SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2003; **33**:1035–1049 (DOI: 10.1002/spe.535)

Efficient implementation of lazy suffix trees

R. Giegerich[1], S. Kurtz[2] and J. Stoye[1,*,†]

[1]*Faculty of Technology, University of Bielefeld, 33594 Bielefeld, Germany*
[2]*Centre for Bioinformatics, University of Hamburg, Bundesstraße 43, 20146 Hamburg, Germany*

SUMMARY

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 that 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 only when 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. Copyright © 2003 John Wiley & Sons, Ltd.

KEY WORDS: string matching; suffix tree; space-efficient implementation; lazy evaluation

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 any substring w of t to be located in $O(|w|)$ steps, independent of the size of t. This instant access to substrings is most convenient in a 'myriad' [1] of situations, and in Gusfield's book [2], 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 [3] 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:

*Correspondence to: J. Stoye, AG Genominformatik, Faculty of Technology, Universität Bielefeld, 33594 Bielefeld, Germany.
†E-mail: stoye@techfak.uni-bielefeld.de

Contract/grant sponsor: Deutsche Forschungsgemeinschaft; contract/grant number: Ku 1257/1-1

Received 26 November 2001
Revised 4 November 2002
Accepted 8 March 2003

Published online 25 June 2003

- The linear-time constructions by Weiner [4], McCreight [5] and Ukkonen [6] 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 [8] causes a significant loss of efficiency on cached processor architectures.
- Although asymptotically linear, suffix trees have a reputation of being greedy for space. For example, the representation of McCreight [5] 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 [9], to search the complete text anew for each pattern.

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

- In [8], 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 expected case 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 [10], Kurtz developed a space-efficient representation that allows suffix trees to be computed 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; see for example [11].
- The question of 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 [8], 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 explain 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[‡]. 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 [10]. Other recently published suffix tree representations require $17n$ bytes [14] and $21n$ bytes [15]. Theoretical developments in [16,17] describe suffix tree representations that use only $n \log_2 n + O(n)$ bits of space. Another one is the *compressed suffix tree* of [18], which requires only $O(n)$ bits of space. However, these data structures are confined to text searching and, to the best of our knowledge, no implementation exists based on these techniques.

There are other space-efficient data structures for string pattern matching. The *suffix array* of [12] requires $9n$ bytes, including the space for construction. The *level compressed trie* of [19] takes

[‡]The suffix array construction of [12] and the linear time suffix tree construction of [13] also do not have the alphabet factor in their running times. For the linear time suffix tree constructions of [4–6] the alphabet factor can be avoided by employing hashing techniques, see [5]; however, at the cost of using considerably more space, see [10].

about $12n$ bytes. The suffix binary search tree of [20] requires $10n$ bytes. The *suffix cactus* of [21] can be implemented in $10n$ bytes, and the *suffix oracle* [22] in $8n$ bytes. Finally, the *PT*-tree of [23] requires $n \log_2 n + O(n)$ bits.

While some of these data structures require less space than suffix trees, we expect that for a moderate number of pattern searches our lazy construction *wotdlazy* is more space efficient than these methods. Moreover, some of these index structures can only be computed efficiently by first constructing the complete suffix tree.

Experimental results show that our implementation technique leads to programs that are superior to previous ones in many situations. For example, when searching $0.01n$ patterns of length between 10 and 20 in a text of length n, the lazy *wotd*-algorithm (*wotdlazy*, for short) is on average 43% faster and almost 50% more space efficient than a linked list implementation of McCreight's linear time suffix tree algorithm [5]. It is 38% faster and 65% more space efficient than a hash table implementation of McCreight's linear time suffix tree algorithm, ten times faster and 42% more space efficient than a program based on suffix arrays [12], and 40 times faster than the iterated application of the Boyer–Moore–Horspool algorithm [9]. 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 with its eager variant *wotdeager* if the complete tree is built. In addition to its usefulness for searching string patterns, *wotdlazy* is interesting for other problems (see the list in [2]), 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 of the programs *wotdeager* and *wotdlazy* is available at `http://bibiserv.techfak.uni-bielefeld.de/wotd`. A preliminary version of this paper appeared in [24].

THE *WOTD* SUFFIX TREE CONSTRUCTION

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 α in T has at most one a-edge $\alpha \stackrel{av}{\rightarrow} \beta$ for some string v and some node β. An edge leading to a leaf is a *leaf edge*. Let α be a node in T. We denote α by the path that leads us there. Formally, α is denoted by $\overline{w}$ if and only if w is the concatenation of the edge labels on the path from the *root* to α. $\overline{\varepsilon}$ 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 $ST(t)$. We define the *leaf set* $\ell(\alpha)$ of a node α as follows. For a leaf $\overline{s_j}$, $\ell(\overline{s_j}) = \{j\}$. For a branching node $\overline{u}$, $\ell(\overline{u}) = \{j \mid \overline{u} \stackrel{v}{\rightarrow} \overline{uv}$ is an edge in $ST(t)$, $j \in \ell(\overline{uv})\}$.

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 $R(\overline{u}) := \{s \mid us \text{ is a suffix of } t\$\}$ 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.

For example, consider the input string $t = babab$. 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 six columns in Figure 1. The algorithm recognizes three groups of suffixes. The a-group, the b-group, and the \$-group. The a-group contains two and the b-group contains three 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. We obtain two singleton groups of suffixes, and thus two leaf edges outgoing from $\overline{ab}$. These leaf edges are labeled as $ab\$$ and \$. The unevaluated branching node corresponding to the b-group is evaluated recursively in a similar way, see Figure 1.

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 need not be revisited in the construction of other parts of the tree (unlike the linear-time constructions of [4–6,13]). As the order of subtree construction is otherwise independent, 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 [8,19,25,26], but at 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. [8]), and experiments in [8] 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, no more than a 'current path' of the tree need 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

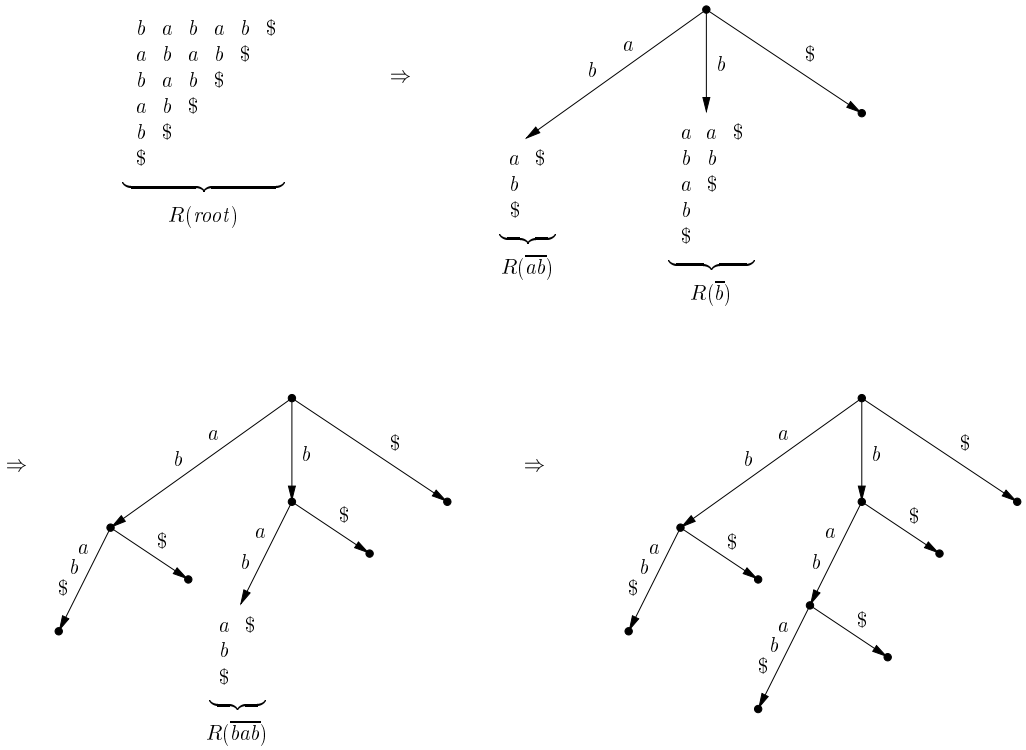

Figure 1. The write-only top-down construction of $ST(babab)$.

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 alphabets.

Aside from the linear constructions already mentioned, there are $O(n \log n)$ time suffix tree constructions (e.g. [26,27]), which are based on Hopcroft's partitioning technique [28]. 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.

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 that is not only

more efficient and flexible in special applications, but the performance of which is 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.

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 $\overline{uv}$ and $\overline{uw}$ be two different nodes in $ST(t)$ that are children of the same branching node $\overline{u}$. Then $\overline{uv} \prec \overline{uw}$ if and only if $\min \ell(\overline{uv}) < \min \ell(\overline{uw})$. Note that leaf sets are never empty and $\ell(\overline{uv}) \cap \ell(\overline{uw}) = \emptyset$. 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 $v = t'_i \ldots t'_j$ (see Figure 2). 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 store only a left pointer, if we choose it appropriately (see Figure 3): 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 $v = t_i \ldots t_{i+|v|-1} = t_i \ldots t_{lp(\overline{uv})+|v|-1} = t_i \ldots t_{lp(\overline{uvw})-1} = t_i \ldots t_{r-1}$. 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 table T where $lp(\overline{u})$ is stored. To decode the suffix 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 that does not have a right brother w.r.t. $\prec$. Figure 4 shows a table T representing $ST(babab)$.

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* that contains pointers to suffixes of $t\$$. For each unevaluated node $\overline{u}$, there is an interval in *suffixes* that stores starting positions 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 *left*$(\overline{u})$ and *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*.

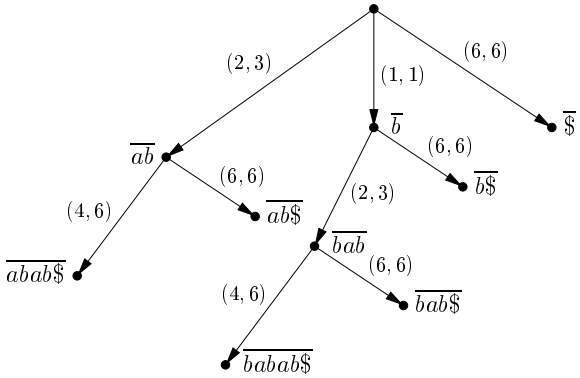

Figure 2. The final suffix tree $ST(babab)$ from Figure 1 with edge labels as pairs (left pointer, right pointer).

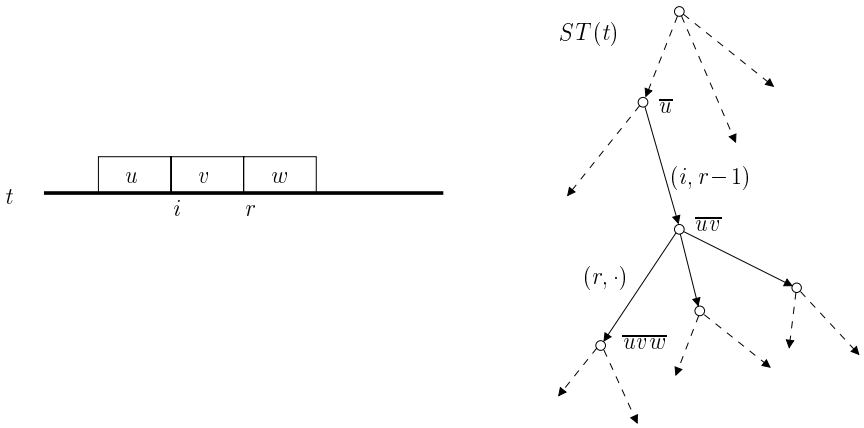

Figure 3. The definition of $lp(\overline{uv})$.

Now we can describe how $\overline{u}$ is evaluated: the edges outgoing from $\overline{u}$ are obtained by a simple counting sort [29], using the first character of each suffix stored in the interval $[left(\overline{u}), right(\overline{u})]$ 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 $t_{suffixes[i]+j}$ for $j = 1, 2, \ldots$ and

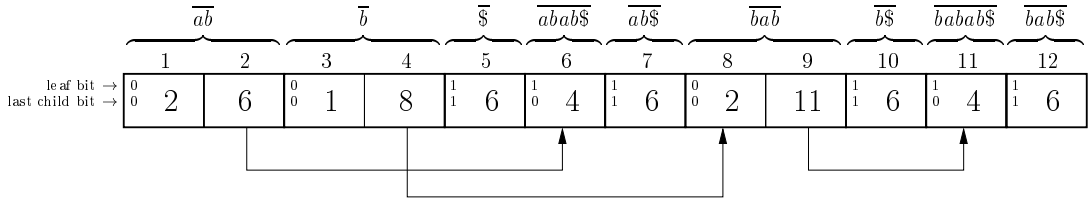

Figure 4. A table T representing $ST(babab)$ (see Figure 2). The input string as well as T are indexed starting with 1. A branching node occupies two table entries, a leaf occupies one table entry. The first value for a branching node $\overline{u}$ is $lp(\overline{u})$, the second is $firstchild(\overline{u})$, indicated also by arrows below the table. The leaf bit and rightmost (last) child bit are indicated in the first table entry of each node.

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 $lp(\overline{s})$ in the next available position of table T. $lp(\overline{s})$ 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 $left(\overline{v})$ and $right(\overline{v})$ in the next two available positions of table T. In this way, all nodes with the same parent $\overline{u}$ 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 $left(\overline{v})$ and $right(\overline{v})$ 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 $[left(\overline{v}), right(\overline{v})]$ 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 $left(\overline{u})$ and $right(\overline{u})$ in T are replaced by the integers $lp(\overline{u}) := suffixes[left(\overline{u})]$ and $firstchild(\overline{u})$, and the unevaluated bit for $\overline{u}$ is deleted.

The nodes of the suffix tree can be evaluated in an arbitrary order respecting the parent/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 only when 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.

Space requirement

The suffix tree representation as described above 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 [10]. However, one has to be careful when comparing the $2q + n$ integers representation above with the results of [10]. The $2q + n$ integers representation

is tailored for the *wotd*-algorithm and requires extra working space of $2.5n$ integers in the worst case[§]: The array *suffixes* contains n integers, and the counting sort requires a buffer of the width of the interval that 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.

Careful memory management, however, allows space to be saved 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 I. 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.

EXPERIMENTAL RESULTS

For our experiments, we collected a set of nine files of different sizes and types. We restricted ourselves to 7-bit ASCII files, since the suffix tree application we consider (searching for string patterns) is not common for binary files. Our collection consists of the following files. We used three files from the Calgary Corpus: *bib*, *book1*, *book2*. The first one contains formal text (bibliographic items), and the latter two contain English text. We added three files (containing English text) from the Canterbury Corpus: *alice29*, *lcet10*, and *plrabn12*. Finally, we added three large files[¶]: *E.coli* (the complete genome of the bacteria *Escherichia coli*), *bible* (the Bible), and *world* (the Project Gutenberg Edition of The World Factbook 1992).

All programs we consider are written in C. We used the *gcc* compiler, version 2.96, with optimizing option –O3. The programs were run on a computer with a 1.666 MHz AMD Athlon XP 2000+ processor, 512 MB RAM, under Linux. On this computer each integer and each pointer occupies 4 bytes. As a consequence, we have 30 bits to store $left(\overline{u})$ or $lp(\overline{u})$ for a branching node or leaf $\overline{u}$. Moreover, we have 31 bits to store $right(\overline{u})$ or $firstchild(\overline{u})$ for a branching node $\overline{u}$. $left(\overline{u})$, $right(\overline{u})$, and $lp(\overline{u})$ are in the range $[0, n]$. $firstchild(\overline{u})$ is in the range $[0, 3n]$. Hence $3n \leq 2^{31} - 1$ must be satisfied, i.e. the maximal length of the input string is 715 827 882.

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 [5]. *mccl* computes the improved linked list representation, and *mcch* computes the improved hash table representation of the suffix tree, as described in [10]. Table I 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*),

[§]Moreover, the *wotd*-algorithm does not run in linear worst case time, in contrast to e.g. McCreight's algorithm [5] which can be used to construct the $5n$ integers representations of [10] 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 [5] or with Ukkonen's algorithm [6], see [10].

[¶]These files, like the two corpora, can be obtained from `http://corpus.canterbury.ac.nz`.

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

			wotdeager			*mccl*		*mcch*	
File	n	k	*rtime*	*stspace*	*space*	*rtime*	*space*	*rtime*	*space*
bib	111 261	81	**0.27**	8.30	**9.17**	0.54	9.61	0.81	14.54
book1	768 771	82	**0.72**	8.01	**9.09**	1.61	10.00	1.20	14.90
book2	610 856	96	**0.64**	8.25	**9.17**	1.18	10.00	1.05	14.53
alice29	152 089	74	**0.33**	8.25	**9.43**	0.72	10.01	0.92	14.54
lcet10	426 754	84	**0.56**	8.25	**9.24**	1.12	10.00	1.01	14.53
plrabn12	481 861	81	**0.62**	7.94	**8.93**	1.39	10.00	1.14	14.91
E.coli	4 638 690	4	1.83	9.14	**10.47**	**1.53**	12.80	1.77	17.29
bible	4 047 392	63	1.43	8.43	**9.57**	1.42	10.00	**1.28**	14.53
world	2 473 400	94	1.15	8.33	**9.34**	1.18	9.60	**1.04**	14.54
Average			0.84	8.32	9.38	1.19	10.22	1.13	14.92

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 I shows the averages of the values of the corresponding columns. In each row, entries in bold face mark the smallest relative time and the smallest relative space requirement, respectively.

All three programs have similar average 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 [10]. *wotdeager* gives us a runtime advantage on data sets smaller than 10^6 characters, while for larger data, the extra logarithmic factor in its asymptotic construction time becomes visible gradually. *mccl*, plagued by its extra alphabet factor k, wins the race only for $n = 4 \times 10^6$ and $k = 4$. *mcch* is best for large data sizes in connection with large alphabets. In any case, *wotdeager* is more space efficient than the other programs, using 0.84 and 5.54 bytes per input character less than *mccl* and *mcch*, respectively. Note that the additional working space required for *wotdeager* is on average only 1.06 bytes per input character.

As already observed in [10], suffix trees for DNA sequences are larger than for other types of input strings. This is due to the small alphabet, which leads to a denser suffix tree with more branching nodes. This effect can be confirmed here for all algorithms, comparing space requirements for inputs *E.coli* and *bible*.

In a second experiment we studied the behavior of different programs searching for many exact patterns in an input string, a scenario that occurs for example in genome-scale sequencing projects, see [2] (Section 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, one cannot make 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 [12] (p. 946). The suffix array program, referred to by *mamy*, constructs a suffix array in $O(n \log n)$

Table II. Time and space requirement for searching $0.01n$ exact patterns.

File	n	k	*wotdlazy*			*wotdeager*	*mccl*	*mcch*	*mamy*		*bmh*
			rtime	*stspace*	*space*	*rtime*	*rtime*	*rtime*	*rtime*	*space*	*rtime*
bib	111 261	81	**0.18**	0.93	**5.24**	0.36	0.54	0.72	4.22	9.00	0.81
book1	768 771	82	**0.61**	0.90	**5.22**	0.75	1.70	1.21	6.57	9.00	11.24
book2	610 856	96	**0.51**	0.91	**5.22**	0.64	1.26	1.06	6.78	9.00	8.46
alice29	152 089	74	**0.20**	0.93	**5.23**	0.33	0.79	0.92	5.26	9.00	1.32
lcet10	426 754	84	**0.42**	0.88	**5.22**	0.59	1.17	1.05	6.51	9.00	5.67
plrabn12	481 861	81	**0.48**	0.88	**5.22**	0.62	1.47	1.14	7.14	9.00	6.79
E.coli	4 638 690	4	1.80	0.84	**5.42**	1.86	**1.59**	1.82	13.07	9.00	131.30
bible	4 047 392	63	**1.28**	0.81	**5.22**	1.45	1.53	1.31	10.81	9.00	59.32
world	2 473 400	94	**0.95**	0.80	**5.09**	1.17	1.28	1.07	9.38	9.00	32.31
Average			0.71	0.88	5.23	0.86	1.26	1.15	7.75	9.00	28.58
Construction time						0.84	1.19	1.13	7.73		
Search time						0.02	0.07	0.02	0.02		

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 [9], 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 and tested if they occurred in t or not. The proportionality factor ρ was between 0.0001 and 1. The lengths of the substrings 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. The reason for reversing the pattern string s_i in half of the cases is to simulate the fact that a pattern search is often unsuccessful. Table II shows the relative running times for $\rho = 0.01$. 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 I. The space requirement of *bmh* is marginal, so it is omitted in Table II.

Except for the DNA sequence, *wotdlazy* is the fastest and most space efficient program for $\rho = 0.01$. This is due to the fact that the pattern searches only evaluate a part of the suffix tree. Comparing the *stspace* columns of Tables I and II we can estimate that for $\rho = 0.01$ only about 10% of the suffix tree is evaluated. Of course, for the other programs, their tree construction time dominates the search time. We can also deduce that *wotdeager* performs pattern searches 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 new suffix tree representation, the successors

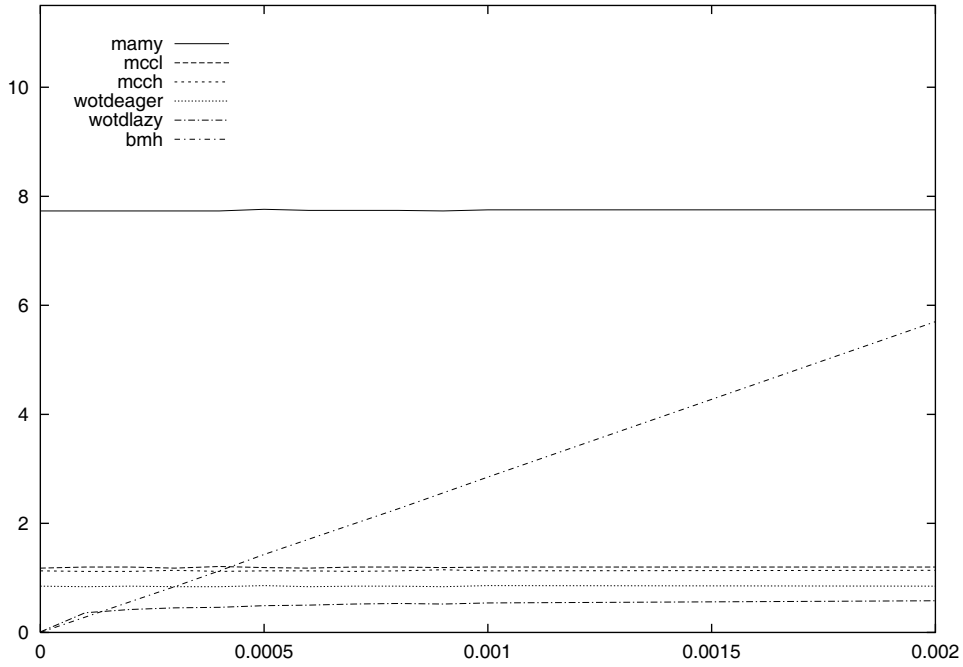

Figure 5. Average relative running time (in seconds) for different values of $\rho \in [0, 0.002]$.

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 ten times faster than *mamy*.

Of course, the space advantage of *wotdlazy* is lost with a larger number of pattern searches. In particular, for $\rho \geq 0.3$ *mamy* is the most space efficient program (data not shown). Figures 5 and 6 give a general overview, showing how ρ influences the running times. Figure 5 shows the average relative running time for all programs and different choices of ρ for $\rho \leq 0.002$. Figure 6 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 $\rho \leq 0.05$, and *wotdeager* is the fastest program for $\rho \geq 0.06$. *bmh* is faster than *wotdlazy* only for $\rho < 0.0002$. Thus the index construction performed by *wotdlazy* already amortizes for a very small number of pattern searches.

We also performed pattern searches on two very large biological sequence files *sprot39* (release 39 of the SWISSPROT protein database) and *yeast* (the complete genome of the baker's yeast *Saccharomyces cerevisiae*). The results are shown in Table III. We left out *bmh*, which would take extremely long running times, and we could also not test *mcch* since this program can only process files of length up to 8 388 605 characters. Our main observations are:

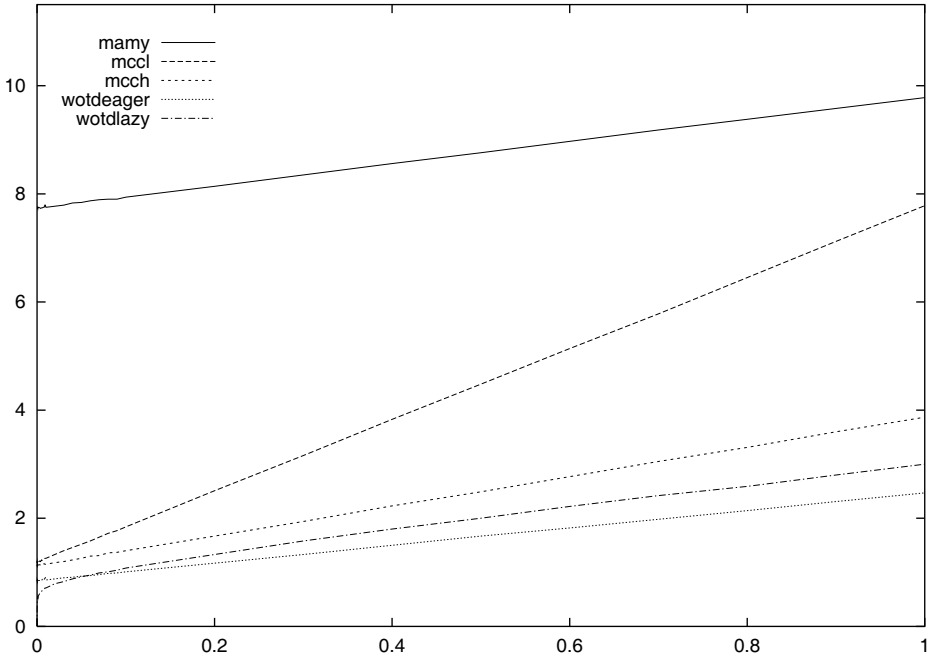

Figure 6. Average relative running time (in seconds) for different values of $\rho \in [0, 1]$.

Table III. Time and space requirements when searching $0.01n$ patterns in very large files.

			wotdlazy			*wotdeager*			*mccl*		*mamy*	
File	*n*	*k*	*rtime*	*stspace*	*space*	*rtime*	*stspace*	*space*	*rtime*	*space*	*rtime*	*space*
sprot39	33 805 297	24	1.58	1.14	5.22	1.60	7.91	8.55	5.66	10.00	51.40	9.00
yeast	12 329 970	5	2.37	0.89	5.62	2.23	9.00	10.55	1.94	12.40	16.44	9.00

- The relative running time for all programs clearly increases.
- 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 *mccl* (data not shown).
- The execution time for *mamy* rises sharply on our largest data set. Here, for searching $0.01n$ patterns for $n = 33 \times 10^6$, the extra logarithmic factor in the search procedure of *mamy* becomes visible.

CONCLUSION

We have developed efficient lazy and eager implementations of the write-only top-down suffix tree construction. These construct a representation of the suffix tree that requires only $12n$ bytes of space in the worst case, plus $10n$ bytes of working space. The total space requirement in practice, though, is only $9.38n$ bytes on average for a collection of files of different types. 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.

A special construction for repetitive strings

We end this exposition by discussing a special case of both practical relevance and theoretical interest. DNA, and in particular human DNA, largely consists of repetitive sequences. Estimates of repetitive contents go up to 50% and more [30] (p. 860). In the context of biosequence analysis, it is unfortunate that the *wotd*-algorithm performs most poorly on repetitive strings. The worst case running time of $O(n^2)$ is achieved on strings of form $a^n\$$. The *wotd*-algorithm performs similarly on Fibonacci strings (see e.g. [31], exercise 1.2.8-36), which are known for their highly repetitive structure. This is particularly unfortunate since Fibonacci strings, with their iterative pattern that grows repetitive strings via recombination of previously generated strings, may be seen as an idealized model of the effects of DNA cutting, repair, and recombination.

Let us analyze why the *wotd*-algorithm performs poorly on repetitive strings. Long repeats often lead to suffix trees with long interior edges, and significant effort is spent on determining the longest common prefix of the suffixes contributing to a subtree below such an edge. To be more precise, the labels of edges leaving the nodes $\overline{w}$ and $\overline{aw}$ for $a \in \Sigma$ and $w \in \Sigma^*$ are often identical, or the former is a prefix of the latter. This is because the suffixes in $R(\overline{aw})$ are (apart from the constant prefix a) a subset of the suffixes in $R(\overline{w})$.

This suggests the following improvement of the *wotd*-algorithm: when evaluating node $\overline{aw}$, we can immediately skip the first m character comparisons if m is the length of the corresponding lcp computed for $\overline{w}$, and only then continue with the normal computation of the lcp. In a more advanced algorithm, one can apply this idea even recursively, but it is not the aim of this exposition to elaborate on such details. A prototype implementation of this idea has been tested, and in fact, it seems to outperform all other suffix tree constructions on Fibonacci strings.

While still being a top-down construction, the algorithm sketched here is no longer write-only, as it accesses node $\overline{w}$ when evaluating $\overline{aw}$. Hence many of the arguments given in favor of the *wotd*-algorithm, in particular locality of memory reference, must be re-evaluated for this variant.

But there are further aspects to this idea, interesting in their own right. The dependence of $\overline{aw}$ on $\overline{w}$ induces a partial ordering on the construction of nodes. In the extreme, the suffix tree is constructed in order of increasing suffix length, and we have a reverse online construction in perfect analogy to Weiner's landmark algorithm of 1973 [4]. This surprising observation may provide new insights into the relationships between different suffix tree constructions.

ACKNOWLEDGEMENTS

We thank Gene Myers for providing a copy of his suffix array code. S. Kurtz was supported by grant Ku 1257/1-1 from the Deutsche Forschungsgemeinschaft.

REFERENCES

1. Apostolico A. The myriad virtues of subword trees. *Combinatorial Algorithms on Words*. Springer: Berlin, 1985; 85–96.
2. Gusfield D. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press: Cambridge, 1997.
3. Skiena SS. Who is interested in algorithms and why? Lessons from the Stony Brook Algorithms Repository. *Proceedings 2nd Workshop on Algorithm Engineering*. IEEE Press: New York, 1998; 204–212. http://www.mpi-sb.mpg.de/∼wae98/PROCEEDINGS/.
4. Weiner P. Linear pattern matching algorithms. *Proceedings 14th Annual IEEE Symposium on Switching and Automata Theory*, 1973; 1–11.
5. McCreight EM. A space-economical suffix tree construction algorithm. *Journal of the ACM* 1976; **23**(2):262–272.
6. Ukkonen E. On-line construction of suffix-trees. *Algorithmica* 1995; **14**(3):249–260.
7. Giegerich R, Kurtz S. From Ukkonen to McCreight and Weiner: a unifying view of linear-time suffix tree constructions. *Algorithmica* 1997; **19**(3):331–353.
8. Giegerich R, Kurtz S. A comparison of imperative and purely functional suffix tree constructions. *Science of Computer Programming* 1995; **25**(2–3):187–218.
9. Horspool RN. Practical fast searching in strings. *Software—Practice and Experience* 1980; **10**(6):501–506.
10. Kurtz S. Reducing the space requirement of suffix trees. *Software—Practice and Experience* 1999; **29**(13):1149–1171.
11. Kurtz S, Choudhuri JV, Ohlebusch E, Schleiermacher C, Stoye J, Giegerich R. REPuter: the manifold applications of repeat analysis on a genomic scale. *Nucleic Acids Research* 2001; **29**(22):4643–4653.
12. Manber U, Myers EW. Suffix arrays: A new method for on-line string searches. *SIAM Journal of Computing* 1993; **22**(5):935–948.
13. Farach M. Optimal suffix tree construction with large alphabets. *Proceedings 38th Annual Symposium on the Foundations of Computer Science*. IEEE Press: New York, 1997; 137–143.
14. Dorohonceanu B, Nevill-Manning CG. Protein classification using suffix trees. *Proceedings of the Eighth International Conference on Intelligent Systems for Molecular Biology*. AAAI Press: Menlo Park, CA, 2000; 128–133.
15. Hunt E, Atkinson MP, Irving RW. A database index to large biological sequences. *Proceedings of the 27th International Conference on Very Large Databases*. Morgan Kaufmann: San Francisco, 2001; 139–148.
16. Clark DR, Munro JI. Efficient suffix trees on secondary storage. *Proceedings of the Seventh Annual ACM–SIAM Symposium on Discrete Algorithms*. ACM Press: New York, 1996; 383–391.
17. Munro JI, Raman V, Rao SS. Space efficient suffix trees. *Journal of Algorithms* 2001; **39**(2):205–222.
18. Grossi R, Vitter JS. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *Proceedings of the 32th Annual ACM Symposium on the Theory of Computing*. ACM Press: New York, 2000; 397–406.
19. Andersson A, Nilsson S. Efficient implementation of suffix trees. *Software—Practice and Experience* 1995; **25**(2):129–141.
20. Irving RW, Love L. Suffix binary search trees and suffix arrays. *Research Report TR-2001-82*, Department of Computer Science, University of Glasgow, 2001.
21. Kärkkäinen J. Suffix cactus: A cross between suffix tree and suffix array. *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching* (*Lecture Notes in Computer Science*, vol. 937). Springer: Berlin, 1995; 191–204.
22. Allauzen C, Crochemore M, Raffinot M. Factor oracle: a new structure for pattern matching. *Proceedings of the 26th Annual Conference on Current Trends in Theory and Practice of Informatics* (*Lecture Notes in Computer Science*, vol. 1725). Springer: Berlin, 1999; 291–306.
23. Colussi L, De Col A. A time and space efficient data structure for string searching on large texts. *Information Processing Letters* 1996; **58**(5):217–222.
24. Giegerich R, Kurtz S, Stoye J. Efficient implementation of lazy suffix trees. *Proceedings of the Third Workshop on Algorithmic Engineering* (*Lecture Notes in Computer Science*, vol. 1668). Springer: Berlin, 1999; 30–42.
25. Martinez HM. An efficient method for finding repeats in molecular sequences. *Nucleic Acids Research* 1983; **11**(13):4629–4634.
26. Gusfield D. An 'increment-by-one' approach to suffix arrays and trees. *Report CSE-90-39*, Computer Science Division, University of California, Davis, 1990.
27. Apostolico A, Iliopoulos C, Landau GM, Schieber B, Vishkin U. Parallel construction of a suffix tree with applications. *Algorithmica* 1988; **3**:347–365.
28. Hopcroft J. An $O(n \log n)$ algorithm for minimizing states in a finite automaton. *The Theory of Machines and Computations*, Kohavi Z, Paz A (eds.). Academic Press: New York, 1971; 189–196.
29. Cormen TH, Leiserson CE, Rivest RL. *Introduction to Algorithms* (2nd edn). MIT Press: Cambridge, MA, 2001.
30. International Human Genome Sequencing Consortium. Initial sequencing and analysis of the human genome. *Nature* 2001; **409**:860–921.
31. Knuth DE. The Art of Computer Programming. *Fundamental Algorithms* (3rd edn), vol. 1. Addison-Wesley: Reading, MA, 1997.

