SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper., **29**(13), 1149–1171 (1999)

Reducing the Space Requirement of Suffix Trees

STEFAN KURTZ*
Technische Fakultät, Universität Bielefeld, Postfach 100 131, 33501 Bielefeld, Germany
(e-mail: kurtz@techfak.uni-bielefeld.de)

SUMMARY

We show that suffix trees store various kinds of redundant information. We exploit these redundancies to obtain more space efficient representations. The most space efficient of our representations requires 20 bytes per input character in the worst case, and 10.1 bytes per input character on average for a collection of 42 files of different type. This is an advantage of more than 8 bytes per input character over previous work. Our representations can be constructed without extra space, and as fast as previous representations. The asymptotic running times of suffix tree applications are retained. Copyright © 1999 John Wiley & Sons, Ltd.

KEY WORDS: data structures; suffix trees; implementation techniques; space reduction

INTRODUCTION

Suffix trees provide efficient access to all substrings of a string, and they can be constructed and represented in linear time and space. These properties make suffix trees a data structure whose simplicity and elegance is surpassed only by their versatility. No other idea in the realm of string processing can be adapted so easily to achieve superb efficiency in such a great variety of applications. Apostolico [1] gives over 40 references on suffix trees, and Manber and Myers [2] add several more recent ones. A very thorough discussion of current knowledge on suffix tree constructions and applications can be found in the textbook by Gusfield [3].

Despite these superior features and the wide acceptance by theoretical computer scientists, suffix trees have not seen widespread use in string processing software, in contrast to, for example, finite automata or hashing techniques. One of the main reasons for this is that suffix trees have a reputation of being very greedy for space. In fact, the suffix tree implementation described by McCreight [4] requires $28n$ bytes in the worst case, where n is the length of the input string.† The space requirement in practice is smaller, but previous authors do not give consistent numbers:

(a) Manber and Myers [2] state that their implementation of suffix trees occupies between $18.8n$ and $22.4n$ bytes of space for real input strings (text, code, DNA).‡

*Correspondence to: Stefan Kurtz, Technische Fakultät, Universität Bielefeld, Postfach 100131, 33501 Bielefeld, Germany.
†We will use *bytes* or *integers* as units when we state results on space requirements. The assumption is always that an integer occupies four bytes. Unless stated otherwise, the given numbers do not include the n bytes for representing the input string.
‡These numbers have been derived from the third column of Table 1 in the paper of Manber and Myers [2]: we just added the space for the suffix links, which is $4q$ bytes where q is the number of internal nodes.

CCC 0038–0644/99/131149–23$17.50

Received 5 February 1999
Revised 7 June 1999
Accepted 14 June 1999

(b) Kärkkäinen [5] claims that a suffix tree can be implemented in $15n - 18n$ bytes of space for real input strings. Unfortunately, it is not shown how to achieve this.
(c) Crochemore and Vérin [6] state that suffix trees require $32.7n$ bytes for DNA sequences.
(d) The *strmat* software package by Knight, Gusfield and Stoye [7] implements suffix trees in $24n - 28n$ bytes for input strings of length at most $2^{23} = 8{,}388{,}608$. However, *strmat* can handle sets of strings, and it is unclear how much of the space requirement is due to this additional feature.

It is important to note that these numbers include the space required during the construction of suffix trees. Recently, Munro *et al.* [8] described a representation of suffix trees which requires $n\lceil \log_2 n \rceil + o(n)$ bits. However, it is restricted to searching for string patterns, and it is not clear if there is a linear time algorithm to directly construct this representation. As a consequence, one first has to construct a suffix tree in a usual, less space efficient representation. So, altogether, the approach of Munro *et al.* sacrifices versatility and it does not give a space advantage in practice.

Faced with the numbers above, and the ever growing size of the input strings to be processed, several authors have developed alternative index structures which store less information than suffix trees and are therefore more space efficient: the *suffix array* of Manber and Myers [2] requires $9n$ bytes (including the space for construction). The *level compressed trie* of Andersson and Nilsson [9] takes about $12n$ bytes. The *suffix binary search tree* of Irving [10] requires $10n$ bytes. The *suffix cactus* of Kärkkäinen [5] can be implemented in $10n$ bytes. Finally, the PT-tree of Colussi and De Col [11] requires $n \log_2 n + O(n)$ bits. These five index structures have two properties in common. First, they are specifically tailored to solve string matching problems, and cannot be adapted to other kinds of problems without severe performance penalties.[§] Thus they are not nearly as versatile and efficient as suffix trees (and they are not expected to be). Second, the direct construction methods for these index structures do not run in linear worst case time.[¶]

Directed acyclic word graphs [12,13] (*dawg*s, for short), and more space efficient variants thereof [14,15], have essentially the same applications as suffix trees. The compact dawg, which is the most space efficient of these index structures, occupies $36n$ bytes in the worst case. Recently, Crochemore and Vérin [6] gave a direct method to construct *compact dawgs*, which makes this index structure useful in practice. We will later see that *compact dawgs* are more space efficient than suffix trees in previous implementations, but less space efficient than suffix trees in an implementation technique we propose. Dawgs, and in particular compact dawgs have been less extensively studied than suffix trees. According to Crochemore and Vérin [6], this may be due to the fact that they display positions of substrings of the input string in a less obvious way.

To allow constructions and applications of suffix trees for very large input strings (like they occur in genome research), other authors [16,17] developed techniques to organize suffix trees on disk, so that the number of disk accesses is reduced. However, again these techniques are mainly optimized for string matching problems, and the behavior for other kinds of applications is unclear. Moreover, direct construction in linear time is not possible.

In this paper, we follow the most natural approach to make suffix trees more practical: we reduce their space requirement. We show that suffix trees store various kinds of

[§]String matching problems are perhaps the most important kind of applications for index structures. However, there are other important applications, like finding repetitive structures in strings or sorting suffixes; see also Table VI.

[¶]All five index structures can be constructed *indirectly* in linear time. The idea is to first construct the corresponding suffix tree, and then to traverse it to read off the information of the particular index structure, but this indirect approach of course means that the space advantage is lost.

redundant information, and we exploit these redundancies to obtain a more space efficient representation. We are mainly interested to reduce the space in practice, but we also improve on the worst case. We emphasize that we do not sacrifice any of the superior virtues of suffix trees as mentioned above. In particular, the suffix tree representations we propose can be constructed in linear worst case time without using extra space and the asymptotic running times of suffix tree applications are retained. This approach, which, to our knowledge, has not been consequently followed since the pioneering work of McCreight,[‖] has an important advantage: suffix trees and their applications have been extensively studied and are well described in textbooks. All this work can be implemented without change of algorithms on top of our space efficient representations.

The main contributions of this paper are as follows:

(a) We make several observations about the node structure of suffix trees, which reveal redundancies of the information stored therein.
(b) We show how to exploit these redundancies to improve the space requirement of previous implementation techniques based on linked list and hash tables. The worst case space requirement of the improved linked list implementation is $5n$ integers, and it is probably even better but we cannot prove this. Thus the improvement in the worst case is $2n$ integers over the technique described by McCreight [4]. It is interesting to note that for the string a^n, where McCreight's techniques occupy $7n$ integers, both of our improved implementation techniques require at most $3n + \frac{2}{31}n$ integers. The worst case space requirement of the improved hash table implementation technique is $7n$ integers. Again, we do not know if this bound is tight. These results hold for input strings of length up to $2^{27} - 1 = 134{,}217{,}727$.
(c) We show that on a 32 bit computer all implementations of suffix trees have similar upper bounds on the maximal length of the input string they allow.
(d) We present experimental results showing that our improved linked list implementation requires on average $10.1n$ bytes of space for a collection of 42 files from different sources (english text, formal text, binary files, DNA sequences, protein sequences, random strings). This is an improvement of 46 per cent over the implementation technique of McCreight, and an improvement of 30 per cent over compact dawgs. The improved hash table implementation technique requires $14.66n$ bytes on average, which is similar to the space consumption of compact dawgs. Our experiments show that the size of the index structures depends on the kind of input data: binary strings lead to the smallest data structures, for formal text and english text all data structures are slightly larger. For protein sequences and in particular DNA sequences the space requirement is considerably higher.
(e) Timing results show that the space efficient representations we propose can be computed with virtually no performance penalty in practice. The linked list implementation proves to be faster than the hash table implementation only if the alphabet is small and the input string is short.
(f) In the conclusion we shortly sketch current and possible applications of our implementation techniques, and give advice on which of the proposed techniques to choose. We argue that it is very important to consider the kind of suffix tree traversals an application requires.

This paper extracts the core of wider report [18], where we give proofs for the observations,

[‖] Andersson and Nilsson [9] consider level compressed tries which are different from suffix trees as defined by McCreight [4].

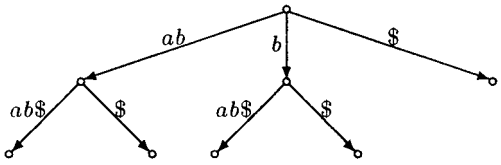

Figure 1. The suffix tree for $x = abab$

and describe how to modify McCreight's suffix tree construction [4] such that it computes the space efficient representations we propose. Documented C-source code constructing the proposed suffix tree representations in linear time is available at
`http://www.techfak.uni-bielefeld.de/~kurtz/Software/suffixtrees.tar.gz`.

The code works on 32 bit as well as on 64 bit machines without any changes.

SUFFIX TREES

Basic definitions

Let Σ be a finite ordered set, the *alphabet*. The size of Σ is k. Σ^* denotes the set of all strings over Σ and ε is the *empty string*. We use Σ^+ to denote the set $\Sigma^*\backslash\{\varepsilon\}$ of non-empty strings. Let $x \in \Sigma^*$ and $x = uvw$ for some possibly empty strings u, v, w. Then u is a *prefix* of x, v is a *substring* of x, w is a *suffix* of x. $|x|$ is the number of characters in x. x_i is the ith character in x. If $|x| = n$, then $x = x_1 x_2 \ldots x_n$.

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.

Let T be a Σ^+-tree. A node in T is *branching* if it has at least two outgoing edges. A *leaf* in T is a node in T with no outgoing edges. An *internal node* in T is either the *root* or a node with at least one outgoing edge. An edge leading to an internal node is an *internal edge*. An edge leading to a leaf is a *leaf edge*. Due to the requirement of unique a-edges at each node of T, paths are also unique. Therefore, we denote v by $\overline{w}$ if and only if w is the concatenation of the edge labels on the path from the *root* of T to the node v. The node $\overline{\varepsilon}$ is the *root*. For any node $\overline{w}$ in T, $|w|$ is the *depth* of $\overline{w}$. A string w *occurs* in T if T contains a node $\overline{wu}$, for some string u.

From now on we assume that $x \in \Sigma^+$ is a string of length $n \geq 1$ and that $\$ \in \Sigma$ is a character not occurring in x, the *sentinel*. The *suffix tree* for x, denoted by ST, is the Σ^+-tree T with the following properties: (i) each node is either a leaf, a branching node, or the *root*; and (ii) a string w occurs in T if and only if w is a substring of $x\$$. Figure 1 shows the suffix tree for $x = abab$. There are several algorithms to construct ST in linear time [4,19,20,21]. Giegerich and Kurtz [22] review three of these algorithms and reveal relationships much closer than one would think.

For any $i \in [1, n+1]$, let $S_i = x_i \ldots x_n\$$ denote the ith non-empty suffix of $x\$$. Note that due to the sentinel, no S_i is a proper prefix of any S_j. Thus, there is a one-to-one correspondence between the non-empty suffixes of $x\$$ and the leaves of ST. This implies that ST has exactly $n+1$ leaves. Moreover, since $n \geq 1$ and $x_1 \neq \$$, the *root* of ST is branching. Hence, each internal node in ST is branching. This means that there are at most n internal nodes in ST. Each node can be represented in constant space. Thus, one needs $O(n)$ space for the nodes. Since ST has at most $2n+1$ nodes, the number of edges is bounded by $2n$. Each edge is labeled by a substring of $x\$$, which can be represented in constant space by

a pair of pointers into $x\$$. Hence, one needs $O(n)$ space for the edges. Altogether, ST requires $O(n)$ space.

The *suffix link* for a node $\overline{aw}$ in ST is an unlabeled directed edge in ST from $\overline{aw}$ to the node $\overline{w}$. We consider suffix links to be a part of the suffix tree data structure. They are required for most of the linear time suffix tree constructions [4,19,20], and for some applications of suffix trees [3].

Head positions

The substring w corresponding to the branching node $\overline{w}$ can be represented by a position delineating an occurrence of w in $x\$$. As w may occur several times in $x\$$, there are several choices for a position, and it is common practice to choose the leftmost occurrence. We shall show now that there is a less obvious, but more convenient choice: the *raison d'etre* of a branching node $\overline{w}$ is not the leftmost occurrence of w in $x\$$, but the leftmost *branching* occurrence. That is, the first occurrence of wa in $x\$$, for some $a \in \Sigma$, such that w occurs to the left, but not wa.

Let $head_1 = \varepsilon$ and for $i \in [2, n+1]$ let $head_i$ be the longest prefix of S_i which is also a prefix of S_j for some $j \in [1, i-1]$. The following two observations show that there is a one-to-one correspondence between the *head*'s and the branching nodes in ST. The proofs for these and all subsequent observations can be found elsewhere [18].

Observation 1 Let $\overline{w}$ be a branching node in ST. Then there is an $i \in [1, n+1]$ such that $w = head_i$.

Observation 2 Let $i \in [1, n+1]$. Then there is a branching node $\overline{head_i}$ in ST.

For each branching node $\overline{w}$ in ST, let $headposition(\overline{w})$ denote the smallest integer $i \in [1, n+1]$ such that $w = head_i$. According to Observation 1, such an integer exists, and hence $headposition(\overline{w})$ is well defined. If $headposition(\overline{w}) = i$, then we say that the *head position* of $\overline{w}$ is i.

While the determination of the head positions seems more complicated than just choosing the position of the leftmost occurrence, the head position is readily available during linear time suffix tree construction [18].

TWO SIMPLE IMPLEMENTATION TECHNIQUES

The most space parsimonious implementation techniques for suffix trees is based on linked lists [2]. McCreight [4] (Fig. 4) showed how to represent ST using five integers for each internal node and two integers for each leaf. No extra space for the edges and their labels is required. Later authors gave the same numbers [2,10]. Recently, Crochemore and Vérin [6] (p. 121) claimed that McCreight's implementation technique would also require five integers for each leaf. This is not true. In the next section we show that one integer suffices for each leaf.

A simple linked list implementation

The simple linked list implementation technique (*SLLI* for short) represents ST by two tables T_{leaf} and T_{branch} which store the following values: for each *leaf number* $j \in [1, n+1]$, $T_{leaf}[j]$ stores a reference to the right brother of leaf $\overline{S_j}$. If there is no such brother, then

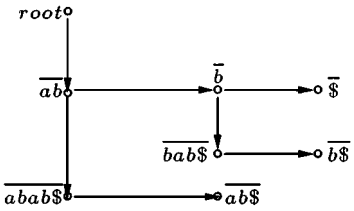

Figure 2. The references of the suffix tree for $x = abab$ (see Figure 1). Vertical arcs stand for firstchild references, and horizontal arcs for branchbrother and T_{leaf} references

$T_{leaf}[j]$ is a nil reference. For each branching node $\overline{w}$, $T_{branch}[\overline{w}]$ stores a *branch record* consisting of five components *firstchild*, *branchbrother*, *depth*, *headposition*, and *suffixlink* whose values are specified as follows:

1. *firstchild* refers to the first child of $\overline{w}$.
2. *branchbrother* refers to the right brother of $\overline{w}$. If there is no such brother, then *branchbrother* is a nil reference.
3. *depth* is the depth of $\overline{w}$.
4. *headposition* is the head position of $\overline{w}$.
5. *suffixlink* refers to the branching node $\overline{v}$, if w is of the form av for some $a \in \Sigma$ and some $v \in \Sigma^*$.

The successors of a branching node are therefore found in a list whose elements are linked via the *firstchild*, *branchbrother* and T_{leaf} references. To speed up the access to the successors, each such list is ordered according to the first character of the edge labels. Figure 2 shows the child and brother references of the nodes of the suffix tree of Figure 1. We use the following notation to denote a record component: for any component c and any branching node $\overline{w}$, $\overline{w}.c$ denotes the component c stored in the branch record $T_{branch}[\overline{w}]$. Note that the head position j of some branching node $\overline{wu}$ tells us that the leaf $\overline{S_j}$ occurs in the subtree below node $\overline{wu}$. Hence, wu is the prefix of S_j of length $\overline{wu}.depth$, i.e. the equality $wu = x_j \ldots x_{j+\overline{wu}.depth-1}$ holds. As a consequence, the label of the incoming edge to node $\overline{wu}$ can be obtained by dropping the first $\overline{w}.depth$ characters of wu, where $\overline{w}$ is the predecessor of $\overline{wu}$:

Observation 3 If $\overline{w} \xrightarrow{u} \overline{wu}$ is an edge in ST and $\overline{wu}$ is a branching node, then $u = x_i \ldots x_{i+l-1}$ where $i = \overline{wu}.headposition + \overline{w}.depth$ and $l = \overline{wu}.depth - \overline{w}.depth$.

Similarly, the label of the incoming edge to a leaf is determined from the leaf number and the depth of the predecessor:

Observation 4 If $\overline{w} \xrightarrow{u} \overline{wu}$ is an edge in ST and $\overline{wu} = \overline{S_j}$ for some $j \in [1, n+1]$, then $u = x_i \ldots x_n\$$, where $i = j + \overline{w}.depth$.

A similar observation was made by Larsson [23], but without a clear statement about its consequences concerning the space requirement of ST. Note that storing the depth of a branching node has some practical advantages over storing the length of the incoming edge to a node (the latter is suggested by McCreight [4]). At first, during the sequential suffix tree constructions [4,19,20], the depth of a node never changes. So it is not necessary to update the depth of a node (the same is true for the head position). Second, the depth of the nodes along a chain of suffix links is decremented by one, a property which can be exploited to store a suffix tree more space efficiently, see the next section. The third advantage of storing the depth is that several applications of suffix trees assume that the depth of a node is available [3].

Table I. Tables T_{leaf} and T_{branch} representing the suffix tree for $x = abab$ (see Figure 1). A bold face number refers to table T_{leaf}

T_{leaf}					
leaf	$\overline{abab\$}$	$\overline{bab\$}$	$\overline{ab\$}$	$\overline{b\$}$	$\overline{\$}$
leaf number j	1	2	3	4	5
$T_{leaf}[j]$	**3**	**4**	nil	nil	nil

T_{branch}			
branching node	*root*	$\overline{ab}$	$\overline{b}$
node number	1	2	3
firstchild	2	**1**	**2**
branchbrother	nil	3	**5**
depth	0	2	1
headposition	1	3	4
suffixlink		3	1

Space requirement

The *firstchild*, *branchbrother* and T_{leaf} references can be implemented as integers in the range $[0, n]$. An extra bit with each such integer tells whether the reference is to a leaf or to a branching node. Each leaf $\overline{S_j}$ is referred to by leaf number j. Suppose there are q branching nodes in ST. Let $b_1, b_2, \ldots, b_q$ be the sequence of branching nodes ordered by their head position, i.e. $b_i.headposition < b_{i+1}.headposition$ for any $i \in [1, q-1]$. Each branching node b_i is referred to by its node number i, which is denoted by $nodenum(b_i)$. Obviously, b_1 is the *root*. Table I depicts T_{leaf} and T_{branch} for the suffix tree of Figure 1.

Like the references, the other components of the branch records can each be implemented by an integer in the range $[0, n]$. Thus, table T_{leaf} requires n integers and table T_{branch} requires $5q$ integers. The total space requirement of the simple linked list implementation is $n + 5q$ integers. The linked list implementation technique of McCreight requires $2n + 5q$ integers. Thus *SLLI* saves n integers.

In the worst case, we have $q = n$, so that *SLLI* requires $6n$ integers. McCreight [4] (p. 268) suggested to store the node with head position i at index i in T_{branch}. In this way, it is not required to store the head position with each internal node. This would reduce the space for each branch record to four integers, and the space requirement would be $5n$ integers, independent of the actual number q of branching nodes. However, q is usually considerably smaller than $0.8n$ ($q = 0.62n$ is the theoretical average value for random strings [24]), so that this worst case improvement would result in a larger space usage in practice. Therefore, we do not consider it further.

Note that storing the nodes of the suffix tree in depth first or breadth first order (as in Giegerich *et al.* [25]) to save the space for the *firstchild*- or *branchbrother*-references does not allow linear time construction. This is because during linear time suffix tree constructions the relations of the nodes change dynamically. That is, if a node is created, then it is not clear what the brother or first child will be in the final suffix tree. Hence, if we would store the nodes in depth first or breadth first order, the insertion of a new node would require an unbounded number of node movements. This is in contrast to the three or four updates of references required if we store the node relations as described above.

A simple hash table implementation

While linked list implementations of suffix trees are space efficient, they have an important disadvantage: it takes $O(k)$ time to select a certain edge (according to some given character)

outgoing from a node. If the alphabet is large this may slow down suffix tree constructions and traversals considerably. Using balanced search trees instead of linked lists would improve worst case access to $O(\log k)$. However, with the additional overhead and the additional space requirement it is not clear whether this would improve the running time in practice. For this reason, we consider hashing techniques. McCreight [4] already suggested these for the implementation of suffix trees.

We found the following simple hash table implementation technique (*SHTI*, for short) to work well in practice: for each edge $\overline{w} \stackrel{av}{\rightarrow} \overline{wav}$ one stores the node number of $\overline{wav}$ in a hash table using the pair $(\overline{w}.headposition, a)$ as a hash key. Given a node $\overline{w}$ and some character a, the hash table allows to check whether there is an a-edge outgoing from $\overline{w}$. In case such an edge exists, say $\overline{w} \stackrel{av}{\rightarrow} \overline{wav}$, a reference to $\overline{wav}$ is delivered, as well as the edge label av.

Space requirement

The number of edges is bounded by $2n$, so a hash table of size $2n$ suffices. Besides the hash table, there is a table storing a record for each branching node. Such a record consists of the three components *depth*, *headposition* and *suffixlink*, as defined above, and so this table requires $3q$ integers. The space requirement for the hash table depends on the hashing technique. We use an open addressing hashing technique, with double hashing [26] to resolve collisions. The hash function is based on the division method. This implies that the actual size of the hash table is the smallest prime larger than $2n$. Each entry of the hash table stores two integers: the hashed value and the left component of the hash key. It is not necessary to store the right component of the hash key (i.e. the character), since this can be retrieved in constant time, provided the depth of the node the edge is outgoing from is known; see Observations 3 and 4. The hash table thus requires $4n$ integers, which means that the total space requirement of *SHTI* is $4n + 3q$ integers.

Note that McCreight recommends to use Lampson's hashing technique (see Knuth [26], section 6.4, p. 543, and Example 13), which belongs to the class of chaining techniques. This hashing technique uses an overflow area and a linked list of synonyms, and saves space by only storing the remainder of the key. However, as remarked by Cleary [27], each hash table entry (including the original hash location) requires a reference to the next overflow record. This reference will be of about the same size as the reduction in the key size. So, Lampson's hashing technique does not lead to net memory space savings over the open addressing technique we used. In other words, the latter is the better choice.

We considered other hashing and tree implementation methods, which are, however, not applicable to suffix trees without severe performance penalties:

(a) Compact Hashing [27] allows to store hash tables in a more space efficient way, by abbreviating a hash key (which we already did by only storing one component of the hash key). It requires randomizing hash keys, which can be very time consuming in practice.
(b) The *Bonsai* implementation technique [28] is based on Compact Hashing, while the *double-array* technique [29,30] combines the advantages of arrays and lists. Both techniques are specifically designed to represent trees space efficiently. However, they both required the tree to be built from the root downward, a precondition which is not met by any of the linear time suffix tree construction methods [4,19,20,21].

REVEALING AND EXPLOITING REDUNDANCIES

Small nodes and large nodes

We now show that the information stored for the branching nodes of the suffix tree contains redundancies. We reveal these by studying properties of the head positions. This leads to a relation between node numbers and suffix links.

Observation 5

1. If $\overline{u}$ and $\overline{w}$ are different branching nodes, then $\overline{u}$.*headposition* $\neq \overline{w}$.*headposition.*
2. For any branching node $\overline{aw}$ in *ST*, $\overline{w}$ is also a branching node in *ST*. Moreover, the inequality $\overline{aw}$.*headposition* $+ 1 \geq \overline{w}$.*headposition* holds.

Observation 5 implies that for any branching node $\overline{aw}$ we either have $\overline{aw}$.*headposition* $+ 1 = \overline{w}$.*headposition* or $\overline{aw}$.*headposition* $> \overline{w}$.*headposition.* We discriminate all non-root nodes accordingly: $\overline{aw}$ is a *small* node if and only if $\overline{aw}$.*headposition* $+ 1 = \overline{w}$.*headposition.* $\overline{aw}$ is a *large* node if and only if $\overline{aw}$.*headposition* $> \overline{w}$.*headposition.* The *root* is neither small nor large. The following observation shows that a small node is always directly followed by another branching node, and that the last branching node is a large node.

Observation 6 Let $\overline{aw}$ be a branching node in *ST*. Then the following holds:

1. If $\overline{aw}$ is small, then $nodenum(\overline{aw}) + 1 = nodenum(\overline{w})$.
2. If $nodenum(\overline{aw}) = q$ and $q > 1$, then $\overline{aw}$ is large.

According to Observation 6, we can partition the sequence $b_2, \ldots, b_q$ of branching nodes into *chains* of zero or more consecutive small nodes followed by a single large node: a *chain* is a contiguous subsequence $b_l, \ldots, b_r$, $r \geq l$, of $b_2, \ldots, b_q$ such that the following holds:

(a) b_{l-1} is not a small node.
(b) $b_l, \ldots, b_{r-1}$ are small nodes.
(c) b_r is a large node.

One easily observes that any branching node (except for the *root*) in *ST* is a member of exactly one chain. The branch records for the small nodes of a chain store some redundant information, as shown in the following observation:

Observation 7 Let $b_l, \ldots, b_r$ be a chain. The following properties hold for any $i \in [l, r-1]$:

1. b_i.*depth* $= b_r$.*depth* $+ (r - i)$.
2. b_i.*headposition* $= b_r$.*headposition* $- (r - i)$.
3. b_i.*suffixlink* $= b_{i+1}$.

We now show how to exploit these redundancies to store the information in the branching nodes in less space. Consider a chain $b_l, \ldots, b_r$. Observation 7 shows that it is not necessary to store b_i.*depth*, b_i.*headposition* and b_i.*suffixlink* for any $i \in [l, r-1]$: b_i.*suffixlink* refers to the next node in the chain, and if the distance $r - i$ of the small node b_i to the large node b_r (denoted by b_i.*distance*) is known, then b_i.*depth* and b_i.*headposition* can be obtained in constant time. This observation leads to the following implementation technique.

Table II. The table T'_{branch} for the suffix tree for $x = abab$ (see Figure 1). A small record is stored in two integers and a large record in four integers. $\overline{ab}$ is a small node with head position 3, and $\overline{b}$ is a large node with head position 4. Both form a chain. The distance of $\overline{ab}$ and $\overline{b}$ is 1, depicted as a tiny 1 in the small record for $\overline{ab}$. Consider the large record for $\overline{b}$: the tiny 0 stands for the unused most significant bits of the first integer, and the third integer stores the small depth (to the right) and the suffix link (to the left). A bold face number refers to T_{leaf}

5	nil	0	1	$_1$ **1**	8	$_0$ **2**	**5**	0 $_1$	4
		root			$\overline{ab}$			$\overline{b}$	

An improved linked list implementation

The improved linked list implementation (*ILLI*, for short) represents *ST* by two tables T_{leaf} and T'_{branch}. T_{leaf} is as in *SLLI*. Table T'_{branch} stores the information for the small and the large nodes: for each small node $\overline{w}$, there is a *small record* which stores $\overline{w}$.*distance*, $\overline{w}$.*firstchild* and $\overline{w}$.*rightbrother*. For each large node $\overline{w}$ there is a *large record* which stores $\overline{w}$.*firstchild*, $\overline{w}$.*rightbrother*, $\overline{w}$.*depth* and $\overline{w}$.*headposition*. Whenever $\overline{w}$.*depth* $\leq 2^\alpha - 1$, for some constant α, we say that the large record for $\overline{w}$ is *complete*. A complete large record also stores $\overline{w}$.*suffixlink*. A large node $\overline{w}$ with $\overline{w}$.*depth* $> 2^\alpha - 1$ is handled as follows: let $\overline{v}$ be the rightmost child of $\overline{w}$. There is a sequence consisting of one *firstchild* reference and at most $k - 1$ *rightbrother*/T_{leaf} references which link $\overline{w}$ to $\overline{v}$. If $\overline{v} = \overline{S_j}$ for some $j \in [1, n + 1]$, then $T_{leaf}[j]$ is a nil reference. Otherwise, if $\overline{v}$ is a branching node, then $\overline{v}$.*rightbrother* is a nil reference. Of course, it only requires one bit to mark a reference as a nil reference. Hence the integer used for the nil reference contains unused bits, in which $\overline{w}$.*suffixlink* is stored. As a consequence, retrieving the suffix link of $\overline{w}$ requires traversing the list of successors of $\overline{w}$ until the nil reference is reached, which encodes the suffix link of $\overline{w}$. This *linear retrieval* of suffix links takes $O(k)$ time in the worst case. However, despite linear retrieval, the suffix tree can still be constructed in $O(kn)$ time, since the suffix link is retrieved at most n times during suffix tree construction [4,18]. Moreover, suffix tree applications which utilize suffix links [3,31,32] have an alphabet factor in their running time anyway (if a linked list implementation of suffix trees is used). As a consequence, linear retrieval of suffix links does not influence the asymptotic running time, neither of suffix tree constructions, nor of suffix tree applications. Recall that linear retrieval of suffix links is required only for large nodes whose depth exceeds $2^\alpha - 1$. α will be chosen such that those nodes are usually very rare. If they occur, then the number of successors is expected to be small, and hence linear retrieval of suffix links is fast.

To guarantee constant time access from a small node b_i to the large node b_r, the records are stored in table T'_{branch}, ordered by the head positions of the corresponding branching nodes. All branching nodes are referenced by their *base address* in T'_{branch}, i.e. the index of the first integer of the corresponding record. Table II depicts table T'_{branch} for the suffix tree of Figure 1.

Space requirement

Suppose a base address can be stored in β bits. A reference is either a base address or a leaf number. To distinguish these, we need one extra bit. Thus a reference requires $1+\beta$ bits. Each

depth and each head position occupies $\gamma = \lceil \log_2 n \rceil$ bits. Consider the range of the distance values. In the worst case, take for example, $x = a^n$, there is only one chain of length $n-1$, i.e. the maximal distance value is $n-2$. However, this case is very unlikely to occur. To save space, we delimit the maximal length of a chain to 2^δ for some constant δ. As a consequence, after at most $2^\delta - 1$ consecutive small nodes an 'artificial' large node is introduced, for which we store a large record. In this way, we delimit the distance value to be at most $2^\delta - 1$, and thus the distance occupies δ bits. Thus we trade a delimited distance value for the saving of $\gamma - \delta$ bits for each small record.

A small record stores two references, a distance value, and one *nil bit* to mark a reference as a nil reference, which add up to $2 \cdot (1+\beta) + \delta + 1 = 3 + 2\beta + \delta$ bits. A large record stores two references, one nil bit, and one *complete bit* which tells whether the large node is complete. Moreover, there are γ bits required for the head position. If the record is complete then β bits are used for the suffix link and α bits for the depth. Otherwise, γ bits are used for the depth. We leave δ bits unused, i.e. they store the 'distance' 0. In this way, we can discriminate large and small nodes, since the latter always have a positive distance value. Altogether a complete large record requires $2 \cdot (1+\beta) + 1 + 1 + \gamma + \beta + \alpha + \delta = 4 + 3\beta + \gamma + \alpha + \delta$ and an incomplete large record requires $2 \cdot (1+\beta) + 1 + 1 + 2\gamma + \delta = 4 + 2(\beta + \gamma) + \delta$ bits.

To determine the actual space requirement we must choose the constants δ and α and consider the maximal length of the input string we allow. In our current implementation we assume $n \leq 2^{27} - 1$ (which implies $\gamma = 27$), and have chosen $\alpha = 10$ and $\delta = 5$. Then we can store a small record in two integers and reserve four integers for a large record.** As a consequence the maximal base address is $4n - 4$, and any base address is even. Hence $\beta = \gamma + 1$, which means that a small record requires $3 + 2\beta + \delta = 64$ bits, i.e. two integers. An incomplete large record requires $4 + 2(\beta + \gamma) + \delta = 119$ bits, and a complete large record requires $4 + 3\beta + \gamma + \alpha + \delta = 130$ bits. Both fit into four integers, if we store 2 bits for the latter type of record in $T_{leaf}[\overline{w}.headposition]$, where $\overline{w}$ is the corresponding node. Recall that $T_{leaf}[\overline{w}.headposition]$ stores a reference (29 bits) and one nil bit.

Let σ be the number of small records and λ be the number of large records. T'_{branch} requires $2\sigma + 4\lambda$ integers. Table T_{leaf} occupies n integers, and hence the space requirement of *ILLI* is $n + 2\sigma + 4\lambda$ integers. The implementation technique of McCreight [4] requires $2n + 5(\sigma + \lambda)$ integers. Each leaf and each large node saves one integer, and each small node saves three integers. Thus *ILLI* leads to large space savings, if there are many small nodes.

Example 1 Consider the input string $x = a^n$. Then *ST* has $n-2$ small nodes, one large node (i.e. $\overline{a}$), and $2n$ edges. Hence the space requirement of *ILLI* is $3n + \frac{1}{16}n$ integers (there are $\frac{2}{32}n$ extra integers required for the artificial large nodes). This is the best case. In contrast, the space requirement for *SLLI* is $6n$, which is the worst case. Hence *ILLI* requires about half of the space used by *SLLI*.

Example 2 Consider the input string $x = aabbabaaababbbaabaabb$ of length $n = 20$. Then *ST* contains 3 small nodes and 14 large nodes, and hence the space requirement is $(20 + 3 \cdot 2 + 15 \cdot 4)/20 = 4.3$ integers per input character. This is the largest space requirement of *ILLI* for all strings of length 20 over the alphabet $\Sigma = \{a, b\}$. The saving over *SLLI* is 24 integers.

The last example shows that there can be many large nodes. We conjecture that the upper bound on λ occurs for binary alphabets and that it is around $0.7n$, as in Example 2. However,

**We assume that each integer occupies 32 bits. Note that this does not imply that the word size of the computer is 32 bits. In fact, the software constructing our suffix tree representations works on 32 bit as well as on 64 bit machines without any changes.

we cannot prove this and have to calculate with an upper bound $\lambda \leq n$.

Note that the proposed suffix tree representations can be constructed in linear time without extra space, by a slight modification of McCreight's suffix tree algorithm [4]. The basic observation is that this algorithm constructs the branching nodes of *ST* in order of their head positions, which is compatible with our implementation techniques. For details, see Kurtz [18].

An improved hash table implementation

The redundancy of the information stored in the branching nodes can also be exploited to reduce the space requirement of the simple hash table implementation technique. In the improved hash table implementation technique, referred to by *IHTI*, we use *reference pairs* to address the nodes of *ST*. In particular, each leaf $\overline{S_j}$ is addressed by the reference pair $(0, j)$. Let $l_1, l_2, \ldots, l_p$ be the sequence of large nodes in *ST* ordered by their head position. This is a subsequence of $b_1, b_2, \ldots, b_q$, the sequence of branching nodes, as defined above. Consider the chain which ends with the large node l_i. l_i is referenced by the pair $(1, i)$. Each small node in this chain with distance $d > 0$ to l_i is referenced by the pair $(d+1, i)$. Using these reference pairs, it suffices to store the large records. The *suffixlink*, the *headposition* and the *depth* of each small node can be retrieved in constant time, according to Observation 7, since the distance to the large node of the chain is encoded in a reference pair. The hashing technique of the previous section only has to be slightly modified: consider the edge $\overline{w} \stackrel{av}{\rightarrow} \overline{wav}$, and suppose that $\overline{wav}$ is a node which is addressed by the reference pair p. Then one stores the pair $(\overline{w}.headposition, p)$ in the hash table using $(\overline{w}.headposition, a)$ as a hash key.

Another observation about the leaf edges allows us to reduce the size of the hash table considerably. Note that for a node $\overline{w}$ with head position i there is often a leaf edge $\overline{w} \stackrel{av}{\rightarrow} \overline{S_i}$. Let us call such a leaf edge *identity* edge.

Example 3 Consider the suffix tree for the string *abab*. Then there are two identity edges $\overline{ab} \stackrel{\$}{\rightarrow} \overline{ab\$}$ and $\overline{b} \stackrel{\$}{\rightarrow} \overline{b\$}$, which can be easily deduced from Table I.

There is at most one identity edge outgoing from each branching node. The observation is that it is not necessary to explicitly store identity edges in the hash table. We just need a single bit to mark that there is an identity edge outgoing from a branching node with head position i. Knowing this, we can deduce the leaf number i, the identity edge leads to, as well as the corresponding edge label, see Observation 4. For each $i \in [0, n]$, the ith entry of the hash table contains an unused bit. This can be used as a marking bit, so that no extra space is required to represent the identity edges. If we do not store the identity edges explicitly, then we can reduce the size of the hash table considerably. In fact, we have never found any input string for which the number of non-identity edges exceeds $1.5n$. Hence we only use a hash table of size $1.5n$. For the very unlikely situation that the hash table overflows, one can enlarge it and rehash all entries that are currently stored. Unfortunately, we cannot prove a worst case bound better than $2n$ for the size of the reduced hash table.

Space requirement

A reference pair is implemented by a single integer. We restrict the maximal length of the chains to 31 (one less than for *ILLI*). So the maximal distance value is 30, and the maximal

value in the left component of a reference pair is 31, i.e. it occupies 5 bits. This leaves 27 bits for storing the right component, i.e. the leaf number or the number i of a large node l_i. As a consequence, the length of the string to be processed is delimited to $2^{27} - 1 = 134{,}217{,}727$. The space requirement of *IHTI* is thus $4n + 3\lambda$ integers. Therefore, the saving over *SHTI* is 3σ integers.

Example 4 Consider the string $x = a^n$. Then *ST* contains $n - 2$ small and one large node. There are $2n$ edges, n of which are identity edges. Using a hash table of size $1.5n$, the space requirement for *IHTI* is $3n + \frac{3}{31}n$ integers, which is almost identical to the space requirement of *ILLI*. The saving over *SHTI* is almost $4n$ integers.

Example 5 Consider the input string $x =$ *aabbabaaabababbaabaabb* of Example 2. There are three small nodes, 14 large nodes, and 38 edges, 11 of which are identity edges. Using a hash table of size 30, the space requirement for *IHTI* is $(2 \cdot 30 + 3 \cdot 15)/20 = 5.25$ integers per input character. The reduction in the size of the hash table saves 10 integers, and the three small nodes save nine integers over *SHTI*.

UPPER LIMITS ON THE INPUT STRING LENGTH

Usually, the memory available delimits the maximal length of the input string which can be processed. However, on some computers with very large memory, one also has to take into account the available address space. This is delimited by $2^{\omega} - 1$, where ω is the word size of the computer. The worst case space requirement of *ILLI* is $5n$ integers. With the additional n bytes for representing the input string, the total space requirement is $21n$ bytes in the worst case. Since $21(2^{27} - 1) = 2{,}818{,}572{,}267$ is well below $2^{32} - 1$, we can safely assume that all bytes of the suffix tree representation of *ILLI* can be addressed on a computer with word size $\omega \geq 32$. In case strings longer than $2^{27} - 1$ are to be processed (and enough memory is available), one can modify *ILLI* such that each small and each large record contains an extra integer. This results in an implementation technique *ILLI*′ with a space requirement of $n + 3\sigma + 5\lambda$ integers.

The improved hash table implementation has a worst case space requirement of $7n$ integers. Since $29(2^{27} - 1) = 3{,}892{,}314{,}083$ is well below $2^{32} - 1$, all bytes can be addressed, whenever $\omega \geq 32$. For the other implementation techniques and compact *dawg*s, the upper limit on n is $(2^{\omega} - 1)/wcs$, where *wcs* is the worst case space requirement in bytes (including the n bytes for the input string). In practice, this number is actually smaller since there is a constant amount of memory required for the operating system and program execution. Table III gives an overview of the space requirements and the upper limits on n for $\omega = 32$.

EXPERIMENTS

For our experiments we collected a set of 42 files (total length 18,684,070) from different sources:

(a) We used 17 files from the Calgary Corpus and all 14 files from the Canterbury Corpus [33]. The Calgary Corpus usually consists of 18 files, but since the file *pic* is identical to the file *ptt*5 of the Canterbury Corpus, we did not include it here. Both corpora are widely used to compare lossless data compression programs.

Table III. Overview of the space requirement of different suffix tree implementation techniques and compact *dawg*s. n is the length of the input string, σ is the number of small nodes, and λ is the number of large nodes in the suffix tree. The worst case space requirement is calculated by substituting 0 for σ and n for λ. s is the number of states and t is the number of transitions in the compact *dawg*. The worst case occurs for the input string $x = a^{n-1}b$: then $s = n$ and $t = 2n - 2$. Since each state and each edge occupies three integers [6], the worst case space requirement of compact *dawg*s is $36n$ bytes. For calculating the upper bounds we added n bytes which are required to represent the input string

	Space (in integers)	Upper limit on n for $\omega = 32$
McCreight	$2n + 5(\sigma + \lambda)$	$(2^{32} - 1)/29 = 148,102,320$
SLLI	$n + 5(\sigma + \lambda)$	$(2^{32} - 1)/25 = 171,798,691$
SHTI	$4n + 3(\sigma + \lambda)$	$(2^{32} - 1)/29 = 148,102,320$
ILLI	$n + 2\sigma + 4\lambda$	$2^{27} - 1 = 134,217,727$
ILLI′	$n + 3\sigma + 5\lambda$	$(2^{32} - 1)/25 = 171,798,691$
IHTI	$3(n + \lambda)$	$2^{27} - 1 = 134,217,727$
Compact dawgs	$3(s + t)$	$(2^{32} - 1)/37 = 116,080,197$

(b) We added eight DNA sequences used by Lefévre and Ikeda [15]. These are denoted by their EMBL database accession number.
(c) We extracted a section of 500,000 residues from the PIR database, denoted by *PIR500*. The underlying alphabet is of size 20.
(d) We generated two random strings $R500k4$ and $R500k20$ of length 500,000 over an alphabet of size 4 and over an alphabet of size 20. The characters are drawn with uniform probability.

Space requirement

We compared the space requirement of the described implementation techniques with the space requirement of variants of directed acyclic word graphs [12,13]. To obtain concrete numbers, we developed software to compute *dawg*s. Given a *dawg*, it is fairly easy to compute the number of nodes and edges in the corresponding *position end-set tree* of Lefévre and Ikeda [15] (*pestry*, for short). The same holds for the *compact dawg* [14] (*cdawg*, for short). A tight implementation of a *dawg* and a *pestry*, based on linked lists, requires nine bytes for each node and eight bytes for each edge. For the *cdawg*, 12 bytes are required for each node and for each edge. These numbers are consistent with Crochemore and Vérin [6]. We also counted the number of small and large nodes in the suffix trees to calculate the space requirement for the different suffix tree implementation techniques, according to the formulas given in Table III.

Table IV shows the relative space requirement (in bytes per input char) of the *dawg*, the *pestry*, the *cdawg*, and for suffix trees using the implementation technique of McCreight (*McC*) and the improved implementation techniques *ILLI* and *IHTI* we propose. We emphasize that the given numbers refer to the space required for construction. It does *not* include the n bytes used to store the input string.

The first column of Table IV shows the name of the file and the second its source, as far as it has not been made precise above: CL stands for the Calgary Corpus, CN for the Canterbury Corpus, and EM for the EMBL data base. In addition, a single character denotes the type of

Table IV. Relative space requirement (in bytes/input char) of *dawg*s and suffix trees

File	*Source*	*Length*	*k*	*dawg*	*pestry*	*cdawg*	*McC*	*ILLI*	*IHTI*
book1	*CL/e*	768771	81	30.35	19.97	15.75	18.02	**9.83**	14.73
book2	*CL/e*	610856	96	29.78	17.92	12.71	18.63	**9.67**	14.13
paper1	*CL/e*	53161	95	30.02	18.12	12.72	18.92	**9.82**	14.17
paper2	*CL/e*	82199	91	29.85	18.53	13.68	18.51	**9.82**	14.42
paper3	*CL/e*	46526	84	30.00	19.00	14.40	18.28	**9.80**	14.53
paper4	*CL/e*	13286	80	30.34	19.50	14.76	18.35	**9.91**	14.66
paper5	*CL/e*	11954	91	30.00	18.86	14.04	18.41	**9.80**	14.46
paper6	*CL/e*	38105	93	30.29	18.28	12.80	19.07	**9.89**	14.19
alice29	*CN/e*	152089	74	30.27	18.90	14.14	18.63	**9.84**	14.38
lcet10	*CN/e*	426754	84	29.75	17.84	12.70	18.61	**9.66**	14.12
plrabn12	*CN/e*	481861	81	29.98	19.65	15.13	17.84	**9.74**	14.71
bible	*CN/e*	4047392	63	29.28	16.75	10.87	16.10	**7.27**	12.04
world192	*CN/e*	2473400	94	27.98	14.55	**7.87**	18.81	9.22	13.35
bib	*CL/f*	111261	81	28.53	15.88	9.94	18.76	**9.46**	13.73
news	*CL/f*	377109	98	29.48	17.58	12.10	18.41	**9.54**	14.06
progc	*CL/f*	39611	92	29.73	17.42	11.87	18.69	**9.59**	13.97
progl	*CL/f*	71646	87	29.96	16.27	**8.71**	20.98	10.22	13.55
progp	*CL/f*	49379	89	30.21	16.24	**8.28**	21.39	10.31	13.43
trans	*CL/f*	93695	99	30.47	15.97	**6.69**	22.22	10.49	13.21
fieldsc	*CN/f*	11150	90	29.86	16.39	**9.40**	19.81	9.78	13.59
cp	*CN/f*	24603	86	29.04	16.64	10.44	18.41	**9.34**	13.76
grammar	*CN/f*	3721	76	29.96	17.17	10.60	20.25	**10.14**	13.85
xargs	*CN/f*	4227	74	30.02	18.50	13.10	18.15	**9.63**	14.35
asyoulik	*CN/f*	125179	68	29.97	19.46	14.93	18.02	**9.77**	14.64
geo	*CL/b*	102400	256	26.97	19.09	13.10	13.41	**7.49**	13.99
obj1	*CL/b*	21504	256	27.51	16.68	13.20	14.53	**7.69**	13.61
obj2	*CL/b*	246814	256	27.22	14.23	**8.66**	18.81	9.30	13.46
ptt5	*CN/b*	513216	159	27.86	13.71	**8.08**	19.17	8.94	12.71
kennedy	*CN/b*	1029744	256	21.18	8.35	7.29	9.10	**4.64**	12.31
sum	*CN/b*	38240	255	27.79	14.85	10.26	17.65	**8.92**	13.58
ecoli	*CN/d*	4638690	4	34.01	27.34	23.55	20.84	**12.56**	17.14
J03071	*EM/d*	66495	4	33.70	20.47	13.44	24.14	**12.36**	14.85
K02402	*EM/d*	38059	4	34.12	27.60	23.90	20.83	**12.59**	17.18
M13438	*EM/d*	2657	4	33.95	27.59	23.96	20.65	**12.50**	17.16
M26434	*EM/d*	56737	4	34.10	26.51	22.52	21.38	**12.52**	16.75
M64239	*EM/d*	94647	4	34.10	27.60	23.94	20.87	**12.62**	17.20
V00636	*EM/d*	48102	4	34.02	27.75	24.04	20.72	**12.57**	17.22
V00662	*EM/d*	16569	4	34.14	27.61	24.10	20.90	**12.69**	17.29
X14112	*EM/d*	152261	4	34.13	27.12	23.43	21.12	**12.58**	17.00
PIR500		500000	20	30.35	22.70	15.79	17.51	**9.87**	15.09
R500k4		500000	4	33.93	28.06	24.15	20.44	**12.56**	17.38
R500k20		500000	20	29.83	27.98	20.06	14.93	**9.40**	15.94
Average relative space requirement for all files				30.33	19.78	14.55	18.82	10.10	14.66
Average relative space requirement for DNA				34.03	26.62	22.54	21.27	12.55	16.86

the file: *e* for english text, *f* for formal text (like programs), *b* for binary files (i.e. containing 8-bit characters), and *d* for DNA sequences. Columns three and four show the lengths and the alphabet sizes. In each row, the smallest relative space requirement is shown in boldface. The last two rows show the average relative space requirement for all files and for all DNA sequences. In the following, when we write space requirement we mean average relative space requirement.

There are some interesting findings which can be derived from Table IV. Since DNA sequences are very important for suffix tree applications, we in particular comment on the corresponding behavior of the considered data structures:

1. The *dawg* is the least space efficient data structure.
2. The *pestry* requires about 35 per cent less space than the *dawg*. For DNA sequences the saving over the *dawg* is 22 per cent. This is slightly smaller than the saving of 25–30 per cent reported by Lefévre and Ikeda [15].
3. A *cdawg* requires on average 26 per cent less space than a *pestry*. For DNA sequences the saving over the *pestry* is 15 per cent. As suggested by Crochemore and Vérin [6], the space requirement of the *cdawg* for DNA sequences can be reduced by using arrays instead of linked lists to represent outgoing edges. This results in an implementation referred to by *cdawgA*. It requires 20.66n bytes for the DNA sequences we used. This is consistent with the numbers given by Crochemore and Vérin [6]. For alphabets larger than four, *cdawgA* does not make sense. In some cases, in particular for formal texts, the *cdawg* is the most space efficient data structure. Note that its space requirement varies very much between 6.69n bytes and 24.15n bytes.
4. The suffix tree in the implementation following McCreight uses 30 per cent more space than the *cdawg*. However, for all DNA sequences, except for J03071, it requires less space than the *cdawg*. For DNA sequences the saving over the *cdawg* is 6 per cent, but it uses 3 per cent more space than the *cdawgA*.
5. The suffix tree in the improved linked list implementation is the most space efficient data structure. It improves over the *cdawg* by 30 per cent and over *McC* by 46 per cent. For all classes of files there is an advantage over the *cdawg*. However, there are seven files for which the *cdawg* requires less space. For DNA sequences the saving over the *cdawg* and the *cdawgA* is 44 per cent and 39 per cent, respectively. The space requirement varies between 4.64n bytes and 12.69n bytes. Thus the upper bound on the space requirement is 11.45n bytes smaller than the upper bound for the *cdawg*.
6. The suffix tree in the improved hash table implementation requires 45 per cent more space than the suffix tree in the improved linked list implementation. The space consumption is similar to the *cdawg* and it improves over *McC* by 22 per cent. For DNA sequences the space requirement is 16.86n bytes, which is an improvement over the *cdawg* and *cdawgA* of 25 per cent and 18 per cent. The space requirement varies between 12.31n bytes and 17.38n bytes.

Figure 3 presents the data of Table IV in a more compact way. For each type of file (except random) and each of the considered data structures and implementation techniques, a column shows the average relative space requirement for all files of that type. For DNA sequences we have seven columns, where the last column refers to *cdawgA*. It is obvious that the size of the data structures depends upon the kind of input data: binary strings lead to the smallest data structures, for formal text and english text all data structures are slightly larger. For protein sequences and in particular DNA sequences the space requirement is considerably higher.

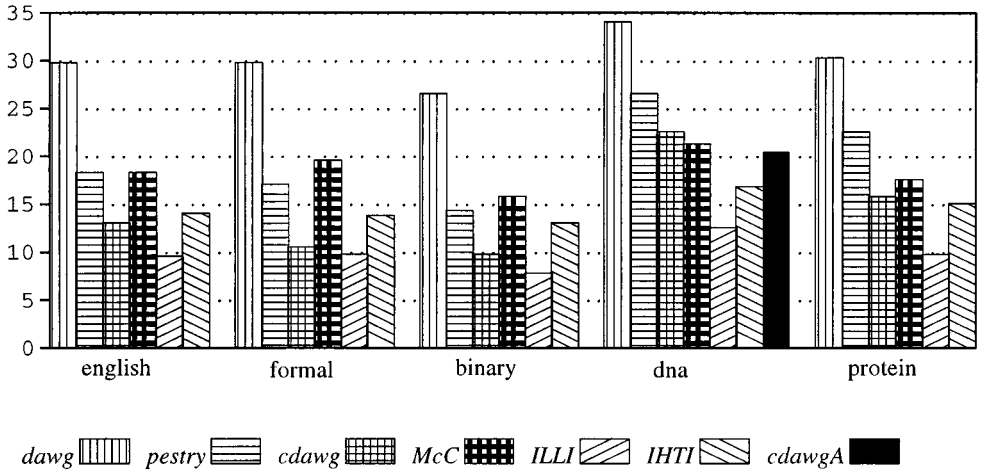

Figure 3. Relative space requirement (in bytes/input char) of dawgs and suffix trees for five groups of files

Running time

For our second experiment we implemented four different variants of McCreight's suffix tree construction named after the implementation technique they employ: *SLLI*, *SHTI*, *ILLI* and *IHTI*. All programs are written in C. We compiled our programs with the *gcc* compiler, version 2.8.1, with optimizing option –O3. The programs were run on a *Sun-UltraSparc*, 300 MHz, 192 megabytes RAM under Solaris 2. Table V shows the lengths of the files and the alphabet sizes. Columns 4–11 present the running times of the four programs: *absolute* user running *time* and *relative* user running time $rtime = (10^6 \cdot time)/n$, i.e. the time required to process 10^6 characters. Times are in seconds, as measured by the *gnu time* utility, averaged over 1000, 100 or 10 runs, depending on the size of the files. The last row shows the total running time and the average relative running time. In each line the smallest relative running time is shown in boldface.

Table V shows that the simple implementation techniques lead to slightly faster programs than the improved implementation techniques. However, the running time advantage of the simple implementation techniques are very small: 2 per cent for the linked list implementation, and 1.5 per cent for the hash table implementation. So, the additional constant overhead for the improved but more complicated implementation techniques are worth the effort.

Comparing the linked list implementations with the hash table implementations, one observes that the former are faster if the alphabet is small (i.e. $k \leq 100$) *and* if the input string is short (i.e. $n \leq 150{,}000$). We explain this as follows: the most time consuming part of the linked list implementation is traversing the list of successors of a particular node. Each such traversal step requires only a few very simple and fast operations. But the nodes accessed during such a traversal may be stored at very distant locations in memory, which means that McCreight's algorithm has a poor locality behavior [34]. If the text is short, then the entire suffix tree representation usually fits into the cache, so that cache misses are rare. So the poor locality does not matter, and hence the good performance of the linked list implementations for the case that the alphabet and the text are small. For larger files one clearly observes that the linked list implementation becomes slower, independent of the alphabet size.

Table V. Running times (absolute and relative in seconds) for different variants of McCreight's algorithm

			SLLI		*ILLI*		*SHTI*		*IHTI*	
File	*Length*	*k*	*Time*	*Rtime*	*Time*	*Rtime*	*Time*	*Rtime*	*Time*	*Rtime*
book1	768771	81	3.40	4.42	3.75	4.88	2.58	**3.35**	2.66	3.46
book2	610856	96	2.21	3.61	2.33	3.82	1.94	3.18	1.90	**3.10**
paper1	53161	95	0.10	**1.95**	0.11	2.05	0.13	2.41	0.12	2.20
paper2	82199	91	0.19	2.26	0.18	**2.22**	0.23	2.74	0.23	2.83
paper3	46526	84	0.09	2.02	0.09	**1.97**	0.11	2.30	0.10	2.20
paper4	13286	80	0.02	**1.71**	0.03	1.93	0.02	1.73	0.02	1.84
paper5	11954	91	0.02	1.81	0.02	**1.73**	0.02	**1.73**	0.02	1.98
paper6	38105	93	0.07	**1.79**	0.07	1.81	0.08	2.20	0.09	2.43
alice29	152089	74	0.43	2.85	0.41	**2.67**	0.44	2.91	0.42	2.74
lcet10	426754	84	1.44	3.37	1.46	3.42	1.33	3.11	1.27	**2.97**
plrabn12	481861	81	1.87	3.88	2.00	4.16	1.56	3.24	1.56	**3.23**
bible	4047392	63	15.33	3.79	15.70	3.88	14.01	3.46	13.55	**3.35**
world192	2473400	94	9.19	3.72	8.96	3.62	7.50	3.03	7.25	**2.93**
bib	111261	81	0.25	2.27	0.23	**2.10**	0.29	2.63	0.28	2.51
news	377109	98	1.82	4.84	1.79	4.74	1.13	3.00	1.09	**2.90**
progc	39611	92	0.07	**1.69**	0.07	1.80	0.09	2.18	0.08	1.97
progl	71646	87	0.11	**1.60**	0.12	1.64	0.19	2.72	0.17	2.43
progp	49379	89	0.07	1.45	0.07	**1.43**	0.12	2.46	0.11	2.14
trans	93695	99	0.15	1.62	0.15	**1.60**	0.27	2.85	0.23	2.47
fieldsc	11150	90	0.01	**1.04**	0.01	1.24	0.02	1.64	0.02	1.69
cp	24603	86	0.03	**1.39**	0.04	1.61	0.04	1.63	0.04	1.67
grammar	3721	76	0.004	**0.98**	0.004	1.18	0.006	1.62	0.006	1.63
xargs	4227	74	0.004	**1.06**	0.006	1.43	0.006	1.53	0.007	1.67
asyoulik	125179	68	0.32	**2.54**	0.32	2.58	0.36	2.84	0.34	2.72
geo	102400	256	0.69	6.75	0.68	6.69	0.21	**2.04**	0.23	2.28
obj1	21504	256	0.04	1.70	0.04	2.04	0.03	**1.43**	0.03	1.59
obj2	246814	256	0.97	3.91	0.98	3.97	0.72	2.93	0.68	**2.77**
ptt5	513216	159	0.91	1.77	0.88	**1.70**	1.44	2.80	1.22	2.37
kennedy	1029744	256	16.16	15.70	16.76	16.27	1.26	**1.22**	1.63	1.58
sum	38240	255	0.07	1.94	0.10	2.59	0.08	1.98	0.07	**1.72**
ecoli	4638690	4	17.73	3.82	18.07	3.90	17.39	**3.75**	20.47	4.41
J03071	66495	4	0.10	1.50	0.10	**1.45**	0.17	2.51	0.16	2.39
K02402	38059	4	0.05	**1.37**	0.06	1.51	0.08	2.01	0.08	2.22
M13438	2657	4	0.003	**1.02**	0.003	1.21	0.004	1.42	0.005	1.74
M26434	56737	4	0.09	**1.60**	0.09	1.62	0.13	2.35	0.14	2.43
M64239	94647	4	0.19	1.97	0.18	**1.92**	0.25	2.61	0.27	2.85
V00636	48102	4	0.07	**1.55**	0.08	1.63	0.11	2.23	0.12	2.41
V00662	16569	4	0.02	**1.19**	0.03	1.66	0.03	1.58	0.03	1.87
X14112	152261	4	0.34	2.25	0.34	**2.23**	0.42	2.78	0.47	3.09
PIR500	500000	20	3.07	6.14	3.22	6.44	1.62	**3.25**	1.68	3.35
R500k4	500000	4	1.69	3.37	1.56	**3.11**	1.56	3.12	1.82	3.63
R500k20	500000	20	3.68	7.36	3.90	7.80	1.48	**2.95**	1.62	3.24
			83.09	2.92	84.98	3.03	59.44	2.46	62.28	2.50

The dominating factor for the hash table implementations is the modulo operation (remember that we use division hashing). On the machine we used, this operation is slower than addition by an order of magnitude. However, for large files, the slow modulo operations are compensated for by the slow paging operations of the memory subsystem required by the linked lists implementation, so that the hash table implementations become faster.

CONCLUSION

Applications

The main topic of this paper was to show that suffix trees can be implemented in much less space than previously thought. This should make suffix trees more attractive for practical applications. Indeed, we have already used our implementation techniques in two applications:

(a) In a lossless data compression program [35], suffix trees in both improved implementation techniques are used to compute the Burrows and Wheeler Transformation [36] of a string in linear time and space. This basically means to sort all suffixes of a string in lexicographic order; see also Table IV, application 13.

(b) In a program called *REPuter* [37], suffix trees in the improved linked list implementation are used to compute maximal repeats in DNA sequences in optimal time. We used an algorithm described by Gusfield [3] (see Table IV, application 11). With the improved linked list implementation we are able to compute all 174,187 maximal repeats of length at least 20 contained in the entire yeast genome ($n = 12{,}147{,}818$) in 68 seconds, using 160 megabytes of space (*Sun-UltraSparc*, 300 MHz, 192 megabytes RAM). On the basis of the average relative space requirement for DNA sequences (see the section on 'Experiments'), we can estimate the corresponding space requirement for *cdawgA* and McCreight's implementation technique with $(20.66 \cdot 12{,}147{,}818)/2^{20} \approx 240$ megabytes and $(21.27 \cdot 12{,}147{,}818)/2^{20} \approx 246$ megabytes, respectively. This does not fit into the main memory of the machine we used, and so very time consuming memory swaps would be required. These are not necessary when *ILLI* is used.

A suffix tree for the human genome

We now develop a conjecture about the resources required for computing the suffix tree for the complete human genome. The size of the human genome is estimated to be about $3 \cdot 10^9$. We assume a computer with 64 bit architecture, so the address space is large enough. We modify implementation technique *ILLI* such that each small node is represented by three integers, each large node occupies 5 integers, and table T_{leaf} consists of $n + 1$ entries each occupying 1.25 integers. This leaves enough space to store references in the range $[0, 5n]$, where $n \leq 2^{32} - 1$. For the DNA sequences we used in our experiments, we determined the average ratios $\sigma/n = 0.258$ and $\lambda/n = 0.406$. We now assume that for the human genome, the same ratios hold. Based on this assumption we estimate that the suffix tree for the complete human genome will require about $3 \cdot 10^9 \cdot (1.25 + 3 \cdot 0.258 + 5 \cdot 0.406) = 1.22 \cdot 10^{10}$ integers or $4 \cdot 1.22 \cdot 10^{10}/2^{30} = 45.31$ gigabytes of main memory. Conservatively estimating that our program requires 10 seconds to process one million characters, we obtain a running time of about $3 \cdot 10^4$ seconds, which is less than nine hours. Given that sequencing of the complete human genome is probably not finished before December 2001 and taking into account the

Table VI. Applications of suffix trees and the kind of traversal they require. The numbers are the application numbers used by Gusfield [3]

	Applications with partial traversals		Applications with complete traversals
1	exact string matching	4	longest common substring of two strings
2	exact set matching	5	recognizing DNA contamination
3	substring problem for a data base of patterns	6	common substrings to more than two strings
8	computing matching statistics	7	building a smaller directed graph for exact matching
9	space efficient longest common substring problem	10	all pairs suffix prefix matching
15	Boyer-Moore approach to exact set matching	11	finding maximal repetitive structures
16	Ziv-Lempel data compression	12	circular string linearization
17	Minimum length encoding of DNA	13	computing suffix arrays

expected advances in hard-ware technology, it seems feasible to compute the suffix tree for the entire human genome on some computers. This would be very helpful for genomic research.

Pragmatics of the choice between ILLI and IHTI

We described two basic implementation techniques to implement suffix trees: linked lists and hash tables. The experiments suggest that the choice of the implementation technique depends on (i) the alphabet size, (ii) the length of the input string, and (iii) whether space requirement or running time is more important. However, there is another important point to consider: we have to take into account the way in which the suffix tree is utilized. There are basically two ways to utilize a suffix tree:

1. The suffix tree is partially traversed according to some given string, e.g. a pattern. This task requires to decide for a given node $\overline{w}$ and character a whether there is an a-edge outgoing from $\overline{w}$, and in case such an edge exists, to deliver this edge.
2. The suffix tree is traversed completely in a particular order. This task requires to have constant time access from one edge $\overline{v} \stackrel{aw}{\rightarrow} \overline{vaw}$ to another edge $\overline{v} \stackrel{cu}{\rightarrow} \overline{vcu}$ which has not been traversed before (if such an edge exists). Sometimes this edge has to be the next edge w.r.t. to some ordering on the first characters of the edge labels.

Partial traversals (see 1) are typical for pattern matching applications, and complete traversals (see 2) are typical for finding repetitive elements in strings. To give concrete examples we considered the first 17 applications of suffix trees given by Gusfield [3], and associated them according to whether they partially or completely traverse a suffix tree. Application 14 (i.e. suffix trees in genome scale projects) subsumes several different applications, and so it cannot uniquely be associated with any of the two kinds. In the remaining 16 applications we have found eight applications which perform partial traversals and eight applications which perform complete traversals. Table VI lists the applications and their association.

Partial traversals can be accomplished with both basic implementation techniques. However, when using a linked list implementation this leads to an alphabet factor in the running time. So, for partial traversals it is usually better to use a hash coded implementation, unless space is at the premium. Complete traversals can easily be accomplished with the linked list implementation. In contrast, the hash table implementation is less useful here,

since it does not immediately reveal the set of edges outgoing from some node. As noted by Larsson [38], it is possible to sort the hash table such that it allows complete traversals. In a first phase all edges stored in the hash table are sorted according to the nodes they are outgoing from. This can be done in time linear in the size of the hash table, i.e. in $O(n)$, using a bucket sort algorithm. This requires at most n extra integers to hold the counts for each node. After the first phase, for each node the edges outgoing from that node are stored in consecutive positions of the hash table. These can be sorted in $O(n)$ time altogether, again using a bucket sort algorithm. Thus the extra sorting phase requires $O(n)$ time and n integers of extra space. We have implemented such a sorting procedure for *IHTI*. Unfortunately, it proved to be slow in practice: the running time of the additional sorting phase is between 27 per cent and 140 per cent of the running time of the corresponding suffix tree construction (average 73 per cent). So, for complete traversals, the hash table representation is inferior, except when the alphabet is large.

Further improvements and analyses

We note that our implementation techniques are not optimized for a particular alphabet size. For DNA sequences, which lead to the largest index structures (see Figure 3), there are some further optimizations possible: if x is a DNA sequence, we can expect that each substring of length $q \leq \log_4 n$ over the DNA alphabet occurs at least twice. This means that most of the possible nodes of depth $\leq q - 1$ occur in the suffix tree, and these can be represented more space efficiently using a heap. A similar technique has already been applied for hashed position trees [16].

Finally, note that the proposed implementation techniques lead to some interesting combinatorial questions: what is the expected number of small and large nodes? Are there better worst case bounds for the number of large nodes? What is the largest/expected number of non-identity edges? Solutions to these problems definitely improve the acceptance of our implementation techniques.

ACKNOWLEDGEMENTS

The author is partially supported by DFG-grant Ku 1257/1-1. Bernhard Balkenhol suggested to further improve preliminary techniques to reduce the space requirement of suffix trees. Robert Giegerich, Jens Stoye, and Dirk Evers read previous versions of this paper and made suggestions to improve the presentation. All their contributions are truly appreciated.

REFERENCES

1. A. Apostolico, 'The myriad virtues of subword trees', *Combinatorial Algorithms on Words*, Springer-Verlag, 1985, pp. 85–96.
2. U. Manber and E. W. Myers, 'Sufix arrays: A new method for on-line string searches', *SIAM Journal on Computing*, **22**(5), 935–948 (1993).
3. D. Gusfield, *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, New York, 1997.
4. E. M. McCreight, 'A space-economical suffix tree construction algorithm', *Journal of the ACM*, **23**(2), 262–272 (1976).
5. J. Kärkkäinen, 'Suffix cactus: A cross between suffix tree and suffix array', *Proc. of the Annual Symposium on Combinatorial Pattern Matching (CPM'95), LNCS 937*, 1995, pp. 191–204.
6. M. Crochmore and R. Vérin, 'Direct construction of compact acyclic word graphs', *Proc. of the Annual Symposium on Combinatorial Pattern Matching (CPM'97), LNCS 1264*, 1997, pp. 116–129.

7. J. Knight, D. Gusfield and J. Stoye, 'The Strmat Software-Package', 1998. http://www.cs.ucdavis.edu/~gusfield/strmat.tar.gz
8. I. Munro, V. Raman and S. Srinivasa Rao, 'Space efficient suffix trees', *Proceedings of the 18th Conference on Foundations of Software Technology and Theoretical Computer Science*, Chennai, India, December 1998. *Lecture Notes in Computer Science 1530*, Springer-Verlag, 1998.
9. A. Andersson and S. Nilsson, 'Efficient implementation of suffix trees', *Software—Practice and Experience*, **25**(2), 129–141 (1995).
10. R. W. Irving, 'Suffix binary search trees', *Research Report*, Department of Computer Science, University of Glasgow, 1996. http://www.dcs.gla.ac.uk/~rwi/papers/sbst.ps
11. L. Colussi and A. De Col, 'A time and space efficient data structure for string searching on large texts', *Information Processing Letters*, **58**(5), 217–222 (1996).
12. A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen and J. Seiferas, 'The smallest automaton recognizing the subwords of a text', *Theoretical Computer Science*, **40**, 31–55 (1985).
13. M. Crochemore, 'Transducers and repetitions', *Theoretical Computer Science*, **45**, 63–86 (1986).
14. A. Blumer, J. Blumer, D. Haussler, R. McConnell and A. Ehrenfeucht, 'Complete inverted files for efficient text retrieval and analysis', *Journal of the ACM*, **34**, 578–595 (1987).
15. C. Lefévre and J.-E. Ikeda, 'The position end-set tree: A small automaton for word recognition in biological sequences', *Comp. Appl. Biosci.*, **9**(3), 343–348 (1993).
16. H. W. Mewes and K. Heumann, 'Genome analysis: pattern search in biological macromolecules', *Proc. of the Annual Symposium on Combinatorial Pattern Matching (CPM'95)*, *LNCS 937*, 1995, pp. 261–285.
17. P. Ferragina and R. Grossi, 'The string B-Tree: a new data structure for string search in external memory and its applications', *Journal of the ACM*, **46**(2), 236–280 (1999).
18. S. Kurtz, 'Reducing the space requirement of suffix trees', *Report 98–03*, Technische Fakultät, Universität Bielefeld, 1998. http://www.TechFak.Uni-Bielefeld.DE/techfak/~kurtz/publications.html
19. P. Weiner, 'Linear pattern matching algorithms', *Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory*, The University of Iowa, 1973, pp. 1–11.
20. E. Ukkonen, 'On-line construction of suffix-trees', *Algorithmica*, **14**(3), (1995).
21. M. Farach, 'Optimal suffix tree construction with large alphabets', *Proceedings of the 38th Annual Symposium on the Foundations of Computer Science, FOCS 97*, IEEE Press, New York, 1997. ftp://cs.rutgers.edu/pub/farach/Suffix.ps.Z
22. R. Giegerich and S. Kurtz, 'From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction', *Algorithmica*, **19**, 331–353 (1997).
23. N. J. Larsson, 'Extended application of suffix trees to data compression', *Proceedings of the IEEE Data Compression Conference*, IEEE Press, Snowbird, Utah, 1996.
24. A. Blumer, A. Ehrenfeucht and D. Haussler, 'Average size of suffix trees and DAWGS', *Discrete Applied Mathematics*, **24**, 37–45 (1989).
25. R. Giegerich, S. Kurtz and J. Stoye, 'Efficient implementation of lazy suffix trees', *Proc. of the Third Workshop on Algorithmic Engineering (WAE99)*, *LNCS 1668*, 1999, pp. 33–42.
26. D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.
27. J. G. Cleary, 'Compact hash tables using bidirectional linear probing', *IEEE Trans. on Computers*, **33**(9), 828–834 (1984).
28. J. J. Darragh, J. G. Cleary and I. H. Witten, 'Bonsai: a compact representation of trees', *Software—Practice and Experience*, **23**(3), 277–291 (1993).
29. J. I. Aoe, K. Morimoto and T. Sato, 'An efficient implementation of trie structures', *Software—Practice and Experience*, **22**(9), 695–721 (1992).
30. K. Morimoto, H. Iriguchi and J. I. Aoe, 'A method of compressing trie structures', *Software—Practice and Experience*, **24**(3), 265–288 (1994).
31. W. I. Chang and E. L. Lawler, 'Sublinear approximate string matching and biological applications', *Algorithmica*, **12**(4/5), 327–344 (1994).
32. S. Kurtz, 'Fundamental Algorithms for a Declarative Pattern Matching System', *Dissertation*, Technische Fakultät, Universität Bielefeld. (Available as Report 95–03, July 1995.)
33. R. Arnold and T. Bell, 'A corpus for the evaluation of lossless compression algorithms', *Proceedings of the Data Compression Conference*, 1997, pp. 201–210. http://corpus.canterbury.ac.nz
34. 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).

35. B. Balkenhol, S. Kurtz and Y. M. Shtarkov, 'Modification of the Burrows and Wheeler data compression algorithm', *Proceedings of the IEEE Data Compression Conference*, IEEE Press, Snowbird, Utah, 1999, pp. 188–197.
36. M. Burrows and D. J. Wheeler, 'A Block-Sorting Lossless Data Compression Algorithm', *Research Report 124*, Digital Systems Research Center, 1994.
http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-124.html
37. S. Kurtz and C. Schleiermacher, '*REPuter*: Fast computation of maximal repeats in complete genomes', *Bioinformatics*, **15**(5), 426–427 (1999).
38. N. J. Larsson, 'The context trees of block sorting compression', *Proceedings of the IEEE Data Compression Conference*, IEEE Press, Snowbird, Utah, 1998, pp. 189–198.

