Reducing the Space Requirement of Suffix Trees*

Stefan Kurtz†

Technische Fakultät, Universität Bielefeld
Postfach 100 131, 33501 Bielefeld, Germany
Email: kurtz@TechFak.Uni-Bielefeld.DE

December 28, 1998

Abstract

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 as efficiently as previous representations, and the asymptotic running times of suffix tree applications are retained.

1 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 [Apo85] gives over 40 references on suffix trees and Manber and Myers [MM93] add several more recent ones. A very thorough discussion of current knowledge on suffix tree constructions and applications can be found in [Gus97].

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 e.g. finite automata or hashing techniques. One of the main reasons for this is that suffix

*Report 98–03, Technische Fakultät, Universität Bielefeld
†partially supported by DFG-grant Ku 1257/1-1

trees have a reputation of being very greedy for space. In fact, the most space efficient previous implementation technique for suffix trees (see [McC76]) requires $28n$ bytes in the worst case where n is the length of the input string.[1] The space requirement in practice is smaller, but previous authors do not give consistent numbers:

- Manber and Myers [MM93] state that their implementation of suffix trees occupies between $18.8n$ and $22.4n$ bytes of space for real input strings (text, code, DNA).[2]
- Kärkkäinen [Kär95] 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.
- Crochemore and Vérin [CV97] state that suffix trees require $32.7n$ bytes for DNA sequences.
- The *strmat* software package by Knight, Gusfield, and Stoye [KGS98] implements suffix trees in $24n - 28n$ bytes for input sequences 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.

Faced with these numbers, 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 [MM93] requires $9n$ bytes (including the space for construction). The *level compressed trie* of [AN93] takes about $12n$ bytes. The suffix binary search tree of [Irv96] requires $10n$ bytes. Finally, the *suffix cactus* of [Kär95] can be implemented in $10n$ bytes. These four index structure have two properties in common: First, they are specifically tailored to solve string matching problems, and cannot not be adapted to other kinds of problems without severe performance penalties.[3] 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.[4]

Directed acyclic word graphs [BBH+85,Cro86], *dawg*s for short, and more space efficient variants thereof [BBH+87, LI93], have essentially the same applications as suffix trees, see [CV97]. The compact dawg, which is the most space efficient of these index structures, occupies $36n$ bytes in the worst case. Recently, Crochemore and Vérin [CV97] gave a direct method to construct *compact dawgs*, which makes this index structure useful in practice.

[1]We will use *bytes* or *integers* as units when we state results on space requirements. The assumption is always that an integer occupies 4 bytes. Unless stated otherwise, the given numbers do not include the n bytes for representing the input string.

[2]These numbers have been derived from the third column of Table 1 in [MM93, page 946]: We just added the space for the suffix links, which is $4q$ bytes where q is the number of internal nodes.

[3]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 4.

[4]All four 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.

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 [CV97], 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), several authors developed techniques to organize suffix trees on disk, so that the number of disk accesses is reduced [BRCR94, MH95, FG98]. 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 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 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 [McC76],[5] 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:

- In Section 3.1, we consider a simple linked list implementation technique which already improves by n integers over the best previous implementation technique of McCreight [McC76].
- In Section 3.2, we describe a simple hash table implementation technique, which improves on the method of McCreight [McC76].
- In Section 4.1, we make several observations about the node structure of suffix trees, which reveal redundancies of the information stored therein.
- In Sections 4.2 and 4.3, we show how to exploit these redundancies to improve the space requirement of the simple linked list implementation and the simple hash table implementation. The worst case space requirement of the improved linked list implementation is $5n$ integers, and it is probably even better but we can not prove this. Thus the improvement in the worst case is $2n$ integers over the most space efficient previous technique by McCreight [McC76]. It is interesting to note that for the string a^n where McCreight's techniques occupy $7n$ integers, both of our improved

[5] The paper by Andersson and Nilsson [AN93] actually considers level compressed tries (see above) which are different from suffix trees as introduced in [McC76].

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 $2^{27} - 1 = 134{,}217{,}727$.

- In Section 5, we show that all implementations of suffix trees have similar upper bounds on the maximal length of the input string they allow.
- In Section 6, we show how to modify the suffix tree construction of McCreight [McC76] such that it computes the space efficient representations we propose.
- In Section 7.1, 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% over the implementation technique of McCreight, and an improvement of 30% 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 structure, 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.
- In Section 7.2, we show that the space efficient representations we propose can be computed with virtually no performance penalty in practice. The linked list implementations proves to be faster than the hash table implementations only if the alphabet is small and the input string is short.
- In Section 8, 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.

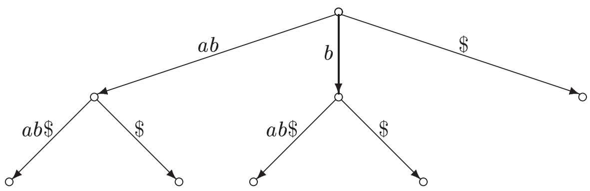
We start our description with a short introduction to the basic notions related to suffix trees.

2 Suffix Trees

2.1 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 $\Sigma^* \setminus \{\varepsilon\}$ 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 $x = x_1 x_2 \ldots x_n$.

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



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*.

$path(v)$ denotes the concatenation of the edge labels on the path from the *root* of T to the node v. 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 $path(v) = w$. The node $\overline{\varepsilon}$ is the *root*. Let $\overline{w}$ be a node 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. $words(T)$ denotes the set of strings occurring in T. Let $w \in words(T)$. A *location* of w in T is a pair $(\overline{u}, v)$ where $\overline{u}$ is a node in T and v is a string such that $w = uv$. The location $(\overline{u}, \varepsilon)$ is identified with the node $\overline{u}$. A location $(\overline{u}, v)$ of w in T is *canonical* if u is the longest prefix of w such that $\overline{u}$ is a node in T. The canonical location of w in T is denoted by $[w]_T$.

A Σ^+-tree is *compact* if every node is either the *root*, a leaf, or a branching node. A compact Σ^+-tree T is uniquely determined by $words(T)$.

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 compact Σ^+-tree T s.t. $words(T) = \{w \in \Sigma^* \mid w \text{ is a substring of } x\$\}$. Figure 1 shows the suffix tree for $x = abab$. ST can be constructed in $O(n)$ time using one of the algorithms described in [Wei73, McC76, Ukk95, Far97]. See also [GK97] which reviews [Wei73, McC76, Ukk95] and reveals relationships between these algorithms much closer that one would think.

For any $i \in [1, n+1]$, let $S_i = x_i \dots x_n\$$ 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 $x\$$. Such a label 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.

Let T be a compact Σ^+-tree. The *suffix link* for a node $\overline{aw}$ in T is an unlabeled directed edge in T from $\overline{aw}$ to the node $\overline{w}$, if the latter exists in T. We consider suffix links to be a part of suffix tree data structure. They are required for most of the linear time suffix tree constructions (see [Wei73, McC76, Ukk95]), and for some applications of suffix trees (see [Gus97]).

2.2 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.

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.

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

Proof: Let $\overline{w}$ be the *root*. Since $head_1 = head_{n+1} = \varepsilon$ and the *root* is branching the claim holds. Let $w = av$ for some $a \in \Sigma$ and some $v \in \Sigma^*$. Since $\overline{av}$ is branching, there are two different characters c and d such that avd and avc are substrings of $x\$$. Let $l = |av|$, $x' = x\$$, and $i \in [2, n]$ be the smallest position in x' such that $av = x'_i \ldots x'_{i+l-1}$, and there is a position $j \in [1, i-1]$ such that $av = x'_j \ldots x'_{j+l-1}$ and $x'_{i+l} \neq x'_{j+l}$. We say that i is the *first repetition* of av *witnessed by position* j. Suppose that avx'_{i+l} is a prefix of S_p for some $p \in [1, i-1]$. Hence $x'_i \ldots x'_{i+l-1}x'_{i+l} = x'_p \ldots x'_{p+l-1}x'_{p+l}$. This implies $x'_{p+l} = x'_{i+l} \neq x'_{j+l}$ and therefore $j \neq p$. Suppose that $p < j$. Then j is the first repetition of av witnessed by p, a contradiction. Suppose $j < p$. Then p is the first repetition of av witnessed by j, a contradiction. Hence avx'_{i+l} is not a prefix of S_p for some $p \in [1, i-1]$. That is, av is the longest prefix of S_i which occurs as a prefix of some longer suffix. Thus $av = head_i$. □

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

Proof: If $head_i = \varepsilon$, then the claim holds since the *root* is a branching node in ST. Let $head_i = av$ for some $a \in \Sigma$ and some $v \in \Sigma^*$. Then $i \in [2, n]$. By definition av is a prefix of S_i. Moreover, av is a prefix of S_j for some $j \in [1, i-1]$. Let $l = |av|$. Then $av = x_i \ldots x_{i+l-1} = x_j \ldots x_{j+l-1}$ and $x_{i+l} \neq x_{j+l}$. Since avx_{i+l} and avx_{j+l} are two different substrings of $x\$$, there must be a branching node $\overline{av} = \overline{head_i}$ 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.

3 Two Simple Implementation Techniques

The most space parsimonious implementation technique for suffix trees is based on linked lists [MM93]. McCreight [McC76, Fig. 4] showed how to represent ST using 5 integers for each internal node and 2 integers for each leaf. No extra space for the edges and their labels is required. Later authors gave the same numbers, e.g. [MM93, Irv96]. Recently, Crochemore and Vérin claimed that McCreight's implementation technique would also require 5 integers for each leaf [CV97, page 121]. This is not true. In the next section we show that one integer suffices for each leaf.

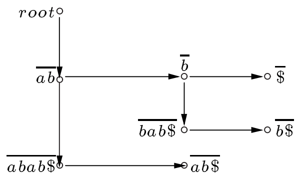
3.1 A Simple Linked List Implementation

We have found the following implementation technique to work well in practice: ST is represented 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 $T_{leaf}[j]$ 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

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.



in the subtree below node $\overline{wu}$. Hence wu is the prefix of S_j of length $\overline{wu}.depth$, i.e. the equality $wu = x_j \dots x_{j+\overline{wu}.depth-1}$ 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 we have $u = x_i \dots x_{i+l-1}$ 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 $u = x_i \dots x_n\$$ where $i = j + \overline{w}.depth$.

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 in [McC76]). At first, during the sequential suffix tree constructions of [Wei73, McC76, Ukk95], 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 Section 4. The third advantage of storing the depth is that several applications of suffix trees assume that the depth of a node is available, see [Gus97].

The references *firstchild*, *branchbrother*, and $T_{leaf}[j]$ 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 $b_1, b_2, \dots, b_q$ be the sequence of branching nodes ordered by their head position, i.e. $headposition(b_i) < headposition(b_{i+1})$ for any $i \in [1, q-1]$. Each branching node b_i is referred to by it's node number i, which is denoted by $nodenum(b_i)$. Obviously, b_1 is the *root*. Figure 3 depicts T_{leaf} and T_{branch} for the suffix tree of Figure 1.

Like the references, the other components of the branch records can 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 (*SLLI*, for short) is $n + 5q$ integers. The linked list implementation technique of McCreight requires $2n + 5q$ integers. Thus *SLLI* saves n integers.

Figure 3: The 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

In the worst case we have $q = n$, so that *SLLI* requires $6n$ integers. McCreight [McC76, page 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 requirement for each branch record to 4 integers, and the space complexity would be $5n$ integers, independent on 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, see [BEH89]), so that this worst case improvement would result in a larger space usage in practice. Therefore we do not further consider it.

Note that storing the nodes of the suffix tree in breadth first or depth first order to save the space for the brother or child references does not make sense here. This is because during suffix tree constructions the relation of the nodes change dynamically. More precisely, if a node is created, then it is not clear what the brother or child will be in the final suffix tree. So if we would store the nodes in breadth first or depth first order, the insertion of a new node would require an unbounded number of node movements. This is in contrast to the 3 or 4 updates of references required if we store the node relations as described above.

3.2 A Simple Hash Table Implementation

While linked list implementations of suffix trees are very 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 [McC76] 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 $\overline{w} \xrightarrow{av} \overline{wav}$ one stores the node number of $\overline{wav}$ in a hash table using the pair $(headposition(\overline{w}), a)$ 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 $\overline{w} \xrightarrow{av} \overline{wav}$, a reference to $\overline{wav}$ is delivered, as well as the edge label av. Notice that 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 in Section 3.1, 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 to resolve collisions [Knu73]. 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 [Knu73, sec. 6.4, p. 543, ex. 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 [Cle84], 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 open addressing technique we use is the better choice.

Since the space requirement of the hash table is still quite large, we considered compact hashing [Cle84] and the Bonsai implementation technique [DCW93] to save more space. Unfortunately, these method cannot be applied for suffix trees:

- Compact Hashing [Cle84] 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. Moreover, in order to sort the hash table, the hash key (or at least the part we stored) is required (see Section 8).

- The *Bonsai* implementation technique [DCW93] is based on Compact Hashing. It was specifically designed to represent trees very space efficiently. However, this technique requires a tree to be build from the root downward, a precondition which is not met by any of the linear time suffix tree construction methods in [Wei73, McC76, Ukk95, Far97].

4 Revealing and Exploiting Redundancies

4.1 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 $headposition(\overline{u}) \neq headposition(\overline{w})$.

(2) For any branching node $\overline{aw}$ in ST, $\overline{w}$ is also a branching node in ST and the inequality $headposition(\overline{aw}) + 1 \geq headposition(\overline{w})$.

Proof: Assume that $\overline{u}$ and $\overline{w}$ are branching nodes in ST such that $headposition(\overline{u}) = headposition(\overline{w})$. We have $u = head_{headposition(\overline{u})} = head_{headposition(\overline{w})} = w$ and therefore $\overline{u} = \overline{w}$. Thus (1) holds. Now suppose that $\overline{aw}$ is a branching node in ST. Then there are two different characters c and d such that awd and awc are substrings of $x' = x\$$. Hence wc and wd are also substrings of x'. This implies that $\overline{w}$ is a branching node in ST. Let $headposition(\overline{aw}) = i$ and j be the minimal position in x where aw occurs. Thus $x'_{i+l} \neq x'_{j+l}$ where $l = |aw|$. Let $headposition(\overline{w}) = i'$ and j' be the minimal position in x' where w occurs. For any $p \in [j', i'-1]$ such that $w = x'_p \dots x'_{p+l-2}$ we have $x'_{j'+l-1} = x'_{p+l-1}$. Obviously $j' \leq j+1 \leq i$. Suppose that $i' > i+1$. Then $j'+1 \leq i+1 \leq i'-1$ and $j' \leq j+1 \leq i \leq i'-2$, i.e. both $i+1$ and $j+1$ are in the range $[j', i'-1]$. Moreover, we have $w = x'_{i+1} \dots x'_{i+l-1} = x'_{j+1} \dots x'_{j+l-1}$. Hence $x'_{i+l} = x'_{i+1+l-1} = x'_{j'+l-1} = x'_{j+1+l-1} = x'_{j+l}$. This is a contradiction. Thus $i' \leq i+1$, which completes the proof. □

Observation 5 implies that for any branching node $\overline{aw}$ we either have $headposition(\overline{aw}) + 1 = headposition(\overline{w})$ or $headposition(\overline{aw}) > headposition(\overline{w})$. We discriminate all non-root nodes accordingly: $\overline{aw}$ is a *small* node if and only if $headposition(\overline{aw}) + 1 = headposition(\overline{w})$. $\overline{aw}$ is a *large* node if and only if $headposition(\overline{aw}) > headposition(\overline{w})$. 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.

Proof: Suppose that $\overline{aw}$ is small. By Observation 5, $\overline{w}$ exists in ST, and by Definition $headposition(\overline{aw}) + 1 = headposition(\overline{w})$. Let $i = nodenum(\overline{aw})$. Then we have $b_i = \overline{aw}$ and therefore $headposition(b_i) = headposition(\overline{aw}) < headposition(b_{i+1})$. This implies $headposition(\overline{w}) = headposition(\overline{aw}) + 1 \leq headposition(b_{i+1})$. Suppose that $\overline{w} \neq b_{i+1}$. Then $headposition(\overline{w}) \neq headposition(b_{i+1})$, and hence $headposition(b_i) =$

$headposition(\overline{aw}) < headposition(\overline{w}) < headposition(b_{i+1})$. This contradicts our numbering scheme for branching nodes. Thus $\overline{w} = b_{i+1}$, which implies $nodenum(\overline{aw}) + 1 = i + 1 = nodenum(\overline{w})$. To show (2), $nodenum(\overline{aw}) = q$ and $q > 1$. That is, $\overline{aw}$ is the last node and it is not the *root*. Suppose $\overline{aw}$ is small. Then by (1), $q+1 = nodenum(\overline{aw})+1 = nodenum(\overline{w})$. However, there is no branching node with node number $q + 1$. So $\overline{aw}$ is large. □

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

- b_{l-1} is not a small node.
- $b_l, \dots, b_{r-1}$ are small nodes.
- 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 $b_l, \dots, b_r$ 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) $b_i.suffixlink = b_{i+1}$

Proof: Let $i \in [l, r - 1]$. Since $i \geq l \geq 2$, b_i is not the *root*. Hence $b_i = \overline{aw}$ for some $a \in \Sigma$ and some $w \in \Sigma^*$. Since $\overline{aw}$ is small, we have $i + 1 = nodenum(\overline{aw}) + 1 = nodenum(\overline{w})$ by Observation 6. That is, $\overline{w} = b_{i+1}$. This immediately implies (3). Moreover, we have $b_i.depth = b_{i+1}.depth+1$, and $b_i.headposition = b_{i+1}.headposition-1$. By a simple induction on $r - i$ one shows properties (1) and (2). □

4.2 An Improved Linked List Implementation

We now show how to exploit the above stated redundancies to store the information in the branching nodes in less space. Consider a chain $b_l, \dots, b_r$. 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 $b_i.distance = r - i$ of the small node b_i to the large node b_r is known, then $b_i.depth$ and $b_i.headposition$ can be obtained in constant time. Note that the distance values are strictly positive.

Our improved linked list implementation technique, called *ILLI* in the sequel, stores two different kinds of records for the branching nodes:

- For any small node $\overline{w}$ there is a *small record* which stores $\overline{w}.distance$, $\overline{w}.firstchild$, and $\overline{w}.branchbrother$.

- For any large node $\overline{w}$ (and the *root*) there is a *large* record which stores $\overline{w}.\mathit{firstchild}$, $\overline{w}.\mathit{branchbrother}$, $\overline{w}.\mathit{depth}$, and $\overline{w}.\mathit{headposition}$. It also stores $\overline{w}.\mathit{suffixlink}$, whenever $\overline{w}.\mathit{depth} \leq 2^{10} - 1$.

We now show that a small record can be stored in 2 integers and a large record in 4 integers. The general assumption is that an integer occupies 32 bits or 4 bytes.

Consider the range of the distance values. In the worst case, take e.g. $x = a^n$, 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. Indeed, experiments show that the expected distance of the small nodes is usually much smaller, and only rarely exceeds 32 (this has nothing to do with the number of bits in an integer). So to save space, we delimit the maximal length of a chain to 32. As a consequence, after at most 31 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 31, and thus the distance occupies 5 bits, which are stored in one of the two integers used for a small record, see Figure 4. Thus we trade a delimited distance value for the saving of one integer for each small record.

Now let us consider how to store the values of the large record. The first two integers of a large record store the *firstchild* reference and the *branchbrother* reference, as in the small record. Some observations about the values stored in the suffix tree allow to use just 2 extra integers to store the remaining values of a large record: Consider some large node, say $\overline{w}$, and let $\overline{v}$ be the rightmost child of $\overline{w}$. There is a sequence consisting of one *firstchild* reference and at most $k - 1$ *branchbrother*/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 $\overline{v}.\mathit{branchbrother}$ 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 we store $\overline{w}.\mathit{suffixlink}$. 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 (see Section 6). Moreover, suffix tree applications which utilize suffix links (see [CL94, Kur95, Gus97]) 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. However, experiments show that it may slow down the suffix tree constructions in practice. For this reason, we use the following method which makes linear retrieval of suffix links an exception: Note that in the usual case, the branching nodes with many successors are close to the *root*, i.e. their depth is small. For these nodes linear traversal of suffix links is slow. However, we only need a few bits to store their depth. Thus the remaining bits can be used to store the suffix link, which can later be retrieved in constant time. For large nodes whose depth exceeds $2^{10} - 1 = 1023$, linear traversal of suffix links is required. In Figure 4 we have depicted a possible bit layout for the integers of the small and the large records which is consistent with the presented ideas.

Figure 4: The bit layout for the integers of a small and a large record. For both kinds of records we use one integer with layout (1) and one integer with layout (2). For a small record, the first 5 bits of integer (1) store a distance value in the range $[1, 31]$. For a large record, the first 5 bits of integer (1) store the value 0. In this way, small records and large records can uniquely be distinguished. The remaining 27 bits of integer (1) and the first two bits of integer (2) are reserved for the *firstchild* reference. The bit marked by $\downarrow$ in integer (2) is the *nil bit*. Suppose that the record, small or large, stores the information for a node $\overline{v}$. If the nil bit is set, then $\overline{v}$ is the rightmost child of a large node $\overline{w}$, and the remaining 29 bits store the *suffixlink* of $\overline{w}$. If the nil bit is not set, then the remaining 29 bits store the *branchbrother* reference. A large record additionally contains two integers: One integer with layout (3) and one integer with layout (4). If the first bit of integer (3) (marked by $\downarrow$) is set, the depth of the large node is at most $2^{10} - 1 = 1023$. In this case, the 10 lower bits of integer (3) store the distance value. The remaining 21 bits of integer (3) and the first 5 bits of integer (4) store the suffix link. If the first bit of integer (3) is not set, then the depth of the large node exceeds $2^{10} - 1 = 1023$. In this case, the lower 27 bits of integer (3) store the depth. The lower 27 bits of integer (4) always store the head position.

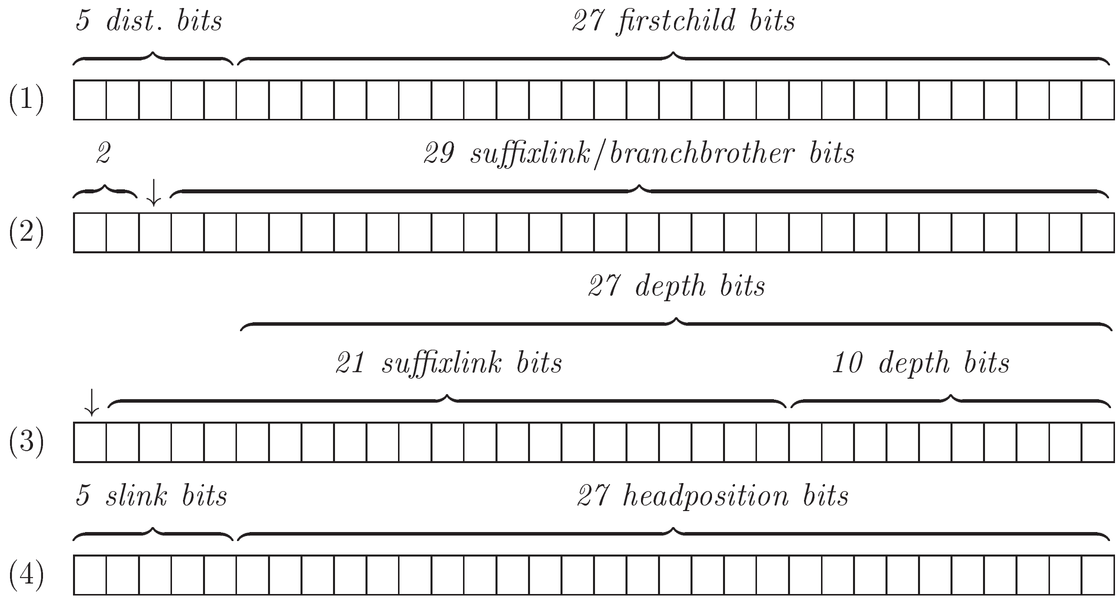


Figure 5: The table T'_{branch} for the suffix tree for $x = abab$ (see Figure 1). $\overline{ab}$ is a small node with head position 3, and $\overline{b}$ is a large node with head position 4. Both form a chain. The distance of $\overline{ab}$ and $\overline{b}$ is 1, depicted as a tiny 1 in the small record for $\overline{ab}$. Consider the large record for $\overline{b}$: the tiny 0 stands for the unused most significant 5 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 table T_{leaf}.

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

To guarantee constant time access from a small node b_i to the large node b_r at the end of the chain b_i belongs to (see Observation 7), all records are stored in the same table T'_{branch}. Just as in *SLLI*, the records are ordered by the head positions of the corresponding branching nodes. All branching nodes are referred to by their *base address* in T'_{branch}. The base address is the index of the first integer of the corresponding record. Let σ be the number of small records and λ be the number of large records. For the *root* we store a large record. Thus table T'_{branch} requires $2\sigma + 4\lambda$ integers, and hence the space requirement of implementation technique *ILLI* is $n + 2\sigma + 4\lambda$ integers. Figure 5 depicts table T'_{branch} for the suffix tree of Figure 1.

Each small record saves 3 integers and each large record saves 1 integer over *SLLI*. Thus *ILLI* leads to large space savings, if there are many small nodes. For some concrete numbers see the following examples.

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 $x = aabbabaaababbaabaabb$ of length $n = 20$. Then ST contains 3 small nodes and 14 large nodes, and hence the space requirement is $(4(20 + 3 \cdot 2 + 15 \cdot 4))/20 = 17.2$ bytes 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 cannot prove this and therefore have to calculate with an upper bound $\lambda \leq n$.

Now we show that *ILLI* allows to process input strings of length up to $2^{27} - 1 =$ 134,217,727 (if there is enough memory available). Let $n \leq 2^{27} - 1$. According to the bit layout of Figure 4, we have 29 bits available for a reference. At first we need one bit to mark whether a reference is a leaf number or the base address of a branching node. Since $\lambda \leq n$, and each large record consists of 4 integers, the size of table T'_{branch} is at most $2^{29} - 1$. Hence the maximal base address is smaller than 2^{29}. Each base address is even. Thus it can be stored in 28 bits. The leaf numbers are in the range $[1, 2^{27}]$, and so we need 27 bits to address a leaf. A suffix link requires 28 bits. In case the depth of a large node, say $\overline{w}$, is at most $2^{10} - 1$, 26 of these bits are stored in the third and fourth integer of the corresponding large record (see Figure 4). The remaining two bits of the suffix link are stored as the most significant bits of the integer $T_{leaf}[\overline{w}.headposition]$. Note that this stores a nil bit and a reference which occupies 29 bits. Thus there are 2 unused bits. Finally, note that the depth and the head position are always $\leq 2^{27} - 1$, so 27 bits are sufficient to store these values.

4.3 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.

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 $2^{27} - 1 = 134{,}217{,}727$.

Note that the hashing technique, as described in Section 3.2, only has to be slightly modified: Consider the edge $\overline{w} \xrightarrow{av} \overline{wav}$ and suppose that $\overline{wav}$ is a node which is addressed by the reference pair p. Then one stores the pair of integers $(headposition(\overline{w}), p)$ in the hash table using $(headposition(\overline{w}), a)$ as a hash key.

Another observation about the leaf edges allows 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 $\overline{w} \xrightarrow{av} \overline{S_i}$. 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 $\overline{ab} \xrightarrow{\$} \overline{ab\$}$ and $\overline{b} \xrightarrow{\$} \overline{b\$}$, which can be easily deduced from Figure 3.

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 occupied 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.

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 $x = aabbabaaababbaabaabb$ of Example 2. There are 3 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 $(4 \cdot (2 \cdot 30 + 3 \cdot 15))/20 = 21$ bytes per input character. The reduction in the size of the hash table saves 10 integers, and the 3 small nodes save 9 integers over *SHTI*.

5 Upper Limits on the Input Sequence Length

Usually, the available memory 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 $\omega \geq 32$. The upper bound of $2^{27} - 1$ on n is rather liberal, and should suffice for most applications of suffix trees. In case longer strings have 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,

Table 1: Overview of the space requirement of different suffix tree implementation techniques and compact dawgs. 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 3 integers (see [CV97]), the worst case space requirement of compact dawgs is $36n$ bytes. For calculating the upper bounds we added n bytes which are required to represent the input string. The upper bounds for *ILLI* and *IHTI* are due to the bit layout we use.

	space (in integers)	upper bound 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	$3(s + t)$	$(2^{32} - 1)/37 = 116{,}080{,}197$

whenever $\omega \geq 32$. For the other implementation techniques and compact dawgs, 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 1 gives an overview of the space requirements and the upper limits on n for $\omega = 32$.

6 Modifying McCreight's Algorithm

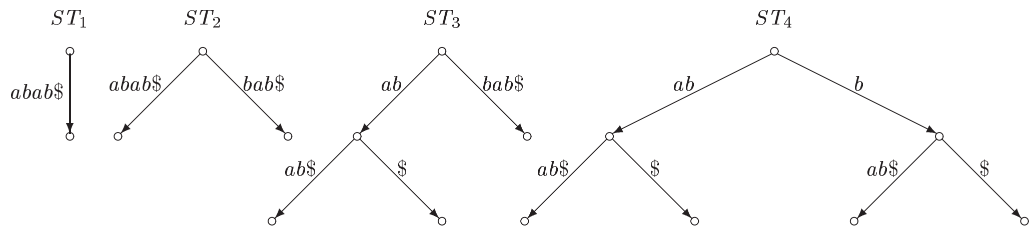
Until now we considered space efficient representations of suffix trees. Now we show how these representation can be computed in linear time, by a modification of McCreight's suffix tree construction [McC76]. In particular, we show how to decide whether a branching node is small or large, and how to set the distance of a small node. Our description abstracts from whether we use a linked list implementation or a hash table implementation.

Let ST_0 be the empty Σ^+-tree. This consists of the *root* only. For any $i \in [1, n+1]$, let ST_i be the compact Σ^+-tree such that

$$words(ST_i) = \{w \in \Sigma^* \mid w \text{ is a prefix of } S_j \text{ for some } j \in [1, i]\}$$

Notice that $ST_{n+1} = ST$. Let $i \in [1, n+1]$. Recall that $head_i$ is a prefix of S_i. $tail_i$ denotes the remaining suffix of S_i, i.e. $tail_i$ satisfies $head_i tail_i = S_i$. We obviously have $tail_{n+1} = \$$

Figure 6: The sequence of Σ^+-Trees constructed by *mcc* for $x = abab$. The last tree, ST_5, is omitted, since it is identical to the suffix tree shown in Figure 1.



and $tail_i \neq \varepsilon$ for any $i \in [1, n+1]$. By Observations 1 and 2, there is a branching node $\overline{v} \neq root$ in ST_i if and only if $v = head_j \neq \varepsilon$ for some $j \in [2, i]$.

The *head*'s and *tail*'s are represented by pointers into the tree and the input string x. We therefore define $headloc_i = [head_i]_{ST_{i-1}}$ and $tailptr_i \in [1, n+1]$ such that $tail_i = x_{tailptr_i} \ldots x_n\$$ for any $i \in [1, n+1]$.

The general structure of McCreight's algorithm is to construct ST by successively inserting the suffixes of $x\$$ into an initially empty tree, from longest to shortest. More precisely, the algorithm constructs the following sequence of compact Σ^+-trees:

$$ST_1, ST_2, \cdots, ST_n, ST_{n+1} \tag{1}$$

Figure 6 shows this sequence for $x = abab$. Additionally, with each ST_i, $i \in [1, n]$, the algorithm constructs the suffix links for all nodes $\overline{head_j} \neq root$, $j \in [2, i-1]$. In other words, the only branching node in ST_i whose suffix link is possibly not constructed, is node $\overline{head_i}$. Note furthermore that there is no suffix link for the *root*.

ST_1 is the compact Σ^+-tree with only one edge $root \xrightarrow{x\$} \overline{S_1}$. This can easily be constructed in constant time. In order to compute ST_i for $i \in [2, n+1]$, it is crucial to compute $headloc_i$ and $tailptr_i$. Once this has been done, ST_i can be constructed by inserting one or two edges, one leaf, and at most one branching node into ST_{i-1} as follows:

(a) If $headloc_i = \overline{u}$ for some node $\overline{u}$ in ST_{i-1}, then $u = \varepsilon$ or $\overline{u}$ is a branching node in ST_{i-1} such that $u = head_i = head_j$ for some $j \in [2, i-1]$. In this case, a new leaf edge $\overline{u} \xrightarrow{tail_i} \overline{S_i}$ is inserted into ST_{i-1}. We assume that this is implemented by a function *insertleaf*.

(b) If $headloc_i = (\overline{u}, av)$ for some node $\overline{u}$ in ST_{i-1}, some character $a \in \Sigma$, and some string $v \in \Sigma^*$, then $\overline{head_i}$ does not exist in ST_{i-1}. Thus it has to be constructed. There is an a-edge $\overline{u} \xrightarrow{avw} \overline{uavw}$ in ST_{i-1} for some $w \in \Sigma^+$. $\overline{head_i} = \overline{uav}$ is constructed by splitting this edge into $\overline{u} \xrightarrow{av} \overline{uav} \xrightarrow{w} \overline{uavw}$, and inserting a new leaf edge $\overline{uav} \xrightarrow{tail_i} \overline{S_i}$. We assume that this is implemented by a function *insertbranch*.

Note that in case (b) the node $\overline{head_i}$ with head position i is created. This means that the branching nodes in ST are created in order of their head position, and thus in the order in which they are stored in table T'_{branch}. This is the main reason why McCreight's algorithm can be used for constructing the proposed suffix tree representations of *ILLI* and *IHTI*. In [GK97] it is shown that Ukkonen's suffix tree algorithm constructs the nodes and edges of the suffix tree in the same order as McCreight's algorithm. As a consequence, one could instead use Ukkonen's algorithm. We prefer McCreight's algorithm, since it is slightly faster than Ukkonen's algorithm, as shown in [GK95].

It is easy to see that the functions *insertleaf* and *insertbranch* can be implemented in constant time. Thus it remains to show how to

- decide whether a branching node is small or large,
- set the distance of a small node,
- determine $headloc_i$ and $tailptr_i$ in constant amortized time.

Consider the following cases:

(1) If $head_{i-1} = \varepsilon$, then one scans a maximal prefix of $S_i = x_{tailptr_{i-1}+1} \dots x_n\$$ in ST_{i-1} down from the *root*, i.e. one computes $(headloc_i, tailptr_i) := scanprefix(ST_{i-1}, root, tailptr_{i-1}+1)$ where the function *scanprefix* is specified as follows: Let $\overline{s}$ be a node in some compact Σ^+-tree T, and $j \in [1, n+1]$. Then $scanprefix(T, \overline{s}, j) = ([su]_T, j+|u|)$ where u is the longest prefix of $x_j \dots x_n\$$ such that $su \in words(T)$.

(2) If $head_{i-1} = x_{i-1}w$ for some $w \in \Sigma^*$, then one exploits the fact that w is a prefix of $head_i$ (see [McC76]) and that the location of w can be determined in constant amortized time via the suffix links in ST_{i-1}. The details are as follows:

   (1) If $headloc_{i-1}$ is a node in ST_{i-1}, then the suffix link $headloc_{i-1} \longrightarrow \overline{w}$ in ST_{i-1} is already constructed. One follows this suffix link to the node $\overline{w}$ which is on the path from the *root* to $headloc_i$. To obtain $headloc_i$ and $tailptr_i$, one scans a prefix of $tail_{i-1}$ down from $\overline{w}$, i.e. one computes $(headloc_i, tailptr_i) := scanprefix(ST_{i-1}, \overline{w}, tailptr_{i-1})$.

   (2) If $headloc_{i-1}$ is not a node in ST_{i-1}, then $\overline{head_{i-1}}$ is the last branching node which was created. Obviously, it is a node in ST_i whose suffix link has not been constructed yet. $headloc_{i-1}$ is of the form $(\overline{u}, av)$ for some node $\overline{u}$ in ST_{i-1}, some $a \in \Sigma$, and some $v \in \Sigma^*$. One now computes $[w]_{ST_{i-1}}$ according to the following two cases:

      (1) If $\overline{u}$ is the *root*, then $av = x_{i-1}w$ and v is a prefix of $head_i$. One obtains $[w]_{ST_{i-1}}$ by scanning v down from the *root*. Note that since v occurs in ST_{i-1}, it suffices to *rescan* v, i.e. one only compares the first character of an edge label with the corresponding character in w, in order to decide which edge to follow. We assume that this is implemented by a function *rescan*.

      (2) If $\overline{u}$ is not the *root*, then $u = x_{i-1}y$, and the suffix link $\overline{u} \longrightarrow \overline{y}$ for $\overline{u}$ in ST_{i-1} is already constructed. One follows this suffix link to node $\overline{y}$ which is on the path from the *root* to $headloc_i$. Since $w = yav$, one computes $[w]_{ST_{i-1}}$ by rescanning av down from $\overline{y}$.

Now $[w]_{ST_{i-1}}$ is computed and one can decide whether the previous node $\overline{head_{i-1}}$ is small or large:

(3) If $[w]_{ST_{i-1}}$ is a node in ST_{i-1}, then its head position is smaller than $i-1$. Hence $\overline{head_{i-1}}$ is a large node with the suffix link $\overline{head_{i-1}} \longrightarrow [w]_{ST_{i-1}}$. Suppose that the chain ending with $\overline{head_{i-1}}$ is of length p and that the base address of $\overline{head_{i-1}}$ is q. Then for each $d \in [1, p-1]$, the node at base address $q-2d$ is small and its distance to $\overline{head_{i-1}}$ is d. Note furthermore that the base address of the next node to be constructed is $q+4$. It remains to determine $headloc_i$ and $tailptr_i$: this is done by scanning $tail_{i-1}$ down from $[w]_{ST_{i-1}}$, i.e. one computes $(headloc_i, tailptr_i) := scanprefix(ST_{i-1}, [w]_{ST_{i-1}}, tailptr_{i-1})$.

(4) If $[w]_{ST_{i-1}}$ is a not a node in ST_{i-1}, then $head_i = w$ (see [Kur95]). Thus $headloc_i = [w]_{ST_{i-1}}$ and $tailptr_i = tailptr_{i-1}$. This means that with the next call of the function *insertbranch*, a new branching node $\overline{head_i}$ will be constructed. Its head position is i and the head position of $\overline{head_{i-1}}$ is $i-1$. Thus $\overline{head_{i-1}}$ is a small node. It remains to note that the base address for the next branching node is $q+2$. Once the end of the current chain is found (case (2.2.3)), $\overline{head_{i-1}}.distance$ can be set.

Figure 7 gives pseudo code for the modified suffix tree construction of McCreight.

As shown in [McC76], McCreight's construction runs in $O(kn)$ time if a linked list implementation is used, and in $O(n)$ time if a hash table implementation is used. Now consider the additional statements we have included for handling the creation of large and small nodes (see statements between $p := p+1$ and case (2.2.4)). The only critical part is the for-loop over all $d \in [1, p-1]$, which establishes the correct distances for the small nodes of the current chain. Since for each small node the distance is only set once, and there are $O(n)$ small nodes, the extra running time of the for-loop is linear in n. Thus the modified algorithm still runs in $O(kn)$ and $O(n)$ time, respectively.

The algorithm of Figure 7 can easily be modified to accommodate the insertion of artificial large nodes: In (2.2.4) one just has to handle the case $p = 31$ (i.e. the current chain reaches it's maximal allowed length) as follows: the suffix link of $\overline{head_{i-1}}$ is set to the node created next, and the distance of all small nodes of the current chain is set as in case (2.2.3).

As one can see the required modifications of McCreight's Algorithm are not to difficult and practice shows that they are worth the effort.

7 Experiments

For our experiments, we collected a set of 42 files from different sources:

- We used 17 files from the Calgary Corpus [BCW90] and all 14 files from the Canterbury Corpus [AB97]. The Calgary Corpus usually consists of 18 files, but since the file *pic* is identical to the file *ptt5* of the Canterbury Corpus, we did not include it here.

Figure 7: McCreight's suffix tree construction with modifications to compute the space efficient suffix tree representations of *ILLI* and *IHTI*

```
Input: string x of length n
Output: suffix tree for x

headloc := root
tailptr := 1
q := 0                              /* base address */
p := 0                              /* length of chain */
T := insertleaf(ST_0, root, 1)
while tailptr ≤ n or headloc ≠ root do
    if headloc = root then                      /* case (1) */
        (headloc, tailptr) := scanprefix(T, headloc, tailptr + 1)
    else                                /* case (2) */
        if headloc is a node in T then
            headloc := headloc.suffixlink   /* case (2.1) */
            (headloc, tailptr) := scanprefix(T, headloc, tailptr)
        else                                /* case (2.2) */
            let (ū, av) := headloc
            if ū = root then
                headloc := rescan(T, root, v)        /* case (2.2.1) */
            else
                ū := headloc.suffixlink       /* case (2.2.2) */
                headloc := rescan(T, ū, av)
            p := p + 1                    /* chain is extended */
            if headloc is a node in T then
                q.suffixlink := base address of headloc /* case (2.2.3) */
                for d := 1 to p − 1 do
                    (q − 2d).distance := d
                p := 0
                q := q + 4
                (headloc, tailptr) := scanprefix(T, headloc, tailptr)
            else
                q := q + 2                  /* case (2.2.4) */
    if headloc is a node in T then
        T := insertleaf(T, headloc, tailptr)       /* case (a) */
    else
        T := insertbranch(T, q, headloc, tailptr)          /* case (b) */
        /* q is the base address of the branching node to be created */
```

Both corpora are widely used to compare lossless data compression programs. Since they are easily available, we propose their usage for comparing programs developed for other string processing tasks.

- We added 8 of the 11 DNA sequences used by Lefévre and Ikeda [LI93]. These are denoted by their EMBL database accession number. For the remaining three DNA sequences used in [LI93], the authors did not give enough information to locate them.
- We extracted a section of 500,000 residues from the PIR database, denoted by *PIR500*. The underlying alphabet is of size 20.
- 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.

7.1 Space Requirement

We compared the space requirement of the described implementation techniques with the space requirement of variants of directed acyclic word graphs [BBH+85, Cro86]. In order 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* (*pestry*, for short) of [LI93]. The same holds for the *compact dawg* (*cdawg*, for short) of [BBH+87]. A tight implementation of a *dawg* and a *pestry*, based on linked lists, requires 9 bytes for each node and 8 bytes for each edge. For the *cdawg*, 12 bytes are required for each node and for each edge. These numbers are consistent with [CV97]. 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 1.

Table 2 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 2 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 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, a grey box marks the smallest relative space requirement. The last row shows the total lengths of all 42 files, and the average relative space requirement. 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 2. Since DNA sequences are very important for suffix tree applications, we in particular comment on the

Table 2: 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
		18684070		30.33	19.78	14.55	18.82	10.10	14.66

corresponding behavior of the considered data structures.

1. The *dawg* is the least space efficient data structure. The space requirement for DNA sequences is $34.03n$ bytes.

2. The *pestry* requires about 35% less space than the *dawg*. For DNA sequences the space requirement is $26.62n$ bytes. Thus the saving over the *dawg* is only 22%. This is slightly smaller than the saving of $25\% - 30\%$ reported by Lefévre and Ikeda [LI93].

3. A *cdawg* requires on average 26% less space than a *pestry*. For DNA sequences the space requirement is $22.54n$ bytes which is 15% less than the *pestry*. As suggested in [CV97] 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 in [CV97]. For alphabets larger than 4, *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.

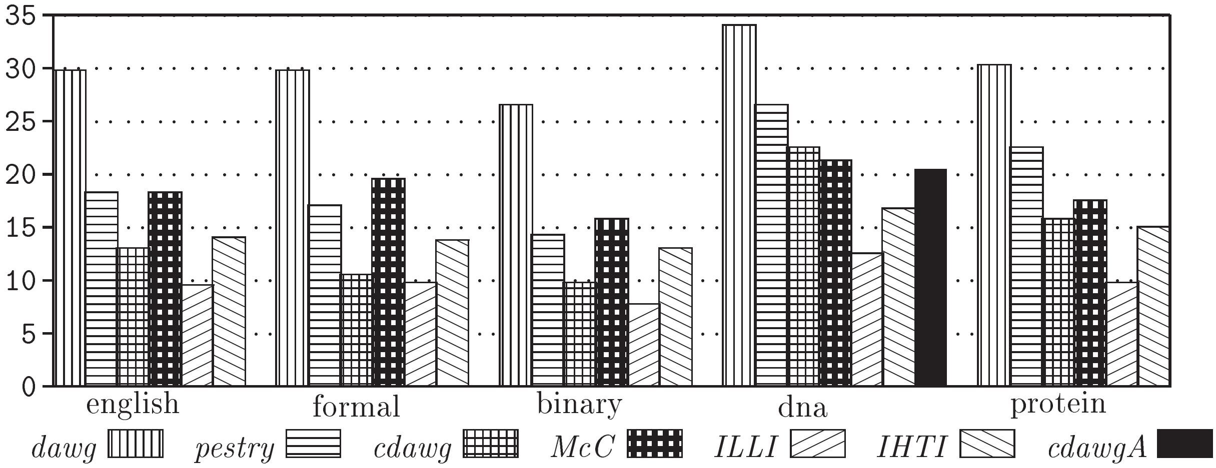
4. The suffix tree in the implementation following McCreight uses 30% more space than the *cdawg*. However, for all DNA sequences, except for J03071, it requires less space than the *cdawg*. The space requirement for DNA sequences is $21.27n$ bytes, which means a saving over the *cdawg* of 6%, and 3% 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% and over *McC* by 46%. For all classes of files there is an advantage over the *cdawg*. However, there are 6 files for which the *cdawg* requires less space. For DNA sequences the space requirement is $12.55n$ bytes, which is an improvement of 44% and 39% over the *cdawg* and the *cdawgA*, 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% 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%. For DNA sequences, the space requirement is $16.86n$ bytes, which is an improvement over the *cdawg* and *cdawgA* of 25% and 18%. The space requirement varies between $12.31n$ bytes and $17.38n$ bytes.

Figure 8 presents the data of Table 2 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 7 columns, where the last column refers to *cdawgA*. It is obvious that the size of the data structures depends on the kind of input data: Binary strings lead to

Figure 8: Relative space requirement (in bytes/input char) of *dawg*s and suffix trees for five groups of files



the smallest data structure, 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.

7.2 Running Time

For our second experiment we implemented four different variants of McCreight's algorithm named after the implementation technique they employ: *SLLI*, *SHTI*, *ILLI*, and *IHTI*, see sections 3.1, 3.2, 4.2, and 4.3. All programs are written in C.

The characters of the input string x are always represented by one byte. That is, $\Sigma \setminus \{\$\}$ is restricted to be a subset of the 256 character ASCII-alphabet. This alphabet has a predefined order, according to which the list of edges outgoing from a branching node is sorted. Since x may contain up to 256 different characters (e.g. for a binary file), we cannot represent the sentinel character \$ by one byte. Instead, we implement it as a pointer *sentinel* to a *virtual* character, that is larger than any character of the ASCII alphabet. More precisely, we store x in an input buffer from index 0 to $n-1$ and let $sentinel = n$. Our implementation takes care that the virtual character, *sentinel* points to, is never compared to any character of the ASCII alphabet. Such a comparison is not necessary, since we always know it's result. Both linked list implementations use a Σ-indexed table which allows to access each successor of the *root* in constant time. This table requires just 256 extra integers and it results in considerable speedups for large alphabets.

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. Figure 3 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 *throughput*

$tput = (10^6 \cdot time)/n$, 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 length of the files, the total running time and the average throughput. In each line a grey box marks the best throughput.

Table 3 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% for the linked list implementation, and 1.5% 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 implementation, one observes that the former is 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 (see [GK95]). 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 a decreasing throughput of the linked list implementation, independent of the alphabet size.

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.

8 Conclusion

8.1 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:

- In a lossless data compression program [BKS99] suffix trees in both improved implementation techniques are used to compute the Burrows and Wheeler Transformation [BW94] of a string in linear time and space. This basically means to sort all suffixes of a string in lexicographic order, see also Table 4, application 13.

- In a program called *REPuter* [KS98] suffix trees in the improved linked list imple-

Table 3: Running times (absolute and throughput in seconds) for different variants of McCreight's Algorithm

			SLLI		*ILLI*		*SHTI*		*IHTI*	
file	*length*	*k*	*time*	*tput*	*time*	*tput*	*time*	*tput*	*time*	*tput*
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
	18684070		83.09	2.92	84.98	3.03	59.44	2.46	62.28	2.50

mentation are used to compute maximal repeats in DNA sequences in optimal time. We used an algorithm described by [Gus97], see Table 4, 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 (12,147,818 bases) 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 Section 7.1), 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.

8.2 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 $3 \cdot 10^9$ bases. 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 3 integers, each large node occupies 5 integers, and table T_{leaf} consists of n 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 the throughput of our program with 10 seconds for one million bases, we obtain a running time of about $3 \cdot 10^4$ seconds which is less than 9 hours. Given the fact that (in December 1998) there are at least $2\frac{1}{2}$ years until sequencing of the complete human genome is finished and taking in to account the expected advances in hardware technology, it seems feasible to compute the suffix tree for the entire human genome on some computers.

8.3 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.

Table 4: Applications of suffix trees and the kind of traversal they require. The numbers are the application numbers used by Gusfield [Gus97]

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

(2) The suffix tree is traversed completely in a particular order. This task requires to have constant time access from one edge $\overline{v} \xrightarrow{aw} \overline{vaw}$ to another edge $\overline{v} \xrightarrow{cu} \overline{vcu}$ 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 [Gus97], 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 8 applications which perform partial traversals and 8 applications which perform complete traversals. Table 4 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 remarked in [Lar98], 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 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 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. More precisely, the running time of the additional sorting phase is between 27% and 140% of the running time of the corresponding suffix tree construction (average 73%). So, for complete traversals, the hash table representation is inferior, except when the alphabet is large.

8.4 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 8), 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, see [MH95].

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

I am indebted to Bernhard Balkenhol who suggested to further improve preliminary techniques to reduce the space requirement of suffix trees. I would like to thank Robert Giegerich and Jens Stoye who read previous versions of this paper and made suggestions to improve the presentation. Thanks go to Dirk Evers for helpful discussions on low-level operations in C.

References

[AB97] R. Arnold and T. Bell. A Corpus for the Evaluation of Lossless Compression Algorithms. In *Proceedings of Data Compression Conference*, pages 201–210, 1997. `http://corpus.canterbury.ac.nz`.

[AN93] A. Andersson and S. Nilsson. Improved Behavior of Tries by Adaptive Branching. *Information Processing Letters*, **46**:295–300, 1993.

[Apo85] A. Apostolico. The Myriad Virtues of Subword Trees. In *Combinatorial Algorithms on Words, Springer Verlag*, pages 85–96, 1985.

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

[BBH$^+$87] 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.

[BCW90] T.C. Bell, J.G. Cleary, and I.H. Witten. *Text Compression.* Prentice Hall, Englewood Cliffs, NJ, 1990.

[BEH89] A. Blumer, A. Ehrenfeucht, and D. Haussler. Average Size of Suffix Trees and DAWGS. *Discrete Applied Mathematics*, **24**:37–45, 1989.

[BKS99] B. Balkenhol, S. Kurtz, and Y.M. Shtarkov. Modification of the Burrows and Wheeler Data Compression Algorithm. In *accepted for publication at the IEEE Data Compression Conference, Snowbird, Utah.* IEEE Computer Society Press, 1999.

[BRCR94] P. Bieganski, J. Riedl, J.V. Carlis, and E.F. Retzel. Generalized Suffix Trees for Biological Sequence Data: Applications and Implementation. In *Proceedings of the IEEE 27th Hawaii International Conference on System Sciences*, pages 35–44. IEEE Computer Society Press, 1994.

[BW94] 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`.

[CL94] W.I. Chang and E.L. Lawler. Sublinear Approximate String Matching and Biological Applications. *Algorithmica*, **12**(4/5):327–344, 1994.

[Cle84] J.G. Cleary. Compact Hash Tables Using Bidirectional Linear Probing. *IEEE Trans. on Computers*, **33**(9):828–834, 1984.

[Cro86] M. Crochemore. Transducers and Repetitions. *Theoretical Computer Science*, **45**:63–86, 1986.

[CV97] M. Crochmore and R. Vérin. Direct Construction of Compact Acyclic Word Graphs. In *Proc. of the Annual Symposium on Combinatorial Pattern Matching* (CPM'97), LNCS 1264, pages 116–129, 1997.

[DCW93] 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.

[Far97] M. Farach. Optimal Suffix Tree Construction with Large Alphabets. In *Proceedings of the 38th Annual Symposium on the Foundations of Computer Science, FOCS 97*, New York, 1997. IEEE Comput. Soc. Press. `ftp://cs.rutgers.edu/pub/farach/Suffix.ps.Z`.

[FG98] 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*, to appear, 1998.

[GK95] 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.

[GK97] 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.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, New York, 1997.

[Irv96] 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`.

[Kär95] J. Kärkkäinen. Suffix Cactus: A Cross between Suffix Tree and Suffix Array. In *Proc. of the Annual Symposium on Combinatorial Pattern Matching* (CPM'95), LNCS 937, pages 191–204, 1995.

[KGS98] J. Knight, D. Gusfield, and J. Stoye. The Strmat Software-Package, 1998. `http://www.cs.ucdavis.edu/~gusfield/strmat.tar.gz`.

[Knu73] D.E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.

[KS98] S. Kurtz and C. Schleiermacher. *REPuter*: Fast Computation of Maximal Repeats in Complete Genomes. submitted, 1998.

[Kur95] S. Kurtz. *Fundamental Algorithms for a Declarative Pattern Matching System.* Dissertation, Technische Fakultät, Universität Bielefeld, available as Report 95-03, July 1995.

[Lar98] N.J. Larsson. The Context Trees of Block Sorting Compression. In *Proceedings of the IEEE Data Compression Conference, Snowbird, Utah, March 30 - April 1*, pages 189–198. IEEE Computer Society Press, 1998.

[LI93] 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.

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

[MH95] H.W. Mewes and K. Heumann. Genome Analysis: Pattern Search in Biological Macromolecules. In *Proc. of the Annual Symposium on Combinatorial Pattern Matching* (CPM'95), LNCS 937, pages 261–285, 1995.

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

[Ukk95] E. Ukkonen. On-line Construction of Suffix-Trees. *Algorithmica*, **14**(3), 1995.

[Wei73] P. Weiner. Linear Pattern Matching Algorithms. In *Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory*, pages 1–11, The Univsersity of Iowa, 15-17 October, 1973.