Suffix Trees in the Functional Programming Paradigm

Robert Giegerich

Stefan Kurtz

Universität Bielefeld, Technische Fakultät, Postfach 100 131

D-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, henceforth called the “text”, is to be searched frequently for occurrences of short sequences, henceforth called “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 have made suffix trees a ubiquitous data structure in a “myriad” of applications [1].

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 [20]. The construction in that paper is given in a somewhat obscure terminology. Later authors [14, 12, 4] have developed more transparent constructions, sometimes tailored to specific additional requirements. The endpoint of the development is currently marked by [18] and [19], presenting a simpler construction in $\mathcal{O}(n)$ space and time, which additionally is *on-line*. It processes the string from left to right, and hence, in this sense it is incremental. What more can one ask for?

Our interest in suffix trees is motivated by our work on a flexible pattern-matching system for biosequence analysis [6]. Besides for locating subwords, suffix trees are useful for finding repetitions and palindromes, deriving q -gram profiles [17], and calculating the so-called matching statistics as a prerequisite for fast approximate matching [3].

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 approach 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. *on-line* 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 constructions¹ 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 observations about the benefits of studying the same algorithm in both the functional and the imperative paradigms.

For reasons of space, we omit all proofs and most of the formal development of the linear-time algorithms. All this can be found in [7].

2 Basic Notions

Let $\mathcal{A}$ be a finite set, called alphabet. The elements of $\mathcal{A}$ are called letters. ε denotes the empty string and $\mathcal{A}^+$ denotes the set of nonempty strings over $\mathcal{A}$. Let t be a string. A string v is called t -word, if and only if $t = uvw$ for some strings u and w . We call a t -word b branching, if and only if there are different letters x and y , such that bx and by are t -words. Let $t = uv$ for some strings u

¹ both are faster than the off-line constructions of [20] and [4], which are also $\mathcal{O}(n)$ (cf. [8]).

and v . Then u is a prefix of t and v is a suffix of t . Let $a \in \mathcal{A}$. A suffix of ta is called nested, if and only if it is a t -word. In other words, a nested suffix of ta has another occurrence as a t -word. (By convention ε is a nested suffix of ε .) Let s be a suffix of t . A prefix p of s is called nested, if and only 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 .

3 The Suffix Tree Family

We give a rather liberal definition of suffix trees, and then three more restricted instances of it, called the naive suffix tree, the position suffix tree, and the linear suffix tree. We mention 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 its simpler construction or better speed of traversal.

An $\mathcal{A}^+$ -tree is a rooted tree with edge labels from $\mathcal{A}^+$. For each $a \in \mathcal{A}$, a node has at most one a -edge $k \xrightarrow{a} k'$. 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 $path(k) = w$.² 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. $\square$

Definition 3.2 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 naive suffix tree for t and denoted by $nst(t)$.
2. If for all $\overline{w} \xrightarrow{a} \overline{v} \in E$ 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 linear suffix tree for t and denoted by $lst(t)$. $\square$

$lst(t)$ 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 . 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 severe paging problems.

² This is a most elegant, but also deceptive convention. It is easy to express relationships between tree nodes, e.g. $\overline{cw}$ and $\overline{w}$, that are quite unrelated in the tree structure.

Fig. 1. Different suffix trees for the string *agcgacgag*.

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.

4.1 The Lazy Suffix Tree Construction

We call a suffix tree construction 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.

Aside from matters of efficiency, which are further discussed in section 4.4, our lazy construction is probably the simplest suffix tree construction that can be given.

We start with a graphical explanation: Write down the *root* with a sorted list l of all non-nested 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 .

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

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 character, and then choose a common prefix of each group for the edge label. This construction is reflected literally in the functional implementation shown below.³

```

sTree * ::= Leaf | Branch [( [*], sTree *)]
edgeFunction * == [( [*]] -> ( [*], [( [*]] )

lazyTree :: (edgeFunction *) -> [*] -> [*] -> sTree *
lazyTree edge alpha t
  = sTr (suffixes t)
  where sTr [] = Leaf
        sTr ss = Branch [(a:p, sTr (y:ys)) | a <- alpha; (p,y:ys) <- [sel a
                                where sel a = edge [ys | y:ys <- ss; y = a]

lazy_pst :: [*] -> [*] -> sTree *
lazy_pst = lazyTree edge_pst

edge_pst :: edgeFunction *
edge_pst [s] = (s, [( [])])
edge_pst ss = ([], ss)

suffixes :: [*] -> [( [*]]      || returns all non-empty suffixes
suffixes [] = []
suffixes (x:xs) = (x:xs):suffixes xs

```

By supplying different edge functions, we construct $nst(t)$ and $lst(t)$ in the same way: To construct $nst(t)$ take an edge function $edge_nst$ that labels *all* edges by a single letter. To construct $lst(t)$ take an edge function $edge_lst$ that chooses for each edge the longest common prefix of its suffix list, ignoring nested suffixes. An implementation of $edge_nst$ and $edge_lst$ can be found in [7].

³ For presentation we use the lazy functional language *Miranda*TM [16].

4.2 Ukkonen's on-line Suffix Tree Construction, Functional Version

In this section, we review Ukkonen's linear-time *on-line* suffix tree construction. The differences in our treatment compared to [18] are the following:

- While Ukkonen derives his construction in an operational style using the naive 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 [15].

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

If $\bar{s}$ is a leaf in $lst(t)$, then a leaf-edge $\bar{b} \xrightarrow{u} \bar{s}$ of $lst(t)$ gives rise to a leaf-edge $\bar{b} \xrightarrow{ua} \bar{s}a$ in $lst(ta)$. This observation led Ukkonen to the idea of representing such leaf-edges by "open" edges of the form $\bar{b} \xrightarrow{(i,\infty)} \bar{L}$, where $\bar{L}$ 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 *on-line* 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 $\bar{b} \xrightarrow{(l,r)} \bar{v}$, where the index pair (l,r) represents the edge label $t_l \dots t_{\min\{length,r\}}$.

Note that the edge label $t_l \dots t_r$ 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 *on-line* 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 $\bar{s}$ is a leaf. Hence we only have to consider the case that $\bar{s}$ is not a leaf in $lst(t)$, or equivalently s is a nested suffix of t .

Definition 4.1 A suffix sa of ta is called relevant, if and only if s is a nested suffix of t and sa is not a t -word. $\square$

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

- (*) Insert all relevant suffixes sa of ta into $lst(t)$.

Before we describe how to insert a relevant suffix of ta into $lst(t)$, 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 .⁴ $\square$

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 $\downarrow$ and the active suffix is printed in bold face.

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

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 $|\alpha(t)a| \geq |sa| > |\alpha(ta)|$. $\square$

By Lemma 4.4 we know that the relevant suffixes of ta are “between” $\alpha(t)a$ and $\alpha(ta)$. Furthermore it is easy to show that $\alpha(ta)$ is a suffix of $\alpha(t)a$ [7]. Hence $\alpha(ta)$ 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 $lst(t)$ 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.

Definition 4.5 Let E be an $\mathcal{A}^+$ -tree and s be a string that occurs in E . We call $(\bar{b}, u)$ reference pair of s with respect to E , if $\bar{b}$ is the *root* or a branching node in E and $s = bu$. If b is the longest such prefix of s , then $(\bar{b}, u)$ is called canonical reference pair of s with respect to E . In such a case we write $\hat{s} = (\bar{b}, u)$. $\square$

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

⁴ The canonical reference pair (see Definition 4.5) of an active suffix corresponds to the notion “active point” introduced in [18].

1. If $\bar{s}$ is a node in $lst(t)$ then $u = \varepsilon$ and $\bar{s} = \bar{b}$ has no a -edge, since otherwise sa would be a t -word. Thus we only add a new open a -edge $\bar{b} \xrightarrow{(i,\infty)} \bar{L}$, where $i = length = |ta|$.
2. If $\bar{s}$ is not a node in $lst(t)$, then $u = cw$ for some letter $c \neq a$ and some string w . Let $\bar{b} \xrightarrow{(l,r)} \bar{v}$ be a c -edge in E and let $k = l + |w|$. Then we introduce $\bar{s}$ by splitting $\bar{b} \xrightarrow{(l,r)} \bar{v}$ into $\bar{b} \xrightarrow{(l,k)} \bar{s} \xrightarrow{(k+1,r)} \bar{v}$ and add a new open a -edge $\bar{s} \xrightarrow{(i,\infty)} \bar{L}$, where $i = length = |ta|$.

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

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

Lemma 4.6 *update*($lst(t)$, $\alpha(t)a$) returns the pair ($lst(ta)$, $\alpha(ta)$). $\square$

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 $\bar{b} \xrightarrow{(l,r)} \bar{v}$ to introduce $\bar{s}$, if necessary. Both is trivial once we have computed the canonical reference pair $(\bar{b}, u)$ for s : If $u = \varepsilon$ then sa is a t -word, if and only if $\bar{b}$ has an a -edge. If $u = cw$ for a letter c and a string w , then there is a c -edge $\bar{b} \xrightarrow{(l,r)} \bar{v}$ and sa is a t -word, if and only if $t_{l+|u|} = a$.

The easiest way to determine $(\bar{b}, u)$ 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 $\bar{s}$, the tree may be de- and reconstructed in $\mathcal{O}(1)$ for each node visited. Thus the local update is turned into a global one with no effect on asymptotic efficiency. An implementation of *naiveOnline* is given below.

```

isTword :: [*] -> * -> (sTree *) -> bool    || if s occurs in st then
isTword [] a (Branch ts)                    || isTword s a st iff sa occurs in st
  = [(c:w,st) | (c:w,st) <- ts; c = a] ~= []
isTword (x:xs) a (Branch ts)
  = w!#xs = a,                               if #xs < #w
  = isTword (drop (#w) xs) a st, otherwise
    where (c:w,st) = hd [(c:w,st) | (c:w,st) <- ts; c = x]

update :: (sTree *, [*]) -> [*] -> (sTree *, [*]) || literally from above
update (st,s) (a:as)
  = (st,s++[a]),                               if isTword s a st
  = (insRelSuff [] (a:as) st, []),             if s = []
  = update (insRelSuff s (a:as) st, drop 1 s) (a:as), otherwise

```



```

insRelSuff::[*]->[*]->(sTree *)->(sTree *)  || insert relevant suffix
insRelSuff [] (a:as) (Branch ts)
  = Branch (g ts)
    where g [] = [(a:as,Leaf)]
          g ((c:w,st):ts') = (c:w,st):g ts',          if c < a
                              = (a:as,Leaf):(c:w,st):ts', otherwise
insRelSuff (x:xs) (a:as) (Branch ts)
  = Branch (g ts)
    where g ((c:w,st):ts')
      = (c:w,st):g ts',          if c ~ = x
      = (x:xs,Branch ts''):ts', if #xs < #w
      = (c:w,insRelSuff (drop (#w) xs) (a:as) st):ts', otherwise
        where z:zs = drop (#xs) w
              ts'' = [(a:as,Leaf),(z:zs,st)], if a < z
                   = [(z:zs,st),(a:as,Leaf)], otherwise

naiveOnline::[*]->(sTree *,[*])
naiveOnline t = foldl update (Branch [],[]) (suffixes t)

```

4.3 McCreight's Suffix Tree Construction, Functional Version

In this section (and in section 5.3) we consider a string $t = t_1 \dots t_n$, $n \geq 2$ 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 [14] is to compute $lst(t)$ 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 $(\bar{h}, q)$ of $head_t(s)$ and the starting position j of $tail_t(s)$ and returns $T_t(s) = T_t(as) \sqcup ((\bar{h}, q), j)$. The easiest way to determine $(\bar{h}, q)$ 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*:

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

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

$$\begin{aligned}
 insertSuffix(E, i) &= E \sqcup ((\bar{h}, q), j) \\
 &\quad \text{where } ((\bar{h}, q), j) = scan(E, root, i)
 \end{aligned}$$

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.

```

insertSuffix::(sTree *)->[*]->sTree *
insertSuffix (Branch ts) (x:xs)
  = Branch (g ts)
    where g [] = [(x:xs,Leaf)]
          g ((a:w,st):r)
            = (a:w,st):g r,           if x > a
            = (x:xs,Leaf):(a:w,st):r, if x < a
            = h bs cs:r,              otherwise
          where (as,bs,cs) = lcp w xs
                h bs [] = (a:w,st)
                h [] (c:cs) = (a:w,insertSuffix st (c:cs))
                h (b:bs) (c:cs)
                  = (a:as,Branch [(c:cs,Leaf),(b:bs,st)]), if c < b
                  = (a:as,Branch [(b:bs,st),(c:cs,Leaf)]), otherwise

lcp::[*]->[*]->([*],[*],[*]) || longest common prefix plus rest suffixes
lcp [] ys = ([],[],ys)
lcp xs [] = ([],xs,[])
lcp (x:xs) (y:ys) = (x:as,bs,cs), if x = y
                  = ([],x:xs,y:ys), otherwise
                  where (as,bs,cs) = lcp xs ys

naiveInsertion::[*]->sTree *
naiveInsertion t = foldl insertSuffix (Branch []) (suffixes t)

```

4.4 Asymptotic and Empirical Efficiency of the Functional Algorithms

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 [2]. 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.

The asymptotic efficiency of *lazyTree* is determined by considering the number of characters read from all suffixes, and the number of operations per character 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 character. Since the functional *lazyTree* uses iteration over $\mathcal{A}$ to group suffixes according to their first character, each character 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 [2]. 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.

We present some empirical results with implementations of the algorithms derived in the previous sections. All algorithms were measured on random texts (Bernoulli-distribution) over alphabets with various sizes ($k = 4, 20, 50, 90$), running on a *SPARCstation* 10/40 with 32 MB. For reasons of space, we include only diagrams for $k = 50$, a sort of compromise between the amino acid alphabet ($k = 20$) and what we find in human readable ASCII files ($k \approx 90$). Measurements were done with the unix tool *rusage* and averaged over 10 runs.

Functional versions of *lazyTree*, *naiveOnline* and *naiveInsertion* were implemented in *Haskell* [5]. In contrast to *Miranda*, *Haskell* provides arrays and its compiler uses more advanced compilation techniques.⁵ We considered the following variants:

- *lazyTreeS*, *naiveOnlineS* and *naiveInsertionS*, where we represented edge labels by explicit strings (as lists of characters),
- *lazyTreeI*, *naiveOnlineI* and *naiveInsertionI*, where we represented edge labels by index pairs into the global text array, in analogy to the implementations in C.

The empirical running time of *naiveOnlineS* in fact seems to be quadratic. It is not shown in the diagram below, in order to keep it readable. We attribute the poor behaviour of *naiveOnlineS* to the fact that the algorithm repeatedly appends characters to the end of the active suffix represented as a list.

From Figure 3 and related figures in [7] we obtain the following results:

- all algorithms (except *naiveOnlineS*) show close to linear behaviour.
- independent of the alphabet size, both *naiveOnlineI* and *naiveInsertionI* are slower than all the other algorithms (by a factor between 2 and 4), with *naiveInsertionI* always being the faster of the two.
- *lazyTreeS* is somewhat faster than *lazyTreeI*, with the advantage decreasing for the larger alphabets.
- *naiveInsertionS* is almost as fast as *lazyTreeS* for $k = 4$, and just beats the lazy algorithms for $k = 20, 50, 90$.

Comparative measurements of functional programs were only done up to $n = 10.000$, since some of the slower programs ran into stack size problems somewhere above. The most space-efficient program is *lazyTreeS*. However, we have not yet tuned our functional implementations with respect to space efficiency.

⁵ We used the Chalmers *Haskell* compiler. Note that there may be *Haskell* compilers that produce better code [10].

Fig. 3. Running Times of the Haskell-Programs (in seconds) for $k = 50$

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*. For reasons of space, we can give only a sketch of the imperative versions.

5.1 *lazyTree*, Imperative Version

A careful imperative implementation of *lazyTree* is an interesting topic of its own right (see [9]). Our current version retains the basic recursion structure, uses a distribution sort 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 on-line 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 $(\bar{b}, u)$ and implement a function *link* which provides direct access from $(\bar{b}, u)$ 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 $f : B \setminus \{\text{root}\} \longrightarrow B$ by $f(\overline{cw}) = \overline{w}$, if and only if $(\overline{cw}, \overline{w}) \in B \setminus \{\text{root}\} \times B$. An element $(\overline{cw}, \overline{w}) \in f$ is called suffix link [14] and f is called suffix link function for E . $\square$

Using the suffix links to proceed from one relevant suffix to the next, we obtain *ukk*, Ukkonen's *on-line* construction. For the details of this development, see [7].

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 algorithm $(\overline{h}, q)$ (the canonical reference pair of $\text{head}_t(s)$) and j (the starting position of $\text{tail}_t(s)$) must be computed in constant time (averaged over all steps). McCreight's algorithm does this by exploiting the following relationships:⁶

Lemma 5.2 Let $\text{head}_t(as) = aw$ for some string w . Then

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

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. This leads to the linear-time algorithm *mcc* (details in [7]).

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. With our simple tree data structure, the asymptotic efficiency is $\mathcal{O}(kn)$ for worst and expected case.

Our imperative version of *lazyTree* uses a distribution sort for grouping suffixes, avoiding iteration over the characters of the alphabet. Since the number of characters inspected is the same as in the functional implementation, we obtain a worst case of $\mathcal{O}(n^2)$ and an expected case of $\mathcal{O}(n \log n)$. Independence of alphabet size is a property shared by the suffix array algorithm of [13], whereas all other known suffix tree constructions must use more complicated data structures to reduce the alphabet factor.

⁶ McCreight explicitly uses only the second relationship, see Lemma 1 in [14].

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 4 from $n = 10.000$ to $n = 100.000$.

Fig. 4. Running Times of the C-Programs (in seconds) for $k = 50$

From this and other diagrams in [7] 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 (by a factor ≤ 3) 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 $n = 100.000$.

The last point is due⁷ to *lazyTree*'s independence of the alphabet size.

⁷ But only partly, since an earlier version of *lazyTree* did depend on the alphabet size, but already performed very well.

6 Conclusion

The linear-time algorithms *ukk* and *mcc* cannot – today – be implemented in a functional language while retaining their $\mathcal{O}(n)$ efficiency. This is for two reasons:

- 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, a remedy may be on the way. 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 adding mutable data types to functional programming languages [11], and thus incorporating local, in-place updates, apply to this case. A change of the data structure may become necessary, e.g. representing the tree by mutable arrays (which is what we use in our imperative implementation of *ukk* and *mcc*). These techniques are not yet available in today's functional language implementations.

Let us end with some remarks on methodology. Looking at these algorithms in the two paradigms has been a very rewarding exercise. Maybe the most remarkable observations made in our experiments were the following:

- The winner in the imperative league was *lazyTree* – a new construction that was discovered in the functional paradigm, and then transliterated into the imperative world.
- At the same time, *lazyTree* was a good, but not the best competitor in the functional league. Instead, this contest was won by *naiveInsertionS*, an algorithm whose basic idea seems inherently imperative.

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.

Given that *lazyTree* directly derived from an inductive definition, one may wonder why it has not been studied before. We think that this is due to the fact that in the imperative paradigm, $\mathcal{O}(n)$ solutions were known since [20]. Although it has been suggested that simpler $\mathcal{O}(n \log n)$ algorithms could do well in practise, these were usually derived from *ukk* or *mcc* (as we did with *naiveOnline* and *naiveInsertion*). Trying to avoid updates altogether was the first motivation for defining *lazyTree*. Thus, the use of functional programming has led us to this very competitive imperative algorithm.

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. This will be explicated in [8].

References

1. A. Apostolico. The Myriad Virtues of Subword Trees. In *Combinatorial Algorithms on Words*. Springer Verlag, 1985.
2. A. Apostolico and W. Szpankowski. Self-Alignments in Words and Their Applications. *Journal of Algorithms*, 13:446–467, 1992.
3. W.I. Chang and E.L. Lawler. Approximate String Matching in Sublinear Expected Time. In *Proceedings 31st FOCS*, pages 116–124, 1990.
4. M.T. Chen and J.I. Seiferas. Efficient and Elegant Subword Tree Construction. In *Combinatorial Algorithms on Words*. Springer Verlag, 1985.
5. J.H. Fasel, P. Hudak, S. Peyton-Jones, and P. Wadler. Haskell Special Issue. *ACM SIGPLAN Notices*, 27(5), 1992.
6. R. Giegerich. Embedding Sequence Analysis in the Functional Programming Paradigm – A Feasibility Study. Report Nr. 8, Technische Fakultät, Universität Bielefeld, 1992.
7. R. Giegerich and S. Kurtz. A comparison of imperative and purely functional suffix tree constructions. Report, Technische Fakultät, Universität Bielefeld, 1994.
8. R. Giegerich and S. Kurtz. A Unifying View of Linear-Time Suffix Tree Construction. *Submitted*, 1994.
9. R. Giegerich and S. Kurtz. Implementing the Lazy Suffix Tree Construction. In *preparation*, 1994.
10. 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.
11. P. Hudak. Mutable Abstract Datatypes or How to Have Your State and Munge It Too. Report, YALEU/DCS/RR-914, Yale University, Dep. of Computer Science, 1993.
12. M.E. Majster and A. Reiser. Efficient on-line Construction and Correction of Position Trees. *SIAM Journal of Computing*, 9(4):785–807, 1980.
13. 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.
14. E.M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
15. C.G. Ponder, P.C. McGeer, and A.P. Ng. Are Applicative Languages Inefficient? *SIGPLAN Notices*, 6:135–139, 1988.
16. D. Turner. An Overview of Miranda. *SIGPLAN Notices*, December 1986, 1986.
17. E. Ukkonen. Approximate String-matching with q-grams and Maximal Matches. *Theoretical Computer Science*, 92:191–211, 1992.
18. 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.
19. E. Ukkonen. On-line construction of suffix-trees (Revised Version of [18]). Report, A-1993-1, Dep. of Computer Science, University of Helsinki, Finland, 1993.
20. P. Weiner. Linear pattern matching algorithms. In *IEEE 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.