a] or matching statistics [CL90]. Our comparison would be misleading if one could not add the suffix links to the tree purely functionally in $\mathcal{O}(n)$ time. The function *addLinks*, as shown in Figure 8 does so by a single tree traversal.

With the suffix links, the tree actually becomes a circular structure. *addLinks* uses the technique of "computing with unknowns" [Rea89] and hence mandates a lazy programming language. A related technique applies to eager languages. The implementation of nodes with lists of subtrees again introduces a factor l, where l is the average length of the subtree lists, so the overall efficiency is $\mathcal{O}(n \cdot l)$.

For *lazyTree* the setting of suffix links can be merged with the tree construction phase, at virtually no extra cost. This is impossible with *naiveOnline* and *naiveInsertion*. When updating a tree that includes suffix links, the links from elsewhere would still retain their old values and hence point to obsolete subtrees.

6.2 From lazyTree to lazyDawg by Subtree Sharing

Directed acyclic word graphs (DAWGs) and their variants [BBH+85, Cro88] are finite automata recognizing all subwords of a text. The following two observations show a simple way from lazy suffix tree to DAWG construction. For a node $\overline{w}$ of a suffix tree T, let $T_{\overline{w}}$ be

the subtree of T at node $\overline{w}$, and $\#\overline{w}$ the number of leaves in this subtree.

Lemma 6.1 In $T = ast(t\$)$, consider two nodes $\overline{aw}$ and $\overline{w}$, connected by a suffix link. If $\#\overline{aw} = \#\overline{w}$, then $T_{\overline{aw}}$ is isomorphic to $T_{\overline{w}}$.

Proof Because w is a subword of aw, $T_{\overline{w}}$ represents at least the suffixes represented by $T_{\overline{aw}}$. Since each suffix corresponds to a leaf (due to the presence of the sentinel \$), $T_{\overline{w}}$ cannot represent any further suffixes when the numbers of leaves are equal. Representing the same words, the two atomic trees are equal. □

Thus, if we have $\#\overline{w}$ available at each node $\overline{w}$, a look along the suffix link of $\overline{aw}$ tells whether $T_{\overline{aw}}$ must be constructed at all, or can be shared with $T_{\overline{w}}$ instead. By such subtree sharing, the tree physically becomes a directed acyclic graph. The next lemma is folklore among the experts, although we have never seen it written.

Lemma 6.2 The atomic suffix tree with subtree sharing is the DAWG of $t\$$.

Proof Straightforward from the definition of the DAWG via end-position equivalences (see [BBH+85]). □

We derive a DAWG construction *lazyDawg* from (the atomic variant of) *lazyTree* in three steps.

1. For each subtree, we record the length of the suffix list from which it is constructed. For each $T_{\overline{w}}$, this is equivalent to $\#\overline{w}$ (and available even before actually constructing $T_{\overline{w}}$).
2. When constructing $T_{\overline{aw}}$, we first inspect the suffix link to $\overline{w}$. If $\#\overline{aw} = \#\overline{w}$, then simply (a pointer to) $T_{\overline{w}}$ is returned.
3. There is no need to actually record the suffix links in the DAWG. It suffices to supply the node $\overline{w}$ as an extra argument to the call of the construction of $T_{\overline{aw}}$.

The final point means that the DAWG is being traversed while being constructed. A Haskell program for *lazyDawg* is shown in Figure 9. Note the strong resemblance to *lazyTree* (Figure 4).

By the way, if we apply modifications 1-3 to the *lazyTree* variant for the compact suffix tree, we obtain a compact version of the DAWG, i.e. an automaton with multi-character transitions. We see no practical benefit in this compaction, except possibly in the case when t is highly periodic.

7 Conclusion

Suffix tree constructions, in particular their linear-time versions, have sometimes been considered to be very difficult to grasp [Ukk93]. We feel that this can no longer be said. Our abstract derivation of Ukkonen's and McCreight's algorithm based on the terminology of nested suffixes/prefixes maps to the implementations in a very transparent way. While the

Figure 9: *lazyDawg* in Haskell

```
data Dawg alf = D [(alf,Dawg alf)] Int

lazyDawg::(Eq alf)=>[alf]->[alf]->Dawg alf
lazyDawg alpha t
 = root
   where root = D es (length t)
         es = [(a,sTr root (x:xs)) | a<-alpha, x:xs<-[select (suffixes t) a]]
         sTr link ss
           | sn == sn'  = link
           | otherwise  = D es sn
             where D es' sn' = link
                   sn = length ss
                   es = [(a,sTr (down' a link) (x:xs)) | a<-alpha, x:xs<-[select ss a]]

down'::(Eq alf)=>alf->Dawg alf->Dawg alf
down' a (D es sn) = head [node | (b,node)<-es, a == b]
```

naive algorithms also show practical behaviour, it is still worth to invest the extra effort and use the linear-time algorithms instead.

A functional implementation of the linear-time algorithms *ukk* and *mcc* faces two problems:

- the linear-time constraint leaves no time for achieving the local updates by global reconstruction.
- previously set suffix links become obsolete by updating the tree.

However, the suffix tree undergoing a sequence of updates satisfies the condition of single-threadedness. No copy of an intermediate tree is used elsewhere in the program. Thus, recent ideas on monads [Wad92] or mutable data types [Hud93] for functional programming languages incorporating local, in-place updates, apply to this case. A change of the data structure becomes necessary. The tree is represented by mutable arrays, closely resembling our imperative implementation of *ukk* and *mcc*. In this way, we have recently achieved a linear-time, purely functional suffix tree construction in monadic style. Its detailed analysis is outside the scope of this paper, and we refer the reader to [Kur95]. Given mutable arrays, we can also make the functional version of *lazyTree* independent of the alphabet factor.

Recalling our discussion on locality in section 5.5, there is another virtue of *lazyTree*: In contrast to all other methods, it avoids random-like tree accesses during tree construction. Hence, it may be very attractive in distributed memory or database applications.[11]

Let us close with some remarks on methodology. Looking at these algorithms in the two paradigms has been a very rewarding exercise. Our original goal was to translate the known algorithms into the functional paradigm. Only after some experiments with *naiveOnline* and *naiveInsertion* (as the best purely functional approximations to *ukk* and *mcc*) we thought about how one would "naturally" write a suffix tree construction in a functional language.

[11] This latter aspect was pointed out to us by Frank Olken.

The good behaviour of *lazyTree* in our first measurements made us look at it more closely. Finally we observed that (not a functional but) an imperative implementation of *lazyTree* could be made independent of the alphabetic factor. In this sense, functional programming has led to this competitive imperative algorithm.

Raw efficiency is but one criterion for choosing among the alternatives we have studied. Online algorithms are suited when we need to search only for first occurrences of patterns. Depending on the application, only the suffix tree for a prefix of the text may need to be constructed. The partial tree may grow incrementally over a series of queries. When looking for all occurrences, the whole text must be scanned anyway, but only a small portion of the tree is needed. The lazy construction — when implemented in a lazy language — will still achieve incrementality by constructing only that part of the suffix tree that needs to be traversed.

As a side-effect of our study, we gained new insights about the close relationship between the linear-time constructions — Ukkonen's, McCreight's, and also Weiner's, which has been considered sort of a mystery for 20 years [GK94a]. The clarification of their relationship may be a purely academic question, but certainly an intriguing one.

8 Acknowledgements

Marc Rehmsmeier and Nils Andersen provided helpful comments on earlier versions of this paper. Jens Stoye was a great help with the measurements.

References

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[AIL+88] A. Apostolico, C. Iliopoulos, G.M. Landau, B. Schieber, and U. Vishkin. Parallel Construction of a Suffix Tree. *Algorithmica*, 3:347–365, 1988.

[Apo85] A. Apostolico. The Myriad Virtues of Subword Trees. In *Combinatorial Algorithms on Words*. Springer Verlag, 1985.

[AS92] A. Apostolico and W. Szpankowski. Self-Alignments in Words and Their Applications. *Journal of Algorithms*, 13:446–467, 1992.

[Aug93] L. Augustsson. Implementing Haskell Overloading. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture, June 1993*, pages 65–73. ACM Press, New York, NY, 1993.

[BBH+85] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M.T. Chen, and J. Seiferas. The Smallest Automaton Recognizing the Subwords of a Text. *Theoretical Computer Science*, 40:31–55, 1985.

[CL90] W.I. Chang and E.L. Lawler. Approximate String Matching in Sublinear Expected Time. In *Proceedings 31st FOCS*, pages 116–124, 1990.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT-Press, Cambridge, MA, 1990.

[Cro88] M. Crochemore. String Matching with Constraints. In *Proceedings of 1988 International Symposium on Mathematical Foundations of Computer Science*, pages 44–58. Lecture Notes in Computer Science 324, Springer Verlag, 1988.

[CS85] M.T. Chen and J.I. Seiferas. Efficient and Elegant Subword Tree Construction. In *Combinatorial Algorithms on Words*. Springer Verlag, 1985.

[DTM94] A. Diwan, D. Tarditi, and E. Moss. Memory Subsystem Performance of Programs Using Copying Garbage Collection. In *Proceedings of the 21st ACM Symposium on Principles of Programming Languages, Portland, OR, January 1994*, pages 1–14, 1994.

[FHPW92] J.H. Fasel, P. Hudak, S. Peyton-Jones, and P. Wadler. Haskell Special Issue. *ACM SIGPLAN Notices*, 27(5), 1992.

[Gie92] R. Giegerich. Embedding Sequence Analysis in the Functional Programming Paradigm – A Feasibility Study. Report Nr. 8, Technische Fakultät, Universität Bielefeld, 1992.

[GK94a] R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction. Report Nr. 94-03, Technische Fakultät, Universität Bielefeld, 1994.

[GK94b] R. Giegerich and S. Kurtz. Suffix Trees in the Functional Programming Paradigm. In *Proceedings of the European Symposium on Programming (ESOP'94)*, pages 225–240. Lecture Notes in Computer Science 788, Springer Verlag, 1994.

[HL93] P.H. Hartel and K.G. Langendoen. Benchmarking Implementations of Functional Languages. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture, June 1993*, pages 341–349. ACM Press, New York, NY, 1993.

[Hud93] P. Hudak. Mutable Abstract Data Types or How to Have Your State and Munge It Too. Report, YALEU/DCS/RR-914, Yale University, Dep. of Computer Science, 1993.

[KBG87] M. Kempf, R. Bayer, and U. Güntzer. Time Optimal Left To Right Construction of Position Trees. *Acta Informatica*, 24:461–474, 1987.

[Kur95] S. Kurtz. *Fundamental Algorithms for a Declarative Pattern Matching System.* Dissertation (in preparation), Technische Fakultät, Universität Bielefeld, 1995.

[McC76] E.M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[MM90] U. Manber and E.W. Myers. Suffix Arrays: A New Method for On-Line String Searches. In *Proceedings of First ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327, 1990.

[MM93] U. Manber and E.W. Myers. Suffix Arrays: A New Method for On-Line String Searches (see also [MM90]). *SIAM Journal on Computing*, 22(5):935–948, 1993.

[MR80] M.E. Majster and A. Reiser. Efficient On-line Construction and Correction of Position Trees. *SIAM Journal on Computing*, 9(4):785–807, 1980.

[PMN88] C.G. Ponder, P.C. McGeer, and A.P. Ng. Are Applicative Languages Inefficient? *SIGPLAN Notices*, 6:135–139, 1988.

[Rea89] C. Reade. *Elements of Functional Programming.* Addison-Wesley, Reading, MA, 1989.

[Ukk92a] E. Ukkonen. Approximate String-Matching with q-Grams and Maximal Matches. *Theoretical Computer Science*, 92:191–211, 1992.

[Ukk92b] E. Ukkonen. Constructing Suffix Trees On-line in Linear Time. *Algorithms, Software, Architecture. J.v.Leeuwen (Ed.), Inform. Processing 92, Vol. I*, pages 484–492, 1992.

[Ukk93] E. Ukkonen. On-line Construction of Suffix-Trees (Revised Version of [Ukk92b]). to appear in: *Algorithmica,* also available as Report, A-1993-1, Dep. of Computer Science, University of Helsinki, Finland, 1993.

[Wad92] P. Wadler. The Essence of Functional Programming. In *Proceedings of the 19th Annual ACM SIGPLAN-SIGACT Symposium of Programming Languages*, pages 1–14, 1992.

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

Figure 10: Running Times of the Haskell-Programs (in seconds) for $k = 4$

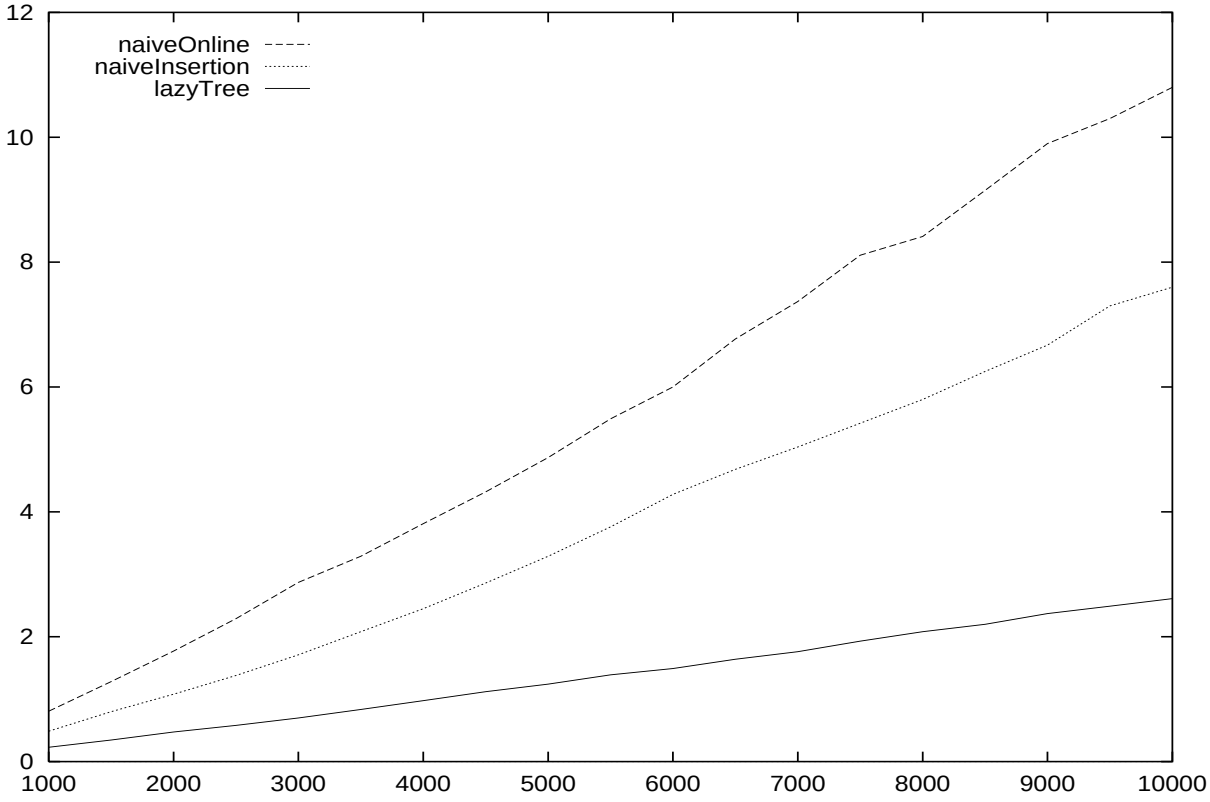


Figure 11: Running Times of the Haskell-Programs (in seconds) for $k = 20$

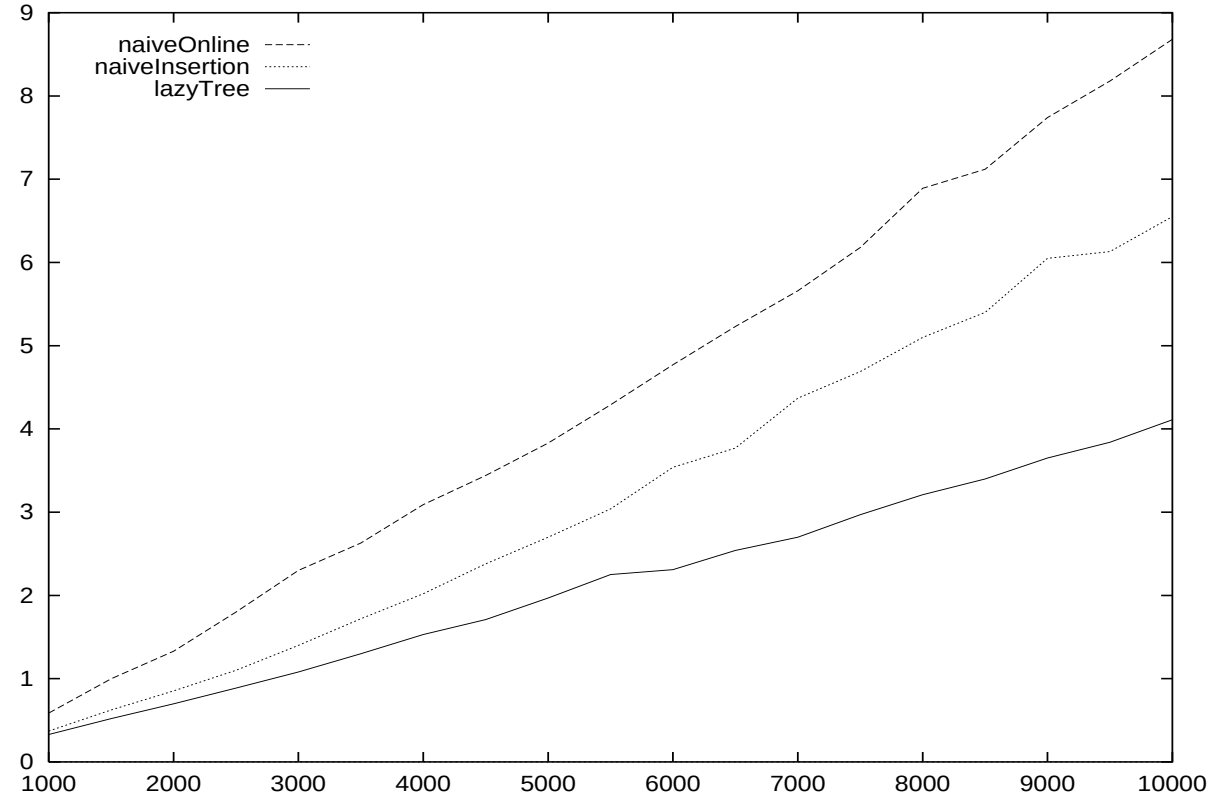


Figure 12: Running Times of the Haskell-Programs (in seconds) for $k = 50$

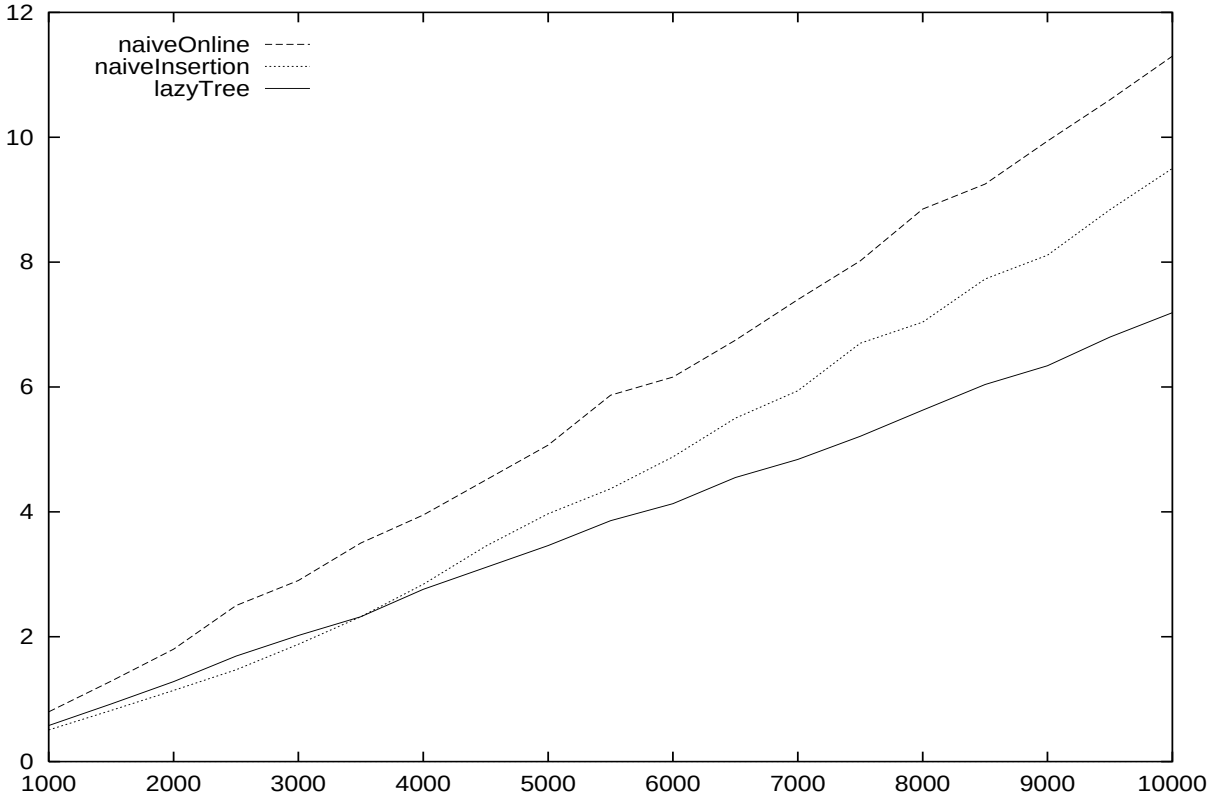


Figure 13: Running Times of the Haskell-Programs (in seconds) for $k = 90$

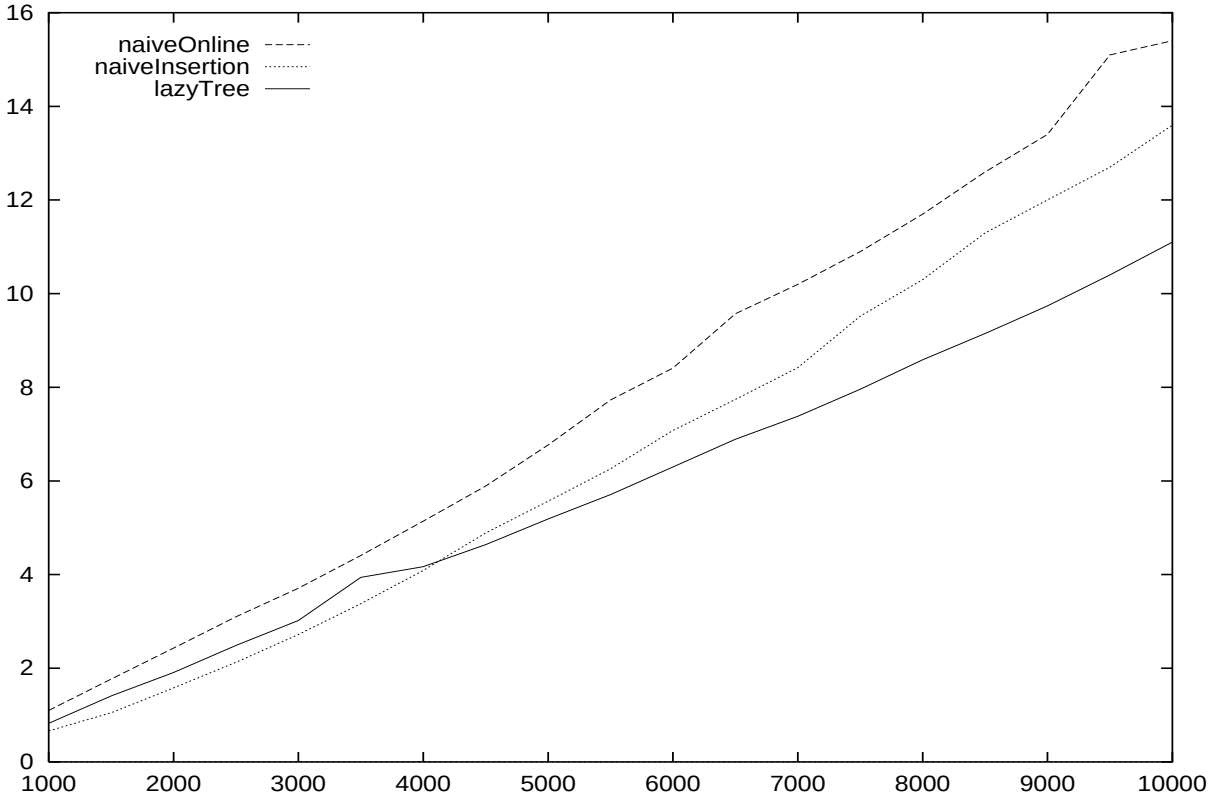


Figure 14: Running Times of the C-Programs (in seconds) for $k = 4$

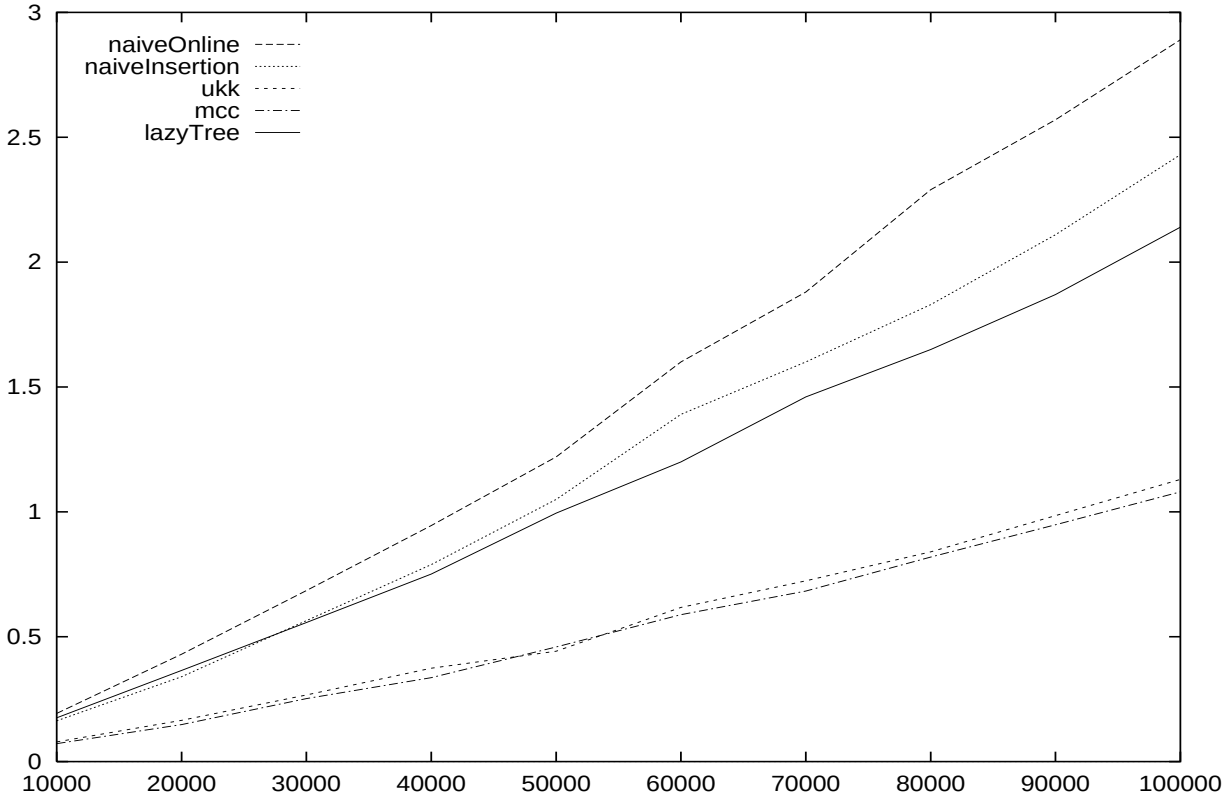


Figure 15: Running Times of the C-Programs (in seconds) for $k = 20$

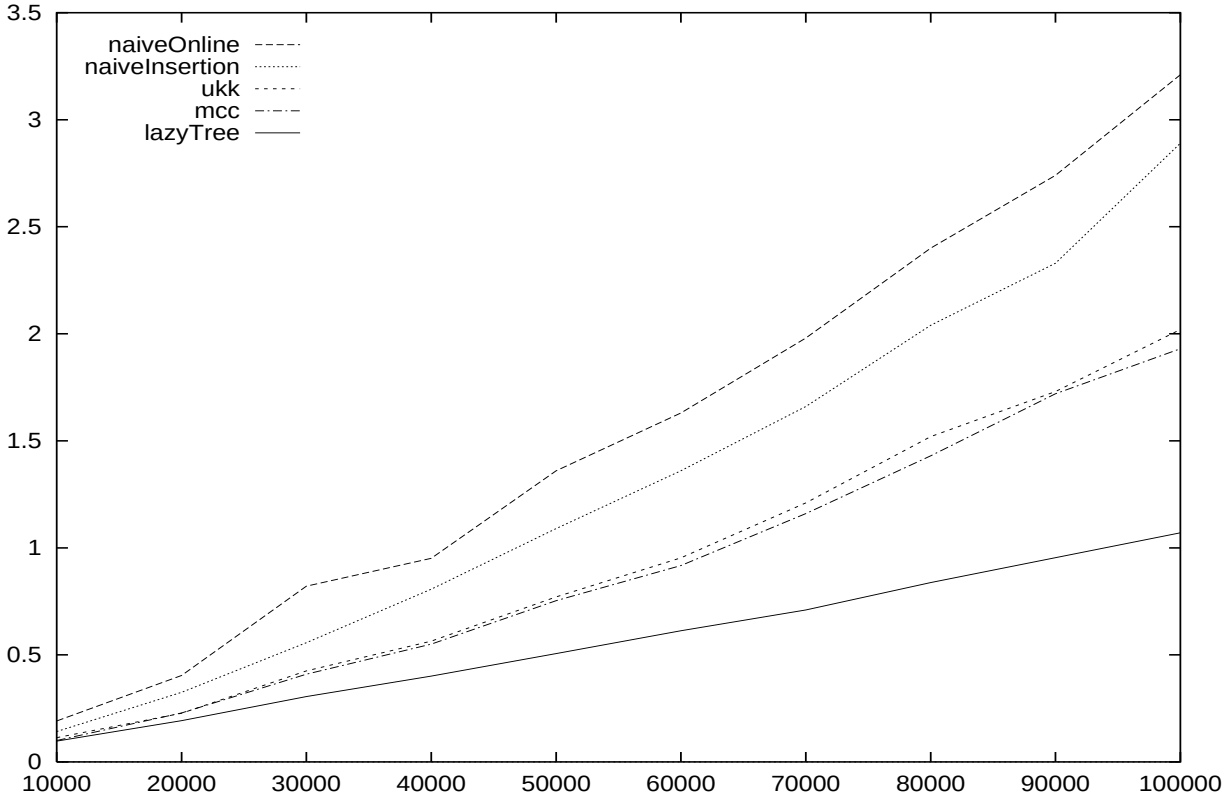


Figure 16: Running Times of the C-Programs (in seconds) for $k = 50$

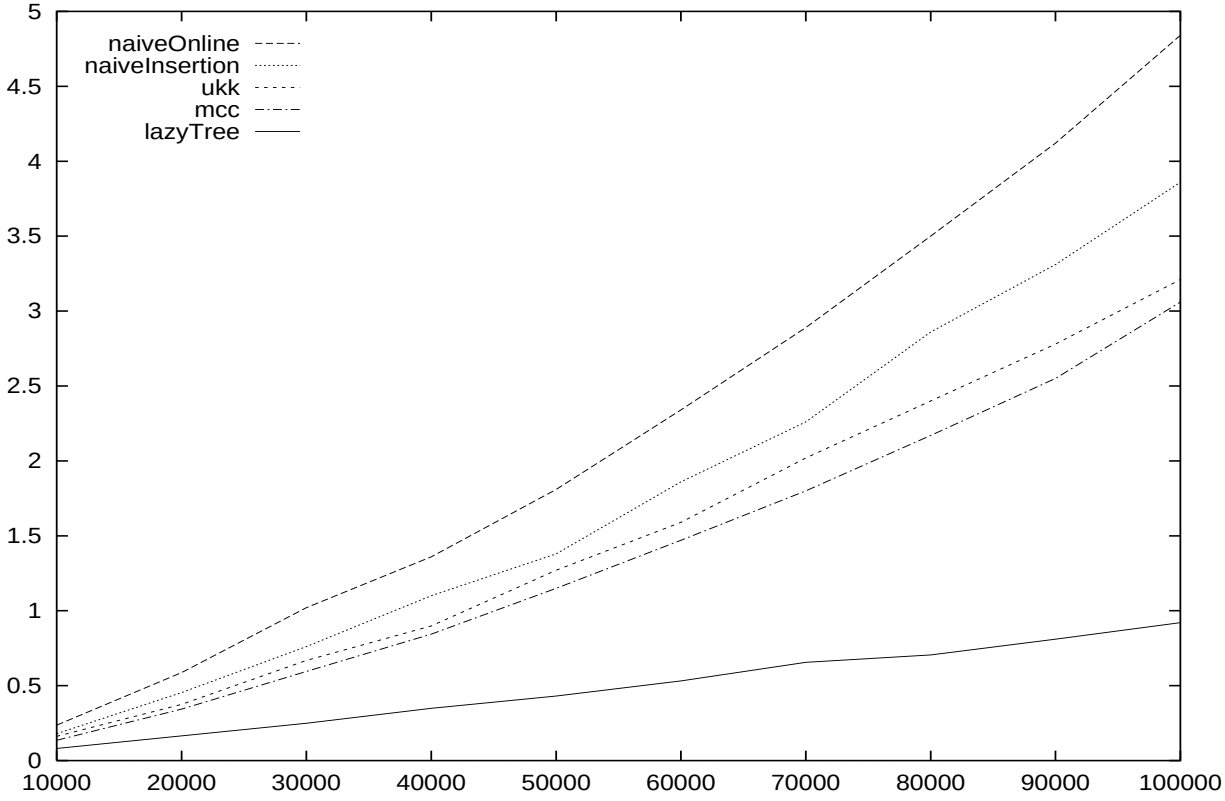


Figure 17: Running Times of the C-Programs (in seconds) for $k = 90$

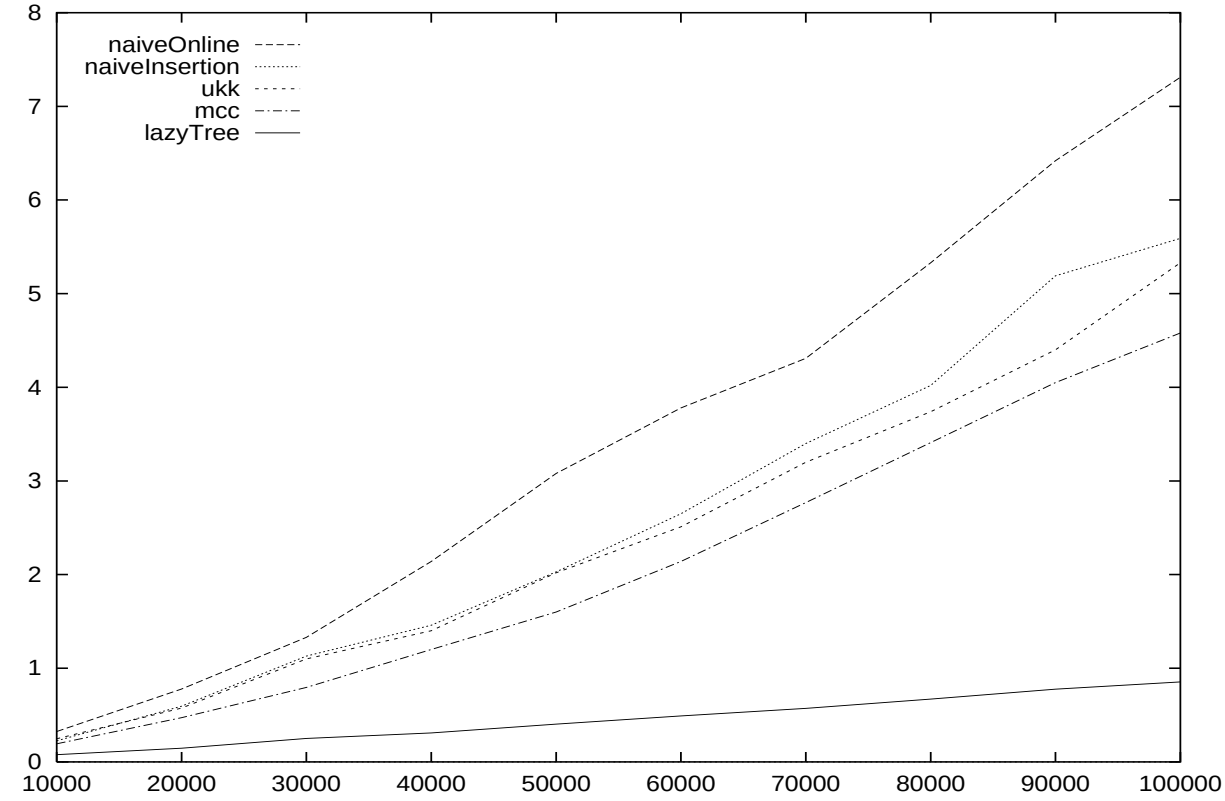


Figure 18: Integral Resident Set Size (in pages) for *lazyTree*

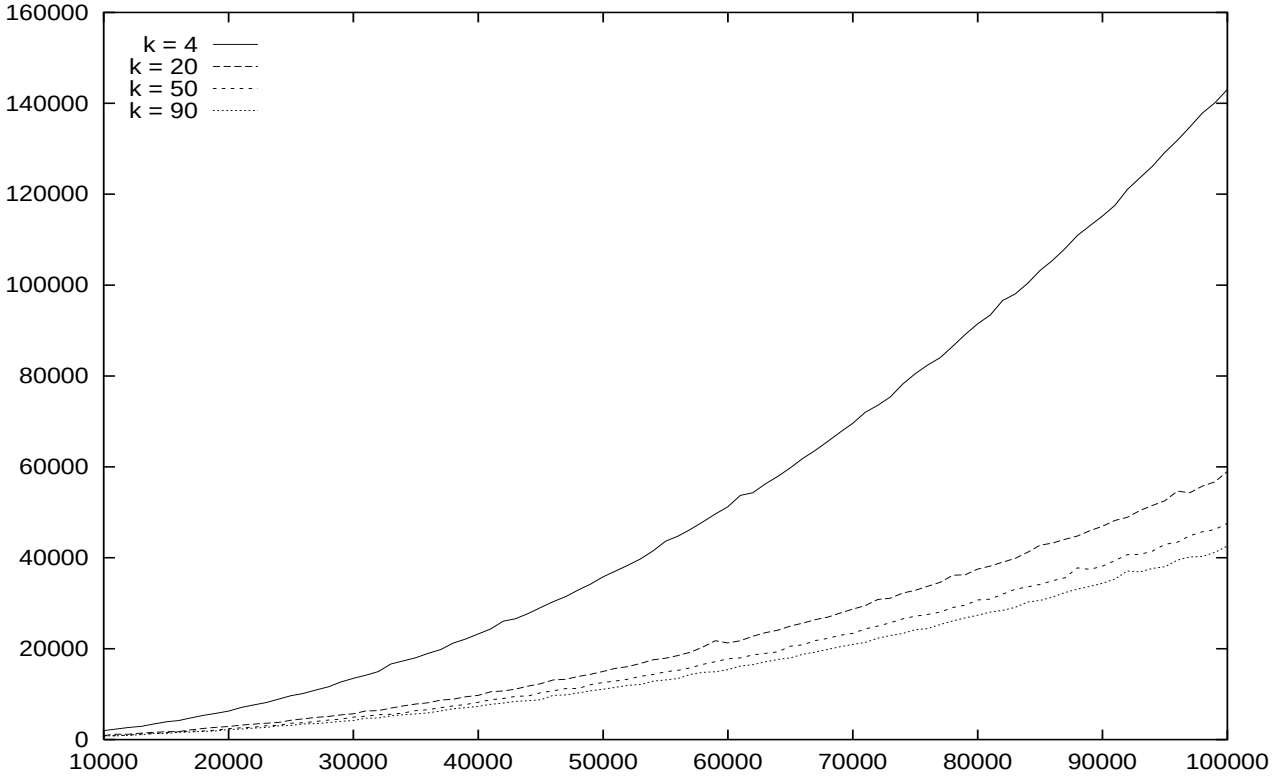


Figure 19: Integral Resident Set Size (in pages) for *mcc*