Proc. of 3rd Workshop on Algorithm Engineering
page 30-42, London, UK, July 1999,
Lecture Notes in Computer Science 1668, Springer Verlag

Efficient Implementation of Lazy Suffix Trees

Robert Giegerich[1], Stefan Kurtz[1]*, and Jens Stoye[2]

[1] Technische Fakultät, Universität Bielefeld, Postfach 100 131, D-33501 Bielefeld, Germany.
{robert,kurtz}@techfak.uni-bielefeld.de
[2] German Cancer Research Center (DKFZ), Theoretical Bioinformatics (H0300), Im Neuenheimer Feld 280, D-69120 Heidelberg, Germany.
j.stoye@dkfz-heidelberg.de

Abstract. We present an efficient implementation of a write-only top-down construction for suffix trees. Our implementation is based on a new, space-efficient representation of suffix trees which requires only 12 bytes per input character in the worst case, and 8.5 bytes per input character on average for a collection of files of different type. We show how to efficiently implement the lazy evaluation of suffix trees such that a subtree is evaluated not before it is traversed for the first time. Our experiments show that for the problem of searching many exact patterns in a fixed input string, the lazy top-down construction is often faster and more space efficient than other methods.

1 Introduction

Suffix trees are efficiency boosters in string processing. The suffix tree of a text t is an index structure that can be computed and stored in $O(|t|)$ time and space. Once constructed, it allows to locate any substring w of t in $O(|w|)$ steps, independent of the size of t. This instant access to substrings is most convenient in a "myriad" [2] of situations, and in Gusfield's recent book [9], about 70 pages are devoted to applications of suffix trees.

While suffix trees play a prominent role in algorithmics, their practical use has not been as widespread as one should expect (for example, Skiena [16] has observed that suffix trees are the data structure with the highest need for better implementations). The following pragmatic considerations make them appear less attractive:

- The linear-time constructions by Weiner [18], McCreight [15] and Ukkonen [17] are quite intricate to implement. (See also [7] which reviews these methods and reveals relationships much closer than one would think.)
- Although asymptotically optimal, their poor locality of memory reference [6] causes a significant loss of efficiency on cached processor architectures.

* Partially supported by DFG-grant Ku 1257/1-1.

– Although asymptotically linear, suffix trees have a reputation of being greedy for space. For example, the efficient representation of McCreight [15] requires 28 bytes per input character in the worst case.
– Due to these facts, for many applications, the construction of a suffix tree does not amortize. For example, if a text is to be searched only for a very small number of patterns, then it is usually better to use a fast and simple online method, such as the Boyer-Moore-Horspool algorithm [11], to search the complete text anew for each pattern.

However, these concerns are alleviated by the following recent developments:

– In [6], Giegerich and Kurtz advocate the use of a write-only, top-down construction, referred to here as the *wotd*-algorithm. Although its time efficiency is $O(n \log n)$ in the average and even $O(n^2)$ in the worst case (for a text of length n), it is competitive in practice, due to its simplicity and good locality of memory reference.
– In [12], Kurtz developed a space-efficient representation that allows to compute suffix trees in linear time in 46% less space than previous methods. As a consequence, suffix trees for large texts, e.g. complete genomes, have been proved to be manageable.
– The question about amortizing the cost of suffix tree construction is almost eliminated by incrementally constructing the tree as demanded by its queries. This possibility was already hinted at in [6], where the *wotd*-algorithm was called "lazytree" for this reason.

When implementing the *wotd*-algorithm in a lazy functional programming language, the suffix tree automatically becomes a lazy data structure, but of course, the general overhead of using a lazy language is incurred. In the present paper, we explicate how a lazy and an eager version of the *wotd*-algorithm can efficiently be implemented in an imperative language. Our implementation technique avoids a constant alphabet factor in the running time.[1] It is based on a new space efficient suffix tree representation, which requires only $12n$ bytes of space in the worst case. This is an improvement of $8n$ bytes over the most space efficient previous representation, as developed in [12]. Experimental results show that our implementation technique leads to programs that are superior to previous ones in many situations. For example, when searching $0.1n$ patterns of length between 10 and 20 in a text of length n, the lazy *wotd*-algorithm (*wotdlazy*, for short) is on average almost 35% faster and 30% more space efficient than a linked list implementation of McCreight's [15] linear time suffix tree algorithm. *wotdlazy* is almost 13% faster and 50% more space efficient than a hash table implementation of McCreight's linear time suffix tree algorithm, eight times faster and

[1] The suffix array construction of [13] and the linear time suffix tree construction of [5] also do not have the alphabet factor in their running time. For the linear time suffix tree constructions of [15, 17, 18] the alphabet factor can be avoided by employing hashing techniques, see [15], however, for the cost of using considerably more space, see [12].

10% more space efficient than a program based on suffix arrays [13], and *wotd-lazy* is 99 times faster than the iterated application of the Boyer-Moore-Horspool algorithm [11]. The lazy *wotd*-algorithm makes suffix trees also applicable in contexts where the expected number of queries to the text is small relative to the length of the text, with an almost immeasurable overhead compared to its eager variant *wotdeager* in the opposite case. Beside its usefulness for searching string patterns, *wotdlazy* is interesting for other problems (see the list in [9]), such as exact set matching, the substring problem for a database of patterns, the DNA contamination problem, common substrings of more than two strings, circular string linearization, or computation of the q-word distance of two strings.

Documented source code, test data, and complete results of our experiments are available at http://www.techfak.uni-bielefeld.de/~kurtz/Software/wae99.tar.gz.

2 The *wotd*-Suffix Tree Construction

2.1 Terminology

Let Σ be a finite ordered set of size k, the *alphabet*. Σ^* is the set of all strings over Σ, and ε is the *empty string*. We use Σ^+ to denote the set $\Sigma^* \setminus \{\varepsilon\}$ of non-empty strings. We assume that t is a string over Σ of length $n \geq 1$ and that $\$ \in \Sigma$ is a character not occurring in t. For any $i \in [1, n+1]$, let $s_i = t_i \ldots t_n\$$ denote the ith non-empty suffix of $t\$$. A *Σ^+-tree* T is a finite rooted tree with edge labels from Σ^+. For each $a \in \Sigma$, every node u in T has at most one a-edge $u \xrightarrow{av} w$ for some string v and some node w. An edge leading to a leaf is a *leaf edge*. Let u be a node in T. We denote u by $\overline{w}$ if and only if w is the concatenation of the edge labels on the path from the *root* to u. $\overline{\varepsilon}$ is the *root*. A string s *occurs* in T if and only if T contains a node $\overline{sv}$, for some string v. The *suffix tree* for t, denoted by $ST(t)$, is the Σ^+-tree T with the following properties: (i) each node is either a leaf or a branching node, and (ii) a string w occurs in T if and only if w is a substring of $t\$$. There is a one-to-one correspondence between the non-empty suffixes of $t\$$ and the leaves of $ST(t)$. For each leaf $\overline{s_j}$ we define $\ell(\overline{s_j}) = \{j\}$. For each branching node $\overline{u}$ we define $\ell(\overline{u}) = \{j \mid \overline{u} \xrightarrow{v} \overline{uv}$ is an edge in $ST(t), j \in \ell(\overline{uv})\}$. $\ell(\overline{u})$ is the *leaf set* of $\overline{u}$.

2.2 A Review of the *wotd*-Algorithm

The *wotd*-algorithm adheres to the recursive structure of a suffix tree. The idea is that for each branching node $\overline{u}$ the subtree below $\overline{u}$ is determined by the set of all suffixes of $t\$$ that have u as a prefix. In other words, if we have the set $R(\overline{u}) := \{s \mid us$ is a suffix of $t\$\}$ of *remaining suffixes* available, we can evaluate the node $\overline{u}$. This works as follows: at first $R(\overline{u})$ is divided into groups according to the first character of each suffix. For any character $c \in \Sigma$, let $group(\overline{u}, c) := \{w \in \Sigma^* \mid cw \in R(\overline{u})\}$ be the *c-group of* $R(\overline{u})$. If for some $c \in \Sigma$, $group(\overline{u}, c)$ contains only one string w, then there is a leaf edge labeled cw outgoing from $\overline{u}$. If $group(\overline{u}, c)$ contains at least two strings, then there is an

edge labeled cv leading to a branching node $\overline{ucv}$, where v is the longest common prefix (*lcp*, for short) of all strings in $group(\overline{u}, c)$. The child $\overline{ucv}$ can then be evaluated from the set $R(\overline{ucv}) = \{w \mid vw \in group(\overline{u}, c)\}$ of remaining suffixes.

The *wotd*-algorithm starts by evaluating the *root* from the set $R(root)$ of all suffixes of $t\$$. All nodes of $ST(t)$ can be evaluated recursively from the corresponding set of remaining suffixes in a top-down manner.

Example Consider the input string $t = abab$. The *wotd*-algorithm for t works as follows: At first, the *root* is evaluated from the set $R(root)$ of all non-empty suffixes of the string $t\$$, see the first five columns in Fig. 1. The algorithm recognizes 3 groups of suffixes. The a-group, the b-group, and the \$-group. The a-group and the b-group each contain two suffixes, hence we obtain two unevaluated branching nodes, which are reached by an a-edge and by a b-edge. The \$-group is singleton, so we obtain a leaf reached by an edge labeled \$. To evaluate the unevaluated branching node corresponding to the a-group, one first computes the longest common prefix of the remaining suffixes of that group. This is b in our case. So the a-edge from the *root* is labeled by ab, and the remaining suffixes $ab\$$ and \$ are divided into groups according to their first character. Since this is different, we obtain two singleton groups of suffixes, and thus two leaf edges outgoing from $\overline{ab}$. These leaf edges are labeled by $ab\$$ and \$. The unevaluated branching node corresponding to the b-group is evaluated in a similar way, see Fig. 1.

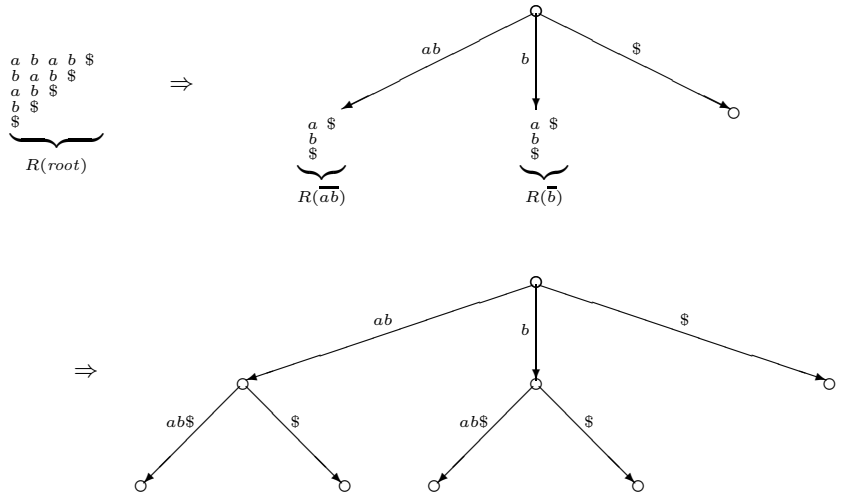

Fig. 1. The write-only top-down construction of $ST(abab)$

2.3 Properties of the *wotd*-Algorithm

The distinctive property of the *wotd*-algorithm is that the construction proceeds top-down. Once a node has been constructed, it needs not be revisited in the construction of other parts of the tree (unlike the linear-time constructions of [5, 15, 17, 18]). As the order of subtree construction is independent otherwise, it may be arranged in a demand-driven fashion, obtaining the lazy implementation detailed in the next section.

The top-down construction has been mentioned several times in the literature [1, 6, 8, 14], but at the first glance, its worst case running time of $O(n^2)$ is disappointing. However, the expected running time is $O(n \log_k n)$ (see e.g. [6]), and experiments in [6] suggest that the *wotd*-algorithm is practically linear for moderate size strings. This can be explained by the good locality behavior: the *wotd*-algorithm has optimal locality on the tree data structure. In principle, more than a "current path" of the tree needs not be in memory. With respect to text access, the *wotd*-algorithm also behaves very well: For each subtree, only the corresponding remaining suffixes 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 the *wotd*-algorithm improves for larger values of k.

Aside from the linear constructions already mentioned, there are $O(n \log n)$ time suffix tree constructions (e.g. [3,8]) which are based on Hopcroft's partitioning technique [10]. While these constructions are faster in terms of worst-case analysis, the subtrees are not constructed independently. Hence they do not share the locality of the *wotd*-algorithm, nor do they allow for a lazy implementation.

3 Implementation Techniques

This section describes how the *wotd*-algorithm can be implemented in an eager language. The "simulation" of lazy evaluation in an eager language is not a very common approach. Unevaluated parts of the data structure have to be represented explicitly, and the traversal of the suffix tree becomes more complicated because it has to be merged with the construction of the tree. We will show, however, that by a careful consideration of efficiency matters, one can end up with a program which is not only more efficient and flexible in special applications, but which performs comparable to the best existing implementations of index-based exact string matching algorithms in general.

We first describe the data structure that stores the suffix tree, and then we show how to implement the lazy and eager evaluation, including the additional data structures.

3.1 The Suffix Tree Data Structure

To implement a suffix tree, we basically have to represent three different items: nodes, edges and edge labels. To describe our representation, we define a total

order $\prec$ on the children of a branching node: Let $\overline{u}$ and $\overline{v}$ be two different nodes in $ST(t)$ which are children of the same branching node. Then $\overline{u} \prec \overline{v}$ if and only if $\min \ell(\overline{u}) < \min \ell(\overline{v})$. Note that leaf sets are never empty and $\ell(\overline{u}) \cap \ell(\overline{v}) = \emptyset$. Hence $\prec$ is well defined.

Let us first consider how to represent the edge labels. Since an edge label v is a substring of $t\$$, it can be represented by a pair of pointers (i, j) into $t' = t\$$, such that $v = t'_i \dots t'_j$. In case the edge is a leaf edge, we have $j = n+1$, i.e., the right pointer j is redundant. In case the edge leads to a branching node, it also suffices to only store a left pointer, if we choose it appropriately: Let $\overline{u} \xrightarrow{v} \overline{uv}$ be an edge in $ST(t)$. We define $lp(\overline{uv}) := \min \ell(\overline{uv}) + |u|$, the *left pointer of* $\overline{uv}$. Now suppose that $\overline{uv}$ is a branching node and $i = lp(\overline{uv})$. Assume furthermore that $\overline{uvw}$ is the smallest child of $\overline{uv}$ w.r.t. the relation $\prec$. Hence we have $\min \ell(\overline{uv}) = \min \ell(\overline{uvw})$, and thus $lp(\overline{uvw}) = \min \ell(\overline{uvw}) + |uv| = \min \ell(\overline{uv}) + |u| + |v| = lp(\overline{uv}) + |v|$. Now let $r = lp(\overline{uvw})$. Then $v = t_i \dots t_{i+|v|-1} = t_i \dots t_{lp(\overline{uv})+|v|-1} = t_i \dots t_{lp(\overline{uvw})-1} = t_i \dots t_{r-1}$. In other words, to retrieve edge labels in constant time, it suffices to store the left pointer for each node (including the leaves). For each branching node $\overline{u}$ we additionally need constant time access to the child of $\overline{uv}$ with the smallest left pointer. This access is provided by storing a reference *firstchild*$(\overline{u})$ to the first child of $\overline{u}$ w.r.t. $\prec$. The *lp*- and *firstchild*-values are stored in a single integer table T. The values for children of the same node are stored in consecutive positions ordered w.r.t. $\prec$. Thus, only the edges to the first child are stored explicitly. The edges to all other children are implicit. They can be retrieved by scanning consecutive positions in table T.

Any node $\overline{u}$ is referenced by the index in T where $lp(\overline{u})$ is stored. To decode the tree representation, we need two extra bits: A *leaf bit* marks an entry in T corresponding to a leaf, and a *rightmost child bit* marks an entry corresponding to a node which does not have a right brother w.r.t. $\prec$. Fig. 2 shows a table T representing $ST(abab)$.

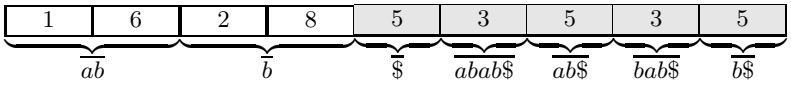

Fig. 2. A table T representing $ST(abab)$ (see Fig. 1). The input string as well as T is indexed from 1. The entries in T corresponding to leaves are shown in grey boxes. The first value for a branching node $\overline{u}$ is $lp(\overline{u})$, the second is *firstchild*$(\overline{u})$. The leaves $\overline{\$}$, $\overline{ab\$}$, and $\overline{b\$}$ are rightmost children

3.2 The Evaluation Process

The *wotd*-algorithm is best viewed as a process evaluating the nodes of the suffix tree, starting at the root and recursively proceeding downwards into the subtrees.

We first describe how an unevaluated node $\overline{u}$ of $ST(t)$ is stored. For the evaluation of $\overline{u}$, we need access to the set $R(\overline{u})$ of remaining suffixes. Therefore we employ a global array *suffixes* which contains pointers to suffixes of $t\$$. For each unevaluated node $\overline{u}$, there is an interval in *suffixes* which stores pointers to all the starting positions in $t\$$ of suffixes in $R(\overline{u})$, ordered by descending suffix-length from left to right. $R(\overline{u})$ is then represented by the two boundaries $\mathit{left}(\overline{u})$ and $\mathit{right}(\overline{u})$ of the corresponding interval in *suffixes*. The boundaries are stored in the two integers reserved in table T for the branching node $\overline{u}$. To distinguish evaluated and unevaluated nodes, we use a third bit, the *unevaluated bit*.

Now we can describe how $\overline{u}$ is evaluated: The edges outgoing from $\overline{u}$ are obtained by a simple counting sort [4], using the first character of each suffix stored in the interval $[\mathit{left}(\overline{u}), \mathit{right}(\overline{u})]$ of the array *suffixes* as the key in the counting phase. Each character c with count greater than zero corresponds to a c-edge outgoing from $\overline{u}$. Moreover, the suffixes in the c-group determine the subtree below that edge. The pointers to the suffixes of the c-group are stored in a subinterval, in descending order of their length. To obtain the complete label of the c-edge, the lcp of all suffixes in the c-group is computed. If the c-group contains just one suffix s, then the lcp is s itself. If the c-group contains more than one suffix, then a simple loop tests for equality of the characters $t_{\mathit{suffixes}[i]+j}$ for $j = 1, 2, \ldots$ and for all start positions i of the suffixes in the c-group. As soon as an inequality is detected, the loop stops and j is the length of the lcp of the c-group.

The children of $\overline{u}$ are stored in table T, one for each non-empty group. A group with count one corresponds to a subinterval of width one. It leads to a leaf, say $\overline{s}$, for which we store $\mathit{lp}(\overline{s})$ in the next available position of table T. $\mathit{lp}(\overline{s})$ is given by the left boundary of the group. A group of size larger than one leads to an unevaluated branching node, say $\overline{v}$, for which we store $\mathit{left}(\overline{v})$ and $\mathit{right}(\overline{v})$ in the next two available positions of table T. In this way, all nodes with the same father $\overline{u}$ are stored in consecutive positions. Moreover, since the suffixes of each interval are in descending order of their length, the children are ordered w.r.t. the relation $\prec$. The values $\mathit{left}(\overline{v})$ and $\mathit{right}(\overline{v})$ are easily obtained from the counts in the counting sort phase, and setting the leaf-bit and the rightmost-child bit is straightforward. To prepare for the (possible) evaluation of $\overline{v}$, the values in the interval $[\mathit{left}(\overline{v}), \mathit{right}(\overline{v})]$ of the array *suffixes* are incremented by the length of the corresponding lcp. Finally, after all successor nodes of $\overline{u}$ are created, the values of $\mathit{left}(\overline{u})$ and $\mathit{right}(\overline{u})$ in T are replaced by the integers $\mathit{lp}(\overline{u}) := \mathit{suffixes}[\mathit{left}(\overline{u})]$ and $\mathit{firstchild}(\overline{u})$, and the unevaluated bit for $\overline{u}$ is deleted.

The nodes of the suffix tree can be evaluated in an arbitrary order respecting the father/child relation. Two strategies are relevant in practice: The *eager* strategy evaluates nodes in a depth-first and left-to-right traversal, as long as there are unevaluated nodes remaining. The program implementing this strategy is called *wotdeager* in the sequel. The *lazy* strategy evaluates a node not before the corresponding subtree is traversed for the first time, for example by

a procedure searching for patterns in the suffix tree. The program implementing this strategy is called *wotdlazy* in the sequel.

3.3 Space Requirement

The suffix tree representation as described in Sect. 3.1 requires $2q + n$ integers, where q is the number of non-root branching nodes. Since $q = n - 1$ in the worst case, this is an improvement of $2n$ integers over the best previous representation, as described in [12]. However, one has to be careful when comparing the $2q + n$ representation of Sect. 3.1 with the results of [12]. The $2q + n$ representation is tailored for the *wotd*-algorithm and requires extra working space of $2.5n$ integers in the worst case:[2] The array *suffixes* contains n integers, and the counting sort requires a buffer of the width of the interval which is to be sorted. In the worst case, the width of this interval is $n - 1$. Moreover, *wotdeager* needs a stack of size up to $n/2$, to hold references to unevaluated nodes.

A careful memory management, however, allows to save space in practice. Note that during eager evaluation, the array *suffixes* is processed from left to right, i.e., it contains a completely processed prefix. Simultaneously, the space requirement for the suffix tree grows. By reclaiming the completely processed prefix of the array *suffixes* for the table T, the extra working space required by *wotdeager* is only little more than one byte per input character, see Table 1. For *wotdlazy*, it is not possible to reclaim unused space of the array *suffixes*, since this is processed in an arbitrary order. As a consequence, *wotdlazy* needs more working space.

4 Experimental Results

For our experiments, we collected a set of 11 files of different sizes and types. We restricted ourselves to 7-bit ASCII files, since the suffix tree application we consider (searching for patterns) does not make sense for binary files. Our collection consists of the following files: We used five files from the Calgary Corpus: *book1*, *book2*, *paper1*, *bib*, *progl*. The former three contain english text, and the latter two formal text (bibliographic items and lisp programs). We added two files (containing english text) from the Canterbury Corpus:[3] *lcet10* and *alice29*. We extracted a section of 500,000 residues from the PIR protein sequence database, denoted by *pir500*. Finally, we added three DNA sequences: *ecoli500* (first 500,000 bases of the ecoli genome), *ychrIII* (chromosome III of the yeast genome), and *vaccg* (complete genome of the vaccinia virus).

[2] Moreover, the *wotd*-algorithm does not run in linear worst case time, in contrast to e.g. McCreight's algorithm [15] which can be used to construct the $5n$ representations of [12] in constant working space. It is not clear to us whether it is possible to construct the $2q + n$ representation of this paper within constant working space, or in linear time. In particular, it is not possible to construct it with McCreight's [15] or with Ukkonen's algorithm [17], see [12].

[3] Both corpora can be obtained from http://corpus.canterbury.ac.nz

All programs we consider are written in C. We used the *ecgs* compiler, release 1.1.2, with optimizing option –O3. The programs were run on a Computer with a 400 MHz AMD K6-II Processor, 128 MB RAM, under Linux. On this computer each integer and each pointer occupies 4 bytes.

In a first experiment we ran three different programs constructing suffix trees: *wotdeager*, *mccl*, and *mcch*. The latter two implement McCreight's suffix tree construction [15]. *mccl* computes the improved linked list representation, and *mcch* computes the improved hash table representation of the suffix tree, as described in [12]. Table 1 shows the running times and the space requirements. We normalized w.r.t. the length of the files. That is, we show the relative time (in seconds) to process 10^6 characters (i.e., $rtime = (10^6 \cdot time)/n$), and the relative space requirement in bytes per input character. For *wotdeager* we show the space requirement for the suffix tree representation (*stspace*), as well as the total space requirement including the working space. *mccl* and *mcch* only require constant extra working space. The last row of Table 1 shows the total length of the files, and the averages of the values of the corresponding columns. In each row a grey box marks the smallest relative time and the smallest relative space requirement, respectively.

			wotdeager			*mccl*		*mcch*	
file	n	k	*rtime*	*stspace*	*space*	*rtime*	*space*	*rtime*	*space*
book1	768771	82	2.82	8.01	9.09	3.55	10.00	2.55	14.90
book2	610856	96	2.60	8.25	9.17	2.90	10.00	2.31	14.53
lcet10	426754	84	2.48	8.25	9.24	2.79	10.00	2.30	14.53
alice29	152089	74	1.97	8.25	9.43	2.43	10.01	2.17	14.54
paper1	53161	95	1.69	8.37	9.50	1.88	10.02	1.88	14.54
bib	111261	81	1.98	8.30	9.17	2.07	9.61	1.89	14.54
progl	71646	87	2.37	9.19	10.42	1.54	10.41	1.95	14.54
ecoli500	500000	4	3.32	9.10	10.46	2.42	12.80	2.84	17.42
ychrIII	315339	4	3.20	9.12	10.70	2.28	12.80	2.70	17.41
vaccg	191737	4	3.70	9.22	11.07	2.14	12.81	2.56	17.18
pir500	500000	20	3.06	7.81	8.61	5.10	10.00	2.58	15.26
	3701614		2.66	8.53	9.71	2.65	10.77	2.34	15.40

Table 1. Time and space requirement for different programs constructing suffix trees

All three programs have similar running times. *wotdeager* and *mcch* show a more stable running time than *mccl*. This may be explained by the fact that the running time of *wotdeager* and *mcch* is independent of the alphabet size. For a thorough explanation of the behavior of *mccl* and *mcch* we refer to [12]. While *wotdeager* does not give us a running time advantage, it is more space efficient than the other programs, using 1.06 and 5.69 bytes per input character less than *mccl* and *mcch*, respectively. Note that the additional working space required for *wotdeager* is on average only 1.18 bytes per input character.

In a second experiment we studied the behavior of different programs searching for many exact patterns in an input string, a scenario which occurs for example in genome-scale sequencing projects, see [9, Sect. 7.15]. For the programs of the previous experiment, and for *wotdlazy*, we implemented search functions. *wotdeager* and *mccl* require $O(km)$ time to search for a pattern string of length m. *mcch* requires $O(m)$ time. Since the pattern search for *wotdlazy* is merged with the evaluation of suffix tree nodes, we cannot give a general statement about the running time of the search. We also considered suffix arrays, using the original program code developed by Manber and Myers [13, page 946]. The suffix array program, referred to by *mamy*, constructs a suffix array in $O(n \log n)$ time. Searching is performed in $O(m + \log n)$ time. The suffix array requires $5n$ bytes of space. For the construction, additionally $4n$ bytes of working space are required. Finally, we also considered the iterated application of an on-line string searching algorithm, our own implementation of the Boyer-Moore-Horspool algorithm [11], referred to by *bmh*. The algorithm takes $O(n + m)$ expected time per search, and uses $O(m)$ working space.

We generated patterns according to the following strategy: For each input string t of length n we randomly sampled ρn substrings $s_1, s_2, \ldots, s_{\rho n}$ of different lengths from t. The proportionality factor ρ was between 0.0001 and 1. The lengths were evenly distributed over the interval $[10, 20]$. For $i \in [1, \rho n]$, the programs were called to search for pattern p_i, where $p_i = s_i$, if i is even, and p_i is the reverse of s_i, otherwise. Reversing a string s_i simulates the case that a pattern search is often unsuccessful. Table 2 shows the relative running times for $\rho = 0.1$. For *wotdlazy* we show the space requirement for the suffix tree after all ρn pattern searches have been performed (*stspace*), and the total space requirement. For *mamy*, *bmh*, and the other three programs the space requirement is independent of ρ. Thus for the space requirement of *wotdeager*, *mccl*, and *mcch* see Table 1. The space requirement of *bmh* is marginal, so it is omitted in Table 2.

Except for the DNA sequences, *wotdlazy* is the fastest and most space efficient program for $\rho = 0.1$. This is due to the fact that the pattern searches only evaluate a part of the suffix tree. Comparing the *stspace* columns of Tables 1 and 2 we can estimate that for $\rho = 0.1$ about 40% of the suffix tree is evaluated. We can also deduce that *wotdeager* performs pattern searches faster than *mcch*, and much faster than *mccl*. This can be explained as follows: searching for patterns means that for each branching node the list of successors is traversed, to find a particular edge. However, in our suffix tree representation, the successors are found in consecutive positions of table T. This means a small number of cache misses, and hence the good performance. It is remarkable that *wotdlazy* is more space efficient and eight times faster than *mamy*. Of course, the space advantage of *wotdlazy* is lost with a larger number of patterns. In particular, for $\rho \geq 0.3$ *mamy* is the most space efficient program. Figs. 3 and 4 give a general overview, how ρ influences the running times. Fig. 3 shows the average relative running time for all programs and different choices of ρ for $\rho \leq 0.005$. Fig. 4 shows the average relative running time for all programs except *bmh* for all values of ρ. We observe that *wotdlazy* is the fastest program for $\rho \leq 0.3$, and *wotdeager* is the

			wotdlazy			*wotdeager*	*mccl*	*mcch*	*mamy*		*bmh*
file	n	k	*rtime*	*stspace*	*space*	*rtime*	*rtime*	*rtime*	*rtime*	*space*	*rtime*
book1	768771	82	3.17	3.14	7.22	3.37	5.70	3.19	20.73	8.04	413.60
book2	610856	96	2.85	3.12	7.22	3.14	5.17	2.96	21.23	8.06	298.25
lcet10	426754	84	2.65	3.07	7.22	3.07	4.52	2.88	20.55	8.07	206.40
alice29	152089	74	2.04	3.13	7.23	2.43	3.62	2.70	17.10	8.14	74.76
paper1	53161	95	1.50	3.23	7.65	2.07	2.82	2.26	9.41	8.68	23.70
bib	111261	81	1.89	3.06	7.23	2.43	3.06	2.43	14.11	8.24	47.28
progl	71646	87	1.81	2.91	7.24	2.79	2.37	2.37	14.52	8.42	32.24
ecoli500	500000	4	3.60	3.71	8.02	3.82	3.36	3.68	24.84	8.52	724.56
ychrIII	315339	4	3.23	3.84	8.02	3.62	3.17	3.49	26.73	8.83	453.39
vaccg	191737	4	2.97	3.81	8.03	3.39	2.87	3.34	23.73	8.34	291.13
pir500	500000	20	2.46	3.55	7.62	3.66	6.34	3.10	29.38	8.06	248.42
	3701614		2.56	3.32	7.52	3.07	3.91	2.94	20.21	8.31	255.79

Table 2. Time and space requirement for searching $0.1n$ exact patterns

fastest program for $\rho \geq 0.4$. *bmh* is faster than *wotdlazy* only for $\rho \leq 0.0003$. Thus the index construction performed by *wotdlazy* already amortizes for a very small number of pattern searches.

We also performed some tests on two larger files (english text) of length 3 MB and 5.6 MB, and we observed the following:

- The relative running time for *wotdeager* slightly increases, i.e. the superlinearity in the complexity becomes visible. As a consequence, *mcch* becomes faster than *wotdeager* (but still uses 50% more space).
- With ρ approaching 1, the slower suffix tree construction of *wotdeager* and *wotdlazy* is compensated for by a faster pattern search procedure, so that there is a running time advantage over *mcch*.

5 Conclusion

We have developed efficient implementations of the write-only top-down suffix tree construction. These construct a representation of the suffix tree, which requires only $12n$ bytes of space in the worst case, plus $10n$ bytes of working space. The space requirement in practice is only $9.71n$ bytes on average for a collection of files of different type. The time and space overhead of the lazy implementation is very small. Our experiments show that for searching many exact patterns in an input string, the lazy algorithm is the most space and time efficient algorithm for a wide range of input values.

Acknowledgements We thank Gene Myers for providing a copy of his suffix array code.

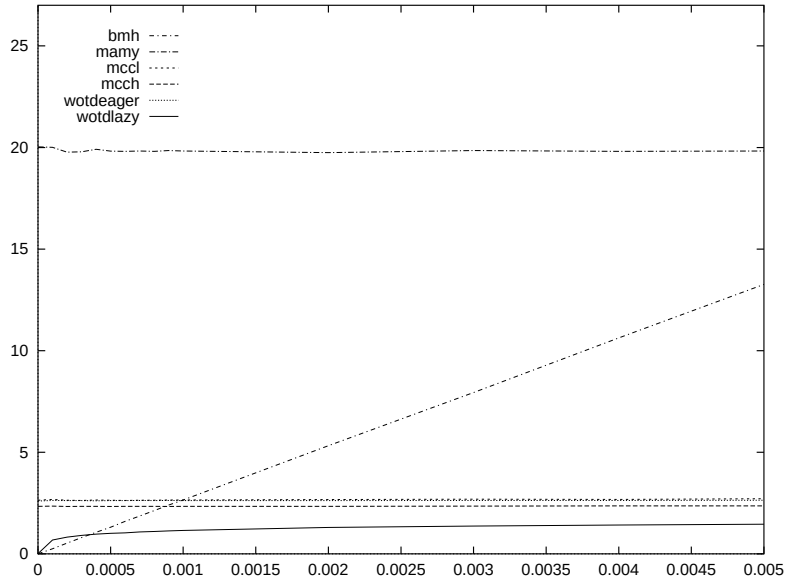

Fig. 3. Average relative running time (in seconds) for different values of $\rho \in [0, 0.005]$

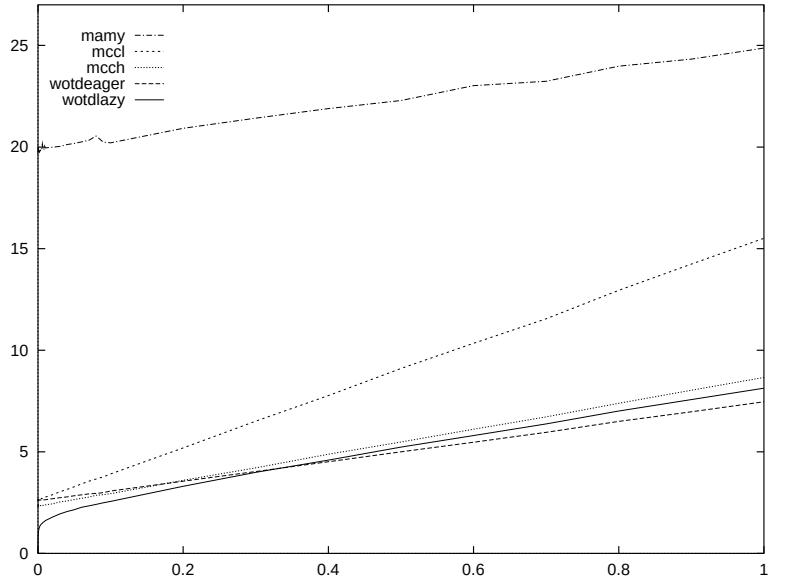

Fig. 4. Average relative running time (in seconds) for different values of $\rho \in [0, 1]$

References

1. A. Andersson and S. Nilsson. Efficient Implementation of Suffix Trees. *Software—Practice and Experience*, 25(2):129–141, 1995.
2. A. Apostolico. The Myriad Virtues of Subword Trees. In *Combinatorial Algorithms on Words*, pages 85–96. Springer Verlag, 1985.
3. A. Apostolico, C. Iliopoulos, G. M. Landau, B. Schieber, and U. Vishkin. Parallel Construction of a Suffix Tree with Applications. *Algorithmica*, 3:347–365, 1988.
4. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.
5. M. Farach. Optimal Suffix Tree Construction with Large Alphabets. In *Proc. of the 38th Annual Symposium on the Foundations of Computer Science (FOCS)*, 1997.
6. R. Giegerich and S. Kurtz. A Comparison of Imperative and Purely Functional Suffix Tree Constructions. *Science of Computer Programming*, 25(2-3):187–218, 1995.
7. R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Constructions. *Algorithmica*, 19:331–353, 1997.
8. D. Gusfield. An "Increment-by-one" Approach to Suffix Arrays and Trees. Report CSE-90-39, Computer Science Division, University of California, Davis, 1990.
9. D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.
10. J. Hopcroft. An $O(n \log n)$ Algorithm for Minimizing States in a Finite Automaton. In *Proceedings of an International Symposium on the Theory of Machines and Computations*, pages 189–196. Academic Press, New York, 1971.
11. R.N. Horspool. Practical Fast Searching in Strings. *Software—Practice and Experience*, 10(6):501–506, 1980.
12. S. Kurtz. Reducing the Space Requirement of Suffix Trees. *Software—Practice and Experience*, 1999. Accepted for publication.
13. U. Manber and E.W. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
14. H.M. Martinez. An Efficient Method for Finding Repeats in Molecular Sequences. *Nucleic Acids Res.*, 11(13):4629–4634, 1983.
15. E.M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
16. S. S. Skiena. Who is Interested in Algorithms and Why? Lessons from the Stony Brook Algorithms Repository. In *Proceedings of the 2nd Workshop on Algorithm Engineering (WAE)*, pages 204–212, 1998.
17. E. Ukkonen. On-line Construction of Suffix-Trees. *Algorithmica*, 14(3), 1995.
18. P. Weiner. Linear Pattern Matching Algorithms. In *Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory*, pages 1–11, The University of Iowa, 1973.