Engineering a Compressed Suffix Tree Implementation

N. VÄLIMÄKI
and
V. MÄKINEN
University of Helsinki,
W. GERLACH
Bielefeld University
and
K. DIXIT
IIT Kanpur

N. V. and V. M. were funded by the Academy of Finland under grant 119815.
The work of W. G. and K. D. was conducted while visiting University of Helsinki.

Addresses:
Niko Välimäki, P. O. Box 68 (Gustaf Hällströmin katu 2 b), 00014 Helsinki, Finland. Email: nvalimak@cs.helsinki.fi. Web: http://www.cs.helsinki.fi/u/nvalimak.
Veli Mäkinen, P. O. Box 68 (Gustaf Hällströmin katu 2 b), 00014 Helsinki, Finland. Email: vmakinen@cs.helsinki.fi. Web: http://www.cs.helsinki.fi/u/vmakinen.
Wolfgang Gerlach, Universität Bielefeld, AG Genominformatik, 33594 Bielefeld. Email: wolfgang.gerlach@cebitec.uni-bielefeld.de. Web: http://www.cebitec.uni-bielefeld.de/~wgerlach.
Kashyap Dixit, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, India. Email: kdixit@iitk.ac.in.

Suffix tree is one of the most important data structures in string algorithms and biological sequence analysis. Unfortunately, when it comes to implementing those algorithms and applying them to real genomic sequences, often the main memory size becomes the bottleneck. This is easily explained by the fact that while a DNA sequence of length n from alphabet $\Sigma = \{A, C, G, T\}$ can be stored in $n \log |\Sigma| = 2n$ bits, its suffix tree occupies $O(n \log n)$ bits. In practice, the size difference easily reaches factor 50.

We report on an implementation of the compressed suffix tree very recently proposed by Sadakane (*Theory of Computing Systems*, in press). The compressed suffix tree occupies space proportional to the text size, i.e. $O(n \log |\Sigma|)$ bits, and supports all typical suffix tree operations with at most $\log n$ factor slowdown. Our experiments show that, e.g. on a 10 MB DNA sequence, the compressed suffix tree takes 10% of the space of the normal suffix tree. At the same time, a representative algorithm is slowed down by factor 30.

Our implementation follows the original proposal in spirit, but some internal parts are tailored towards practical implementation. Our construction algorithm has time requirement $O(n \log n \log |\Sigma|)$ and uses closely the same space as the final structure while constructing it: on the 10 MB DNA sequence, the maximum space usage during construction is only 1.5 times the final product size. As by-products, we develop a method to create *Succinct Suffix Array* directly from Burrows-Wheeler transform and a space-efficient version of *suffixes-insertion* algorithm to build balanced parentheses representation of suffix tree from LCP information.

Categories and Subject Descriptors: E.1 [**Data structures**]: ; E.2 [**Data storage representations**]: ; E.4 [**Coding and information theory**]: Data compaction and compression; F.2.2 [**Analysis of algorithms and problem complexity**]: Nonnumerical algorithms and problems—*Pattern matching, Computations on discrete structures, Sorting and searching*; H.3.2 [**Information storage and retrieval**]: Information storage; H.3.3 [**Information storage and retrieval**]: Information search and retrieval—*Search process*

General Terms: Algorithms, Experimentation

Additional Key Words and Phrases: Algorithm engineering, biological sequence analysis, text indexing, text compression, Burrows-Wheeler transform, wavelet tree, lcp values, balanced parantheses, lowest common ancestor

1. INTRODUCTION

Myriad non-trivial combinatorial questions concerning strings turn out to have efficient solutions via extensive use of *suffix trees* [Apostolico 1985]. As a theoretical tool, suffix trees have a fundamental role in plethora of algorithmic results in the area of string matching and sequence analysis. This is no surprise, since suffix trees summarize the whole substring content of a *text* string in an economic way; suffix trees contain a root to leaf path for each suffix of the text such that each substring of the text can be read as a prefix of some path. Edges of the tree are labeled with text substrings, and can be represented just by pointers to the text. The tree has n leaves and at most $n - 1$ internal nodes, and hence representing pointers in the tree and pointers into the text take overall $O(n)$ computer words, n being the text length. The linear size requirement has made suffix trees attractive for many applications. After all, representing the $O(n^2)$ substrings of a text in $O(n)$ space is a remarkably powerfull tool. Even more advantageous is that suffix trees can be constructed in linear time [Weiner 1973; McCreight 1976; Ukkonen 1995].

Bioinformatics is a field where suffix trees would seem to have the strongest practical potential; unlike the natural language texts formed by words and delimiters (enabling specialized data structures like inverted files), biological sequences

are streams of symbols without any predefined word boundaries. Suffix trees treat any substring equally, regardless of it being a word or not. This perfect synergy has created a vast literature describing suffix tree -based algorithms for sequence analysis problems, see e.g. [Gusfield 1997]. Several implementations exist as well, like `STRMAT`, `WOTD`, `LIBSTREE`, and `MUMMER` [1], to name a few.

Unfortunately, the theoretically attractive properties of suffix trees do not always meet the practical realm. For example, the problem of searching approximate occurrences of a pattern in a long text could be solved using suffix tree -like data structures (see e.g. a recent development in this area [Cole et al. 2004]). In practice, the highly popular software tools like BLAST [Altschul et al. 1990] are based on quite different techniques.

The main reason why suffix trees have remained mainly as theoretical tools is their immense space consumption. Even for a reasonable size genomic sequence of 100 MB, its suffix tree may require 5 GB of main memory. This phenomenon is not just a consequence of constant factors in the implementation of the structure, but rather an asymptotic effect. When examined more carefully, one notices that a sequence of length n from an alphabet Σ requires only $n \log |\Sigma|$ *bits* of space, whereas its suffix tree requires $O(n \log n)$ bits. Hence, the space requirement is by no means linear when measured in bit-level.

The size bottleneck of suffix trees has made the research turn into looking for more space-economic variants of suffix trees. One popular alternative is the *suffix array* [Manber and Myers 1993]. It basically removes the constant factor of suffix trees to 1, as what remains from suffix trees is a lexicographically ordered array of starting positions of suffixes in the text that occupies $n \log n$ bits. Many tasks on suffix trees can be simulated by $\log n$ factor slowdown using suffix arrays. With three additional tables, suffix arrays can be enhanced to support typical suffix tree operations without any slowdown [Abouelhoda et al. 2004].

A recent twist in the development of full-text indexes is the use of *abstract data structures*; the operations supported by a data structure are identified and the best possible implementation is sought for that supports those operations. This line of development has led to *compressed suffix arrays* [Grossi and Vitter 2006; Ferragina and Manzini 2005] (see [Navarro and Mäkinen 2007] for more references). These data structures take, in essence, $n \log |\Sigma|(1 + o(1))$ bits of space, being asymptotically space-optimal. For compressible sequences they take even less space. More importantly, they simulate suffix array operations with logarithmic slowdowns, and support some operations (like pattern search) even faster than plain suffix arrays or suffix trees. These structures are also called *self-indexes* as they do not need the text to function; the text is actually represented compressed within the index.

Very recently Sadakane [Sadakane 2007] extended the abstract data structure concept to cover suffix trees, identifying typical operations suffix trees are assumed to possess. Some of these operations, like navigating in a tree, were already extensively studied by [Munro et al. 2001]. In addition to these navigational operations,

[1] http://www.cs.ucdavis.edu/~gusfield/strmat.html,
http://bibiserv.techfak.uni-bielefeld.de/wotd/,
http://www.cl.cam.ac.uk/~cpk25/libstree/,
http://sourceforge.net/projects/mummer/

suffix trees have several other useful operations such as suffix links, constant time lowest common ancestor (lca) queries, and pattern search capabilities. Sadakane developed a fully functional suffix tree structure by combining compressed suffix arrays with several other non-trivial new structures. Each operation was supported by at most $\log n$ slowdown, often the slowdown being only a constant. The space requirement was shown to be still asymptotically optimal, more accurately, $|CSA| + 6n + o(n)$ bits, where $|CSA|$ is the size of the compressed suffix array used.

This paper studies an implementation of Sadakane's compressed suffix tree. We implemented the structure following closely the original proposal [Sadakane 2007]. Since there are many sub-structures involved, there are many places to consider space-time tradeoff issues. For example, some of the sublinear $o(n)$ structures turn out to have inpractically large constants, and in such cases it is essential to consider whether some constant factor c in space usage can be turned into $O(c)$ time factor. Our aim was to develop a version that has space-time tradeoff parameters whenever possible. We managed to engineer a version with a reasonable space-efficiency (see Sect. 9 for some numbers).

A problem related to the practical implementation is how to construct the compressed suffix tree without using too much extra space at construction time. There are many tasks in compressed suffix tree construction that need special attention: (1) How to construct the Burrows-Wheeler transform on which the compressed suffix arrays are based on; (2) storing sampled text/suffix array positions; (3) direct construction of compressed longest common prefix information, and (4) construction of balanced parentheses representation of suffix tree directly from compressed suffix array. Tasks (1), (3) and (4) have been considered in [Hon and Sadakane 2002] and later improved in [Hon et al. 2003a] so as to obtain an $O(n \log^{\epsilon} n)$ time algorithm to construct compressed suffix trees, where $\epsilon > 0$. Task (2) is related to our choice of implementing compressed suffix arrays using structures evolved from FM-index [Ferragina and Manzini 2005], and is tackled in this paper. Also for task (3) our solution variates slightly from [Hon and Sadakane 2002] as we build on top of the suffixes-insertion algorithm [Crochemore and Rytter 2002] and they build on top of the post-order traversal algorithm of [Kasai et al. 2001]. The final time-requirement of our implementation is $O(n \log n \log |\Sigma|)$, being reasonably close to the best current theoretical result [Hon et al. 2003a].

The outline of the article is as follows. Section 2 gives the basic definitions and a very cursory overview of Sadakane's structure. Section 3 explains how we implemented compressed suffix arrays (related to task (1)) and provides solutions to task (2). Section 4 describes the solution hinted in [Hon and Sadakane 2002] for task (3). Section 5 gives an overview of balanced parentheses and describes our construction algorithm, solving task (4). Section 6 explains how we implemented the lowest common ancestor structure by adding a space-time tradeoff parameter. Section 7 explains the software engineering conventions used. We conclude with some illustrative experimental results in Sect. 8. Some final remarks are given in Sect. 9.

The software package can be downloaded from
`http://www.cs.helsinki.fi/group/suds/cst/`.

2. PRELIMINARIES

A *string* $T = t_1 t_2 \cdots t_n$ is a sequence of *characters* from an ordered *alphabet* Σ. A *substring* of T is any string $T_{i...j} = t_i t_{i+1} \cdots t_j$, where $1 \leq i \leq j \leq n$. A *suffix* of T is any substring $T_{i...n}$, where $1 \leq i \leq n$. A *prefix* of T is any substring $T_{1...j}$, where $1 \leq j \leq n$. A *pattern* is a short string over the alphabet Σ. We say that pattern $P = p_1 p_2 \cdots p_k$ *occurs* at position j of *text* string T iff $p_1 = t_j, p_2 = t_{j+1}, \ldots, p_k = t_{j+k-1}$.

Definition 2.1. (Adopted from [Gusfield 1997]) The *keyword trie* for the set $\mathcal{P}$ of strings is a rooted directed tree $\mathcal{K}$ satisfying three conditions: (1) Each edge is labeled with exactly one character; (2) any two edges out of the same node have distinct labels; (3) every pattern P of $\mathcal{P}$ maps to some node v of $\mathcal{K}$ such that the characters on the path from the root of $\mathcal{K}$ to v spell out P, and every leaf of $\mathcal{K}$ is mapped to by some string in $\mathcal{P}$.

Definition 2.2. The *suffix trie* of text T is a keyword trie for set $\mathcal{S}$, where $\mathcal{S}$ is the set of all suffixes of T.

Definition 2.3. The *suffix tree* of text T is the *path-compressed* suffix trie of T, i.e., a tree that is obtained by representing each maximal non-branching path of the suffix trie as a single edge labeled by the catenation of the labels in the corresponding edges of the suffix trie. The *edge labels* of suffix tree correspond to substrings of T; each edge can be represented as a pair (l, r), such that $T_{l...r}$ gives the label.

A *path label* of a node v is the catenation of edge labels from root to v. Its length is called *string depth*. The number of edges from root to v is called *node depth*. The *suffix link* $sl(v)$ of an internal node v with path label $x\alpha$, where x denotes a single character and α denotes a possibly empty substring, is the node with path label α.

A typical operation on suffix trees is the *lowest common ancestor* query, which can be used to compute the *longest common extension* $lce(i, j)$ of two arbitrary suffixes $T_{i...n}$ and $T_{j...n}$: Let v and w be the two leaves of a suffix tree having path labels $T_{i...n}$ and $T_{j...n}$, respectively. Then the path label α of the lowest common ancestor node of v and w is the longest prefix shared by the two suffixes. We have $lce(i, j) = |\alpha|$.

The following abstract definition captures the above mentioned typical suffix tree operations.

Definition 2.4. An *abstract suffix tree* for a text supports the following operations:

(1) $root()$: returns the root node.
(2) $isleaf(v)$: returns Yes if v is a leaf, and No otherwise.
(3) $child(v, c)$: returns the node w that is a child of v and the edge (v, w) begins with character c, or returns 0 if no such child exists.
(4) $sibling(v)$: returns the next sibling of node v.
(5) $parent(v)$: returns the parent node of v.
(6) $edge(v, d)$: returns the d-th character of the edge-label of an edge pointing to v.

(7) $depth(v)$: returns the string depth of node v.
(8) $lca(v, w)$: returns the lowest common ancestor between nodes v and w.
(9) $sl(v)$: returns the node w that is pointed to by the suffix link from v.

The rest of the paper studies an approach to support the abstract suffix tree operations efficiently, while using less space than the pointer-based classical suffix tree implementations.

2.1 Overview of Compressed Suffix Tree

[Sadakane 2007] shows how to implement each operation listed in Def. 2.4 by means of a sequence of operations on (1) compressed suffix array, (2) *lcp*-array [2], (3) balanced parentheses representation of suffix tree hierarchy, and (4) a structure for *lca*-queries. In the following sections we explain how we implemented those structures.

3. COMPRESSED SUFFIX ARRAY

Suffix array is a simplified version of suffix tree; it only lists the suffixes of the text in lexicogaphic order. Let $SA[1 \dots n]$ be a table such that $T_{SA[i]\dots n}$ gives the i-th smallest suffix in lexicographic order. Notice that this table can be filled by a depth-first traversal on a suffix tree following its edges in the lexicogaphic order.

As the array SA takes $n \log n$ bits, there has been considerable effort in building *compressed suffix arrays* to reduce its space requirement, see [Navarro and Mäkinen 2007]. The following captures typical suffix array operations on an abstract level.

Definition 3.1. An *abstract suffix array* for a text T supports the following operations:

—$lookup(i)$: returns $SA[i]$,
—$inverse(i)$: returns $j = SA^{-1}[i]$, defined such that $SA[j] = i$,
—$\Psi(i)$: returns $SA^{-1}[SA[i] + 1]$, and
—$substring(i, l)$: returns $T[SA[i] \dots SA[i] + l - 1]$.

The function $\Psi[i]$ is defined as follows:

Definition 3.2.

$$\Psi(i) = \begin{cases} i' \text{ such that } SA[i'] = SA[i] + 1 & (\text{if } SA[i] < n) \\ 1 & \text{if } SA[i] = n \end{cases}$$

3.1 Our Implementation

We used *Succinct Suffix Array* (SSA) of [Mäkinen and Navarro 2005] to implement the abstract suffix array operations. The base structure is the *wavelet tree* [Grossi et al. 2003] build on the *Burrows-Wheeler transform* [Burrows and Wheeler 1994]. [3] Let us briefly revise the structure, as we extend it to support functions Ψ and $inverse$ that are not considered in the original proposal.

[2] [Sadakane 2007] uses name *Height*-array.
[3] For background on these techniques, see a recent survey [Navarro and Mäkinen 2007].

The Burrows-Wheeler transform T^{bwt} is defined as $T^{bwt}[i] = T_{SA[i]-1}$ (where $SA[i]-1 = SA[n]$ when $SA[i] = 1$). A property of T^{bwt} used in compressed suffix arrays is so-called LF-mapping:

Definition 3.3.

$$LF(i) = \begin{cases} i' \text{ such that } SA[i'] = SA[i] - 1 & (\text{if } SA[i] > 1) \\ n & \text{if } SA[i] = 1 \end{cases}$$

It can be shown [Ferragina and Manzini 2005] that LF-mapping can computed by the means of T^{bwt}:

Lemma 3.4 [Ferragina and Manzini 2005]. *Let $c = T^{bwt}[i]$. Then*

$$LF(i) = C[c] + rank_c(T^{bwt}, i), \tag{1}$$

where $C[c]$ is the the number of positions of T^{bwt} containing a character smaller than c and $rank_c(T^{bwt}, i)$ tells how many times character c occurs upto position i in T^{bwt}.

Table $C[1 \ldots |\Sigma|]$ can be stored as is in $|\Sigma| \log n$ bits of space, and space-efficient data structures built for storing $rank_c$-function values. For example, a simplified version of the wavelet tree (see [Mäkinen and Navarro 2005, Sect. 5]) stores those values in $n \log |\Sigma|(1 + o(1))$ bits so that each $rank_c$ value (as well as value $T^{bwt}[i]$) can be computed in $O(\log |\Sigma|)$ time.

Let us now consider how the abstract suffix array operations can be simulated using LF-mapping. First, notice that LF-mapping lets us browse the text backwards starting from any given position. We store for every R-th text position $i' \cdot R$ its location in suffix array explicitly: $\texttt{sampledSAinverse}[i'] = j$ such that $SA[j] = i' \cdot R$. Now, the $substring(i, l)$-query can be supported as follows. We compute the smallest integer i' such that $i + l \leq i' \cdot R$. Then substring $T_{i \ldots i' \cdot R - 1}$ is retrieved in reverse order by applying LF-mapping repeatedly: $t_{i' \cdot R - 1} = T^{bwt}[j]$, $t_{i' \cdot R - 2} = T^{bwt}[LF[j]]$, $t_{i' \cdot R - 3} = T^{bwt}[LF[LF[j]]]$, ... Retrieving a single character takes $O(\log |\Sigma|)$ time, hence the total time complexity for $substring(i, l)$ is $O((l+R) \log |\Sigma|)$. Answering $inverse(i)$ is analogous: LF-mapping is applied $i' \cdot R - i$ times starting from $\texttt{sampledSAinverse}[i']$. The index j reached in the end has the desired property $S[j] = i$. The time needed is $O(R \log |\Sigma|)$.

For answering $lookup(i)$ and $\Psi(i)$ we need more structures. We store values $B[j] = 1$ such that $SA[i]$ is divisible by R. That is, we mark the suffix array indices containing sampled text positions. We store these sampled positions in the suffix array order into another table $\texttt{sampledSA}$ such that $\texttt{sampledSA}[rank_1(B, j)] = SA[j]$ whenever $B[j] = 1$. Function $lookup(i)$ can now be answered by applying $j = LF[j]$ starting with $j = i$ until $B[j] = 1$. Then $lookup(i) = \texttt{sampledSA}[rank_1(B, j)] + k$, where k is the number of times LF-mapping was applied. The time needed is still $O(R \log |\Sigma|)$, as the binary $rank_1(B, j)$-query can be answered in constant time after building $o(n)$ bits data structures on top of B [Jacobson 1989].

Finally, to answer $\Psi(i)$, we first apply $j = lookup(i)$, then apply LF-mapping starting from $\texttt{sampledSAinverse}[j/R + 1]$ until reaching again index i. Let i' be the index reached just before applying $LF[i'] = i$. By definition $\Psi(i) = i'$. This computation also takes $O(R \log |\Sigma|)$ time.

In our implementation, we use the Huffman-tree shape as advised in [Mäkinen and Navarro 2005], so that the structure takes overall $\frac{2n}{R}\log n + n(H_0+2)(1+o(1))$ bits of space and supports all the abstract suffix array operations in $O(R \cdot H_0)$ average time. (Worst case $O(R \cdot \log n)$. Use $R+l$ instead of R for $substring(i,l)$ function time requirement.) Here H_k is the *k-th order entropy* of T [Manzini 2001]. Recall that $H_k \leq H_{k-1} \leq \cdots \leq H_0 \leq \log|\Sigma|$. Fixing any $R = \Omega(\frac{\log n}{\log|\Sigma|})$, the structure takes $O(n\log|\Sigma|)$ bits.

3.1.1 *Space-efficient Construction via Dynamic Structure.* The construction of the structure is done in two phases. First the Burrows-Wheeler transform is constructed, then the additional structures. (wavelet tree, tables C, `sampledSA`, `sampledSAinverse`, and B and its `rank` structures) are created.

The first phase can be executed in $O(n\log n\log|\Sigma|)$ time and using $nH_k + o(n\log|\Sigma|)$ bits of space by using the dynamic self-index explained in [Mäkinen and Navarro 2006; 2007a]. We implemented the simplified version that uses $O(n\log|\Sigma|)$ bits: Instead of using the more complicated solution to solve *rank*-queries on dynamic bitvectors, we used the $O(n)$ bits structure of [Chan et al. 2004] (see also [Mäkinen and Navarro 2006, Sect. 3.2]). Using this inside dynamic wavelet trees one obtains the claimed result (see the paragraph just before Sect. 6 in [Mäkinen and Navarro 2006]). The result is actually a dynamic wavelet tree of the Burrows-Wheeler transform supporting $rank_c$-queries in $O(\log n\log|\Sigma|)$ time. This is easily converted into a static structure of the original SSA (in time linear in the size of the structure) that supports $rank_c$-queries in $O(\log|\Sigma|)$ time. In our implementation, we use the Huffman-shaped wavelet tree to improve the space to $O(nH_0)$ bits. This conversion is also easily done by extracting the Burrows-Wheeler transform from the dynamic wavelet tree with a depth-first traversal and creating the Huffman-balanced static wavelet tree instead as in [Mäkinen and Navarro 2005].

We are left with explaining how to construct the rest of the structures. Table C is trivial to construct in $O(|\Sigma|+n)$ time. Tables `sampledSA`, `sampledSAinverse` and bitvector B can be constructed as follows. We apply LF-mapping from the index of the last text position on (which is now possible as table C and wavelet tree to support $rank_c$-queries of Lemma 3.4 are ready). That is, we virtually scan the text backwards by using LF-mapping. Whenever we are at a text position divisible by R, say at position $i \cdot R$, we also know the suffix array index, say j. That is, we can directly mark $B[j]=1$ and store `sampledSAinverse`$[i]=j$. After virtually scanning the text backwards we have filled B and `sampledSAinverse` correctly. To fill in table `sampledSA`, we first preprocess B for $rank_1(B,i)$ queries, and then virtually scan the text backwards again. Analogously as before, whenever we are at a text position divisible by R, say at position $i \cdot R$, we also know the suffix array index, say j. At those positions, we store `sampledSA`$[rank_1(B,j)]=i$. The space used for the construction is the same as what the resulting structures take. The time needed is $O(n\log|\Sigma|)$ as each LF-step takes $O(\log|\Sigma|)$ time and we have $2n$ steps.

The bottleneck in the construction time is the creation of the Burrows-Wheeler transform within $O(n\log|\Sigma|)$ bits of space. Our implementation uses $O(n\log n\log|\Sigma|)$ time for the task. This can be sped up in theory using e.g. the $O(n\log\log|\Sigma|)$ time algorithm of [Hon et al. 2003b] that guarantees the same

asymptotic space.

4. LCP-ARRAY

Array $lcp[1 \ldots n-1]$ is used to store the longest common prefix information between consecutive suffixes in the lexicographic order. That is, $lcp[i] = |prefix(T_{SA[i]\ldots n}, T_{SA[i+1]\ldots n})|$, where $prefix(X,Y) = x_1 \cdots x_j$ such that $x_1 = y_1$, $x_2 = y_2$, ..., $x_j = y_j$, but $x_{j+1} \neq y_{j+1}$. [Sadakane 2007] describes a clever encoding of the lcp-array that uses $2n + o(n)$ bits. The encoding is based on the fact that values $i + lcp[i]$ are increasing when listed in the text position order; sequence $S = s_1, \ldots, s_{n-1} = 1 + lcp[SA^{-1}[1]], 2 + lcp[SA^{-1}[2]], \ldots, n-1+lcp[SA^{n-1}[n-1]]$ is increasing (see next subsection to see why).

To encode the increasing list S, it is enough to encode each $\texttt{diff}(i) = s_i - s_{i-1}$ in unary: $0^{\texttt{diff}(i)}1$, where we assume $s_0 = 0$ and 0^d denotes repetition of 0-bit d-times. This encoding, call it H, takes at most $2n$ bits. We have the connection $\texttt{diff}(k) = select_1(H,k) - select_1(H,k-1) - 1$, where $select_1(H,k)$ gives the position of the k-th 1-bit in H. Bitvector H can be preprocessed to answer $select_1(H,k)$-queries in constant time using $o(|H|)$ bits of extra space [Munro 1996].

Computing $lcp[i]$ can now be done as follows. Compute $k = SA[i]$ using $lookup(i)$. The value $lcp[i]$ equals $select_1(H,k) - k$.

4.1 Space-efficient Construction via Kasai et al. Algorithm

[Kasai et al. 2001] gave a linear time algorithm to construct the lcp-array given SA. One could use it to construct the encoding H by applying what is described above, but the intermediate lcp-array would take $n \log n$ bits. Instead, one can easily modify Kasai et al. algorithm to directly give the encoding H [Hon and Sadakane 2002].

Kasai et al. algorithm is based on the observation that lcp-array values for consecutive suffixes in the *text order* cannot decrease much. More concretely, it holds $lcp[SA^{-1}[i+1]] \geq lcp[SA^{-1}[i]] - 1$ [Kasai et al. 2001]. This has the consequence that one can can compute the lcp-values in the text order, at each step taking advantage of the already computed prefix length in the previous step: Let $\ell = \max(0, lcp[SA^{-1}[i]] - 1)$. Then $lcp[SA^{-1}[i+1]] = \ell + |prefix(T_{i+1+\ell\ldots n}, T_{SA[SA^{-1}[i+1]+1]+\ell\ldots n})|$. Function $prefix()$ can be computed trivially by scanning the text; this will take amortized constant time per step, as the comparison position in the first argument will advance at each step. Now, to produce H directly, we notice that the evaluation order is the same as the order in which lcp-values are stored in H. A step of the algorithm becomes simply: Let $\ell = \max(0, lcp - 1)$. Then $lcp = \ell + |prefix(T_{i+1+\ell\ldots n}, T_{SA[SA^{-1}[i+1]+1]+\ell\ldots n})|$ and append H with $0^{lcp}1$. Here $lcp = 0$ initially and accesses $SA[i]$ and $SA^{-1}[j]$ can done by operations $lookup(i)$ and $inverse(j)$ on the compressed suffix array. After producing H, one can preprocess it for constant time $select_1$ queries in linear time.

The construction uses no extra memory in addition to the text, the compressed suffix array, and the outcome of size $2n + o(n)$ bits. Using the compressed suffix array explained earlier in this paper, the time requirement is $O(n \log n)$.

5. BALANCED PARENTHESES

The *balanced parenthesis* representation P of a tree is produced by a preorder traversal printing $'('$ whenever a node is visited the first time, and printing $')'$ whenever a node is visited the last time [Munro et al. 2001]. Letting $'('= 1$ and $')' = 0$, the sequence P takes $2u$ bits on a tree of u nodes. A suffix tree of n leaves can have at most $n-1$ internal nodes, and hence its balanced parenthesis representation takes at most $4n$ bits.

[Munro et al. 2001] explain how to simulate tree traversal by means of P. After building several structures of sublinear size, one can go e.g. from a node to its first child, from a node to its next sibling, and from a node to its parent, each in constant time. [Sadakane 2007] lists many other operations that are required in his compressed suffix tree. All these navigational operations can be expressed as combinations of the following functions: $rank_p$, $select_p$, $findclose$, and $enclose$. Here p is a constant size bitvector pattern, e.g. 10 expresses an open-close parenthesis pair. Function $rank_p(P, i)$ returns the number of occurrences of p in P upto position i. Function $select_p(P, j)$ returns the position of the j-th occurrences of p in P. Function $findclose(P, i)$ returns the position of the matching closing parenthesis for the open parenthesis at position i. Function $enclose(P, i)$ returns the open parenthesis position of the parent of the node whose open parenthesis is at position i.

To get an idea of the power of the above navigational operations, let us consider how to compute the subtree size for a given node v. Let v be the j-th node in the preorder of the tree. Then $i = select_1(P, j)$ gives its location in P. Its subtree is encoded in the subrange $P[i+1 \ldots k-1]$, where $k = findclose(P, i)$. As each node in the subtree of v is encoded by two bits, the number of nodes in the subtree of v is simply $(i-k-1)/2$. Also the number of leaves in the subtree of v is easily calculated: As pattern $p = 10$ represents an open-close paranthesis pair, i.e. a leaf node, the number of leaves in the subtree of v is $rank_p(P, k) - rank(P, i)$.

5.1 Our Implementation

We used the existing *rank* and *select* implementations that are explained and experimented in [González et al. 2005]. There *rank* is the constant time solution of [Clark 1996], but *select* is implemented by binary search on *rank* values. (the constant time solution [Clark 1996] is inferior to this on practical inputs [González et al. 2005]). Section 7 explains how these solutions are modified to the case of short patterns p, as the original implementations assume $p = 1$. For $findclose$ and *enclose* we used Navarro's implementations explained in [Navarro 2004] that are based on [Munro et al. 2001]; these are faster in practice than the original, but the worst case is raised from a constant to $O(\log \log n)$.

5.2 Space-efficient Construction via LCP Information

To build balanced parentheses sequence of suffix tree space-efficiently one cannot proceed naively; doing preorder traversal on a pointer-based suffix tree requires $O(n \log n)$ bits of extra memory. We consider a new approach that builds the parentheses sequence incrementally. A very similar algorithm is already given in [Hon and Sadakane 2002]; we will consider the differences in the end of the section.

Recall from [Crochemore and Rytter 2002, Theorem 7.5, p. 97] the *suffixes-*

insertion algorithm to construct a suffix tree from LCP information: The algorithm adds suffixes in lexicographic order into a tree, having the keyword tree of suffixes $T_{SA[1]...n}, T_{SA[2]...n}, \ldots, T_{SA[i]...n}$ ready after i-th step. Suffix $T_{SA[i+1]...n}$ is then added after finding bottom-up from the rightmost path of the tree the correct insertion point. That is, the *split node* v closest to the rightmost leaf (corresponding to suffix $T_{SA[i]...n}$) whose string depth is smaller or equal to $lcp[i]$ is sought for. If the depth is equal, then a new leaf (corresponding to suffix $T_{SA[i+1]...n}$) is created as its child. Otherwise, its outgoing rightmost edge is split, a new internal node is inserted in between, and the leaf corresponding to suffix $T_{SA[i+1]...n}$ is added as its rightmost child. It is easy to see by an amortizement argument that this algorithm takes linear time.

The problem of the suffixes-insertion algorithm for our purposes is that the tree structure takes $O(n \log n)$ bits. For this reason, we develop a new version of this algorithm that represents the necessary parts of this dynamically changing tree structure by space-efficient data structures.

To obtain a space-efficient version of the algorithm, we maintain the balanced parentheses representation of the tree at each step. Unfortunately, the parentheses structure does not change sequentially, so we need to maintain it using a dynamic bitvector allowing insertions of bits (open/close parentheses) inside it. Such bitvector can be maintained using $O(n)$ bits of space so that accessing the bits and inserting/deleting takes $O(\log n)$ time [Chan et al. 2004]. In addition to the balanced parentheses to store the tree hierarchy, we need more operations on the rightmost path; we need to be able to virtually browse the rightmost path from leaf to root as well as to compute the the string depth of each node visited.

Let us first study string depths. Consider sequence $E(i) = e_1, e_2, \ldots, e_k$ of egde label lengths from leaf to root in the righmost path after i-th step of the algorithm. Naturally $\sum_{j=1}^{k} e_j = n$, as the string depth of the leaf is n. To find the split node v of the $(i+1)$-th step, we just need to compute the smallest j such that $sdepth(j) = n - \sum_{j'=1}^{j} e_{j'} \leq lcp[i]$, as this tells us to skip j edges before (virtually) reaching the split node v. To update the sequence $e_1, e_2, \ldots, e_k$ to correspond the new rightmost path, it is enough to delete values $e_1, \ldots, e_j$ from $E(i)$, insert value $e_{split} = lcp[i] - sdepth(j)$ as the first element in $E(i)$, and then finally insert $e_{leaf} = n - lcp[i]$ as the first element in $E(i)$. These two values correspond to the lengths of the edge labels of the two new edges on the path; if $e_{split} = 0$, i.e. the new leaf is inserted directly as the child of v, then only value e_{leaf} is inserted. After these modifications, we have created the sequence $E(i+1)$ of edge label lengths from leaf to root in the righmost path after $(i+1)$-th step of the algorithm. We will later consider how to maintain sequence $E(i)$ during the algorithm in a succinct form such that the modifications to the beginning are possible.

In addition to the string depths, we need to maintain information to find the insertion position in the balanced parantheses representation P of the tree. This is analogous to the maintainance of string depths. Consider again the i-th step of the algorithm. Each node in the rightmost path of the current tree is represented by an open paranthesis in P. Moreover, these parantheses occur in the same order as the nodes in the path. Hence, we can list the distances between these nodes with a sequence similar to $E(i)$. Let this sequence be $D(i) = d_1, d_2, \ldots d_k$, where

$d_{k'}$ gives the distance between the open parantheses of k'-th and $(k'+1)$-th node computed bottom-up from leaf to root in the rightmost path. Since P is at most of length $2n$, we have that $\sum_{j=1}^{k} d_j \leq 2n$ at each step. To modify P from step i to step $i+1$, we do the following. First, we assume that P is not completed with respect to the rightmost path, i.e., it does not contain the k closing parantheses in the end to close the nodes on the rightmost path (except the leaf). These closing parantheses will be added once the corresponding subtrees become ready (when no more updates are possible). Let $p = |P| - 1$ after step i (position of the last open paranthesis), and again j the smallest value such that $sdepth(j) \leq lcp[i]$. Hence, we may append P by $j-1$ close parantheses, as this is the amount of nodes on the rightmost path whose subtrees become ready. The open paranthesis of the split node v is at position $pos(j) = p - \sum_{j'=1}^{j} d_{j'}$ in P. If a new internal node is to be inserted (in case $sdepth(j) < lcp[i]$), we insert a new open paranthesis just before $r = pos(j-1)$; to see why, notice that $P[pos(j)+1 \ldots r-1]$ contains the balanced paranthesis representation of the subtree of v except the subtree of its rightmost child starting at $P[r]$. In case $sdepth(j) = lcp[i]$ P stays internally unchanged, as the new leaf will be added directly under v. In both cases we append P with a new leaf node (appending a open-close paranthesis pair). Finally, we must update sequence $D(i)$ to correspond to the current state of P. In case $sdepth(j) < lcp[i]$ we notice that d_j can be reused as the distance between v and its new rightmost child node (a new internal node). Hence, it is enough to delete values $d_1, \ldots, d_{j-1}$ from $D(i)$ and insert in the beginning value $d_{leaf} = p + j + 2 - r$ (distance between the new internal node and the new leaf). In case $sdepth(j) = lcp[i]$, we delete values $d_1, \ldots, d_j$ from $D(i)$ and insert in the beginning $d_{leaf} = p + j + 1 - pos(j)$. After these modifications, we have updated P to correspond to step $i+1$ as well as created sequences $E(i+1)$ and $D(i+1)$. By induction, after adding the last suffix (and after closing the rightmost path by adding closing parantheses as many as there are elements in $E(n)$) we have P corresponding to the suffix tree of the text. The pseudocode of the algorithm is given in Fig. 1.

5.2.1 *Handling sequence of variable length integers.* We still need to consider how to manipulate sequences E and D space-efficiently (notice that a trivial linked list approach would take $O(n \log n)$ bits space, being no improvement to the original algorithm). We encode the values using variable length prefix codes. Let us fix Elias δ encoding [Elias 1975]. It has the property that for any integer x, it holds $|\delta(x)| = \log x + o(\log x)$ bits. More importantly, a sequence $\delta(x_1)\delta(x_2)\cdots\delta(x_k)$ can be uniquely decoded into $x_1, x_2, \ldots, x_2$. This can be done in constant time per code, assuming a precomputed table of size $o(N)$, where $N = \sum_{i=1}^{k} x_i$ (See e.g. [Mäkinen and Navarro 2007b]). Notice also that $\sum_{i=1}^{k} |\delta(x_i)| \leq k \log \frac{n}{k}(1 + o(1)) = O(n)$ by the convexity of logarithm. Hence, we can store E and D using $O(n)$ bits.

The only remaining problem is how to support insertions and deletions from the beginning of δ-encoded sequences. This can be done e.g. as follows: Reserve cn bits of space, where c is a constant in the $O(n)$ space limit for the encoded sequence. Store the encoded sequence aligned to the end of the memory area, and remember the starting position. A deletion from the beginning is done by decoding the first code in constant time and shifting the starting position to the right accordingly. Identically an insertion is done by shifting the starting position to the left to make

```
Algorithm BalancedParanthesesViaLCP(lcp, n):
    P.Append(((); { Add root and first leaf }
    p = 2; {Position of the last open paranthesis }
    D.Push(1); E.Push(n); {Initialize stacks storing node/string depth information on rightmost path}
    for i = 1 to n − 1 do {Add the suffixes in the lexicographic order}
        lcp = lcp[i]; {lcp value can also be computed from its compressed representation}
        Find smallest j such that sdepth = n − E.Sum(j) ≤ lcp; {E.Sum(j) = ∑_{j'=1}^{j} e_{j'} }
        Append P with j − 1 closing parentheses;
        if sdepth < lcp then {Add new internal node and a leaf}
            r = p − D.Sum(j − 1); {Position in P}
            P.Insert((, r);
            do j times E.Pop();
            E.Push(lcp − sdepth); E.Push(n − lcp);
            do j − 1 times D.Pop();
            D.Push(p + j + 2 − r);
        else {Add a new leaf}
            r = p − D.Sum(j);
            do j times E.Pop();
            E.Push(n − lcp);
            do j times D.Pop();
            D.Push(p + j + 1 − r);
        end if
        P.Append(());
        p = |P| − 1;
    end for
    Append P with |E| closing parentheses;
```

Fig. 1. Construction of balanced parantheses representation of suffix tree by a space-efficient version of *suffixes-insertion* algorithm.

room for the new code.

We can conlude that given the *lcp*-array, we can construct the balanced parantheses sequence in $O(n \log n)$ time using $O(n)$ bits of working space.

5.2.2 *Improving running time to linear.* Finally, the time requirement can be improved to linear by replacing the dynamic bit vector by a patching technique [Kärkkäinen 2006]: The idea is to postpone the updates until a buffer of length n bits is full. Then sort the $n/\log n$ insertions positions stored in the buffer using Radix sort in $O(n/\log n)$ time, and merge the insertion positions with the already constructed P in $O(n/\log n)$ time under RAM. The buffer can become full only $O(\log n)$ times, and hence the total time used for operations on P is linear.

5.2.3 *Comparison to Hon and Sadakane solution.* [Hon and Sadakane 2002] describe a very similar algorithm. They build on top of an algorithm in [Kasai et al. 2001] that simulates the post-order traversal of suffix tree given the *lcp*-values (Kasai et al. describe the algorithm for ordinary trees, but it can easily be specialized to suffix trees). The string depths (values E) are handled identically to our algorithm. The difference is in handling node depths (values D). We use values D to track the insertion position in P. Hon and Sadakane represent P as a forest of trees such that each root corresponds to a node in the rightmost path. These nodes

partition current P into pieces that do not change during the latter steps of the algorithm. The tree of a piece is such that when preorder traversed one obtains the piece by concatenating the bits stored at each node. Space-efficiency is obtained by creating children only when $O(\log n)$ bits are stored at a node. This means that there are overall $O(n/\log n)$ pointers, needing overall $O(n)$ bits. Handling the buffers of $O(\log n)$ bits is easy, since insertions of `(` to the beginning or of `)` to the end can be done in constant time under the RAM model. The insertion operations take place during the algorithm when new internal nodes are visited in postorder.

There is, however, a problem with texts of type $\mathtt{a}^n\mathtt{\#}$, where `#<a`: Postorder traversal will visit all the leaf nodes first, creating n trees each containing two bits corresponding to `()`. Keeping pointers to those trees takes $O(n \log n)$ bits. These pointers are necessary in order to find out which trees are merged when a new internal node is visited. In fact, these pointers also need to be inserted to a stack, since they will be merged in their reverse creation order. To solve this problem, one can proceed as follows. Merge the small trees (buffers) so that each remaining tree (buffer) has size $\Theta(\log n)$ bits. Use δ-encoding to store the distances of merge-boundaries. This guarantees that there are only $O(n/\log n)$ trees, and the pointers to those trees (and inside them) take overall $O(n)$ bits. Similarly as before, the δ-encoded values occupy $O(n)$ bits.

In fact, these latter δ-values are analogous to the node-depth values D we are using. The difference between the approaches remains the handling of P.

[Hon 2004, page 59] offers a more elegant solution to the problem; instead of trying to form P on the fly, one constructs only a version of P that contains leaves `()` and closing parantheses `)`. That is, remove line $P.Insert(\mathtt{(}, r)$ from the algorithm of Fig. 1. Then run the algorithm reversed reading *lcp*-values from right to left. This creates a version of P that only contains leaves `()` and open parantheses `(`. These two sequences are easy to merge to form P as the leaves `()` occur in the same order, and between two leaves all the closing parantheses appear before the open parantheses. For example, let $P' = \mathtt{())())()))}$ and $P'' = \mathtt{(((()(()()}$ be the two sequences constructed after forward and backward scanning of *lcp*-values. Then after matching the leaves `()`, the placement of open and close parantheses are uniquely defined, that is, $P = \mathtt{(((())(()())))}$. Setting the parantheses in the other order between the second and third leaf would yield another leaf, which is not allowed.

5.2.4 *Our implementation.* In our implementation, we do not use any of the above three different ways to achieve linear time. Our implementation follows the pseudocode given in Fig. 1. Moreover, since we use the compressed *lcp*-values, the time requirement of the balanced parantheses construction remains $O(n \log n)$ even after applying one of the speed-ups. On the other hand, our current implementation follows the theoretically most space-efficient of the options: the dynamic bit-vector implemententation could be improved to $n + o(n)$ bits from the current $O(n)$ to achieve optimal space in handling P.

A practical bottleneck found when running experiments on the first versions of the construction above was the space reserved for Elias codes. The estimated worst case space is $O(n)$ bits but this rarely happens on practical inputs. We chose to reserve initially $o(n)$ bits and double the space if necessary. The parameters were

chosen so that the doubling does not affect the overall $O(n \log n)$ worst case time requirement. This reduced the maximum space usage during the construction on common inputs significantly.

6. LOWEST COMMON ANCESTOR STRUCTURE

[Farach-Colton and Bender 2000] describe a $O(n \log n)$ bits structure that can be preprocessed for a tree in $O(n)$ time to support constant time lowest common ancestor (lca) queries. [Sadakane 2007] modified this structure to take $O(n)$ bits of space without affecting the time requirements. We implemented Sadakane's proposal that builds on top of the balanced parentheses representation of previous section, adding lookup tables taking $o(n)$ bits.

6.0.5 *Implementation remark..* While implementing Sadakane's proposal, we faced a practical problem; one of the sublinear structures for lca-queries takes space $n(\log \log n)^2 / \log n$ bits, which on practical inputs is considerable amount: This lookup table was taking half the size of the complete compressed suffix tree on some inputs. To go around this bottleneck, we added a space-time tradeoff parameter K such that using space $n(\log \log n)^2/(K \log n)$ bits for this structure, one can answer lca-queries in time $O(K)$.

7. IMPLEMENTATION DESIGN

We used object oriented programming (`C++`-language) to create an easily usable and maintainable software package. Each abstract data structure explained above is its own class, making it easy to change the underlying implementations at any phase. For example, one can easily switch to another compressed suffix array implementation just by writing a new class with the same name and same operations supported.

We used to some extend generic programming in order to avoid writing similar code segments. An example of its use is our $rank_p(P, i)$ / $select_p(P, j)$ function implementations (see Sect. 5 for definitions). These operations are needed in Sadakane's compressed suffix tree for many different short patterns p like $0, 1, 10, 01$. It is known how to build $o(n)$ bits structures for each *fixed* p so that $rank_p$ and $select_p$ queries can be answered in constant time. Instead of copy-pasting those codes and changing some details depending on the pattern p, we used one generic implementation that only assumes that short substrings of a *virtual indicator vector* of P can be accessed in constant time. A virtual indicator vector of P with respect to a pattern p is $I(P, p) = I[1 \ldots |P|]$ such that $I[i] = 1$ iff pattern p occurs at position i in P, otherwise $I[i] = 0$. Now, after building a table storing for each $(\log |P|)/2$ length substring α of P the mapping to its indicator vector $I(\alpha, p) = I[1 \ldots |\alpha|]$, one can access any $O(\log |P|)$-length substring of $I(P, p)$ in constant time. This access is enough to guarantee constant time $rank_p$ and $select_p$ operations: Complete $I(P, p)$ is only needed in construction time to build the lookup tables. Later on the $rank_p$ and $select_p$ functions consult the lookup tables and need only access to short fragments of $I(P, p)$. These accesses are independent of p. Only the pointer to the lookup table to map substrings α of P to the indicator vector $I(\alpha, p)$ depends on p. Notice that the alternative approach of keeping the indicator vectors $I(P, p)$ stored in memory for each of the k values of p would require $k|P|$ bits of memory. Now

we are only using $k2^{(\log|P|)/2}(\log|P|)/2 = (k/2)\sqrt{|P|}\log|P| = o(|P|)$ bits. Thus, our approach is just as time/space-efficient as the trivial approach of using tailored code. What we gain is the generality as the code works for any p without any changes to the code (the generation of the lookup table for creating the mapping is parameterized by p as well).

8. EXPERIMENTAL RESULTS

We report experimental results on a 50 MB DNA sequence [4], on a 50 MB collection of English text [5], and on randomly generated texts varying the alphabet size. We used a version of the compressed suffix tree `CST` whose theoretical space requirement is $nH_0 + 10n + o(n\log|\Sigma|)$ bits; other variants are possible by adjusting the space/time tradeoff parameters. Here $n(H_0+1)(1+o(1)) + 3n$ comes from the compressed suffix array `CSA`, and $6n + o(n)$ from the other structures. The maximum average slowdown on suffix tree operations is $O(\log n \log|\Sigma|)$ under this tradeoff. The experiments were run on a 2.6GHz Pentium 4 machine with 1GB of main memory. Programs were compiled using g++ (GCC) compiler version 4.1.1 20060525 (Red Hat 4.1.1-1) and -O3 optimization parameters.

We compared the space usage against classical text indexes: a standard pointer-based implementation of suffix trees `ST`, and a standard suffix array `SA` were used. We also compared to the *enhanced suffix array* `ESA` [Abouelhoda et al. 2004]; we used the implementation that is plugged into the `Vmatch` software package [6]. For suffix array construction, we used the `bpr` algorithm [Schürmann and Stoye 2005] that is the currently the fastest construction algorithm in practice.

Figures 2 and 3 report the space requirements on varying length prefixes of the texts. One can see that the achieved space-requirement is attractive; `CST` takes less space than a plain suffix array.

We also measured the maximum space usage for `CSA` and `CST` during the construction. These values (`CSA, max` and `CST, max`) are quite satisfactory; the maximum space needed during the construction is only 1.5 times larger than the final space both on DNA and on English text.

For the time requirement comparison, we measured both the construction time and the usage time (see Figs. 4 and 5). For the latter, we implemented a well-known solution to the *longest common substring (LCSS)* problem using both the classical suffix tree and the compressed suffix tree. For sanity check, we also implemented an $O(n^3)$ ($O(n^2)$ expected case) brute-force algorithm.

The LCSS problem asks to find the longest substring C shared by two given input strings A and B. The solution using suffix tree is evident: Construct the suffix tree of the concatenation $A\$B$, search for the node whose string depth is largest and its subtree contains both a suffix from A and from B. Notice, that no efficient solution without using suffix tree -alike data structures is known.

To get an idea how much different types of algorithms will slow down when using the `CST` instead of `ST`, we measured the average execution times of some key operations. We used the DNA sequence prefix of length 5 million for the experiment

[4] `http://pizzachili.dcc.uchile.cl/texts/dna/dna.50MB.gz`
[5] `http://pizzachili.dcc.uchile.cl/texts/english/english.50MB.gz`
[6] http://www.vmatch.de

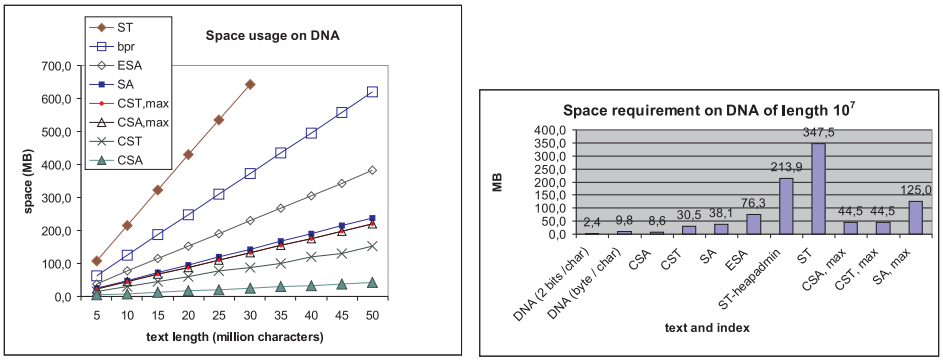

Fig. 2. Comparison of space requirements. We have added the text size to the `SA` and `ST` sizes, as they need the text to function as indexes, whereas `CSA` and `CST` work without. Here `ST-heapadmin` is the space used by suffix tree without the overhead of heap; this large overhead is caused due to the allocation of many small memory fragments. For other indexes, the heap overhead is negligible. Three last values on the right report the maximum space usage during the construction (for `ESA` and `ST` the maximum is the same as the final space requirement).

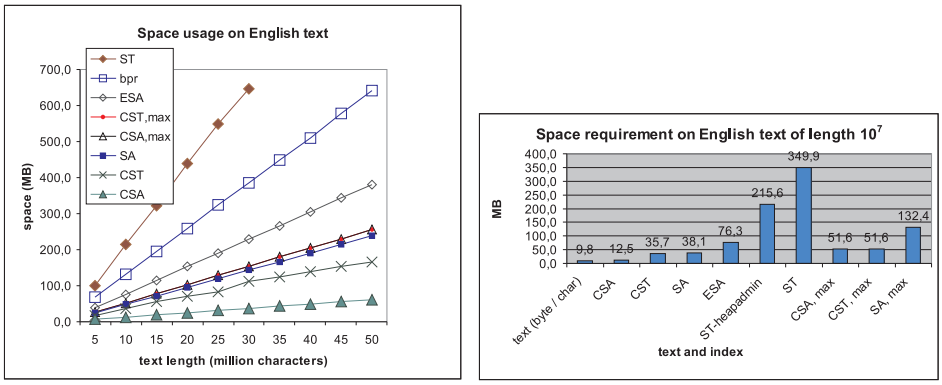

Fig. 3. See Fig. 2 for explanation.

Table I. Average running times (in microseconds) for operations of `ST` and `CST`.

tree operation	isleaf()	parent()	sibling()	edge(*,1)	sl()	lca()	depth()
ST	0,092	0,085	0,085	0,14	0,085	-	-
CST	0,047	0,11	0,22	13,12	11,07	6,66	4,56

and ran each operation repeatedly over the nodes of `ST` and `CST`, respectively, to obtain reliable average running time per operation. The results are shown in Table I.

Notice that `ST` does not support *parent*(), *depth*(), and *lca*() functions. Such functionalities are often assumed in algorithms based on suffix trees. They could be be added to the classical suffix tree as well (two first easily), but this would again increase the space requirement considerably. That is, the space reduction may in practical settings be even more than what is shown in Fig. 2.

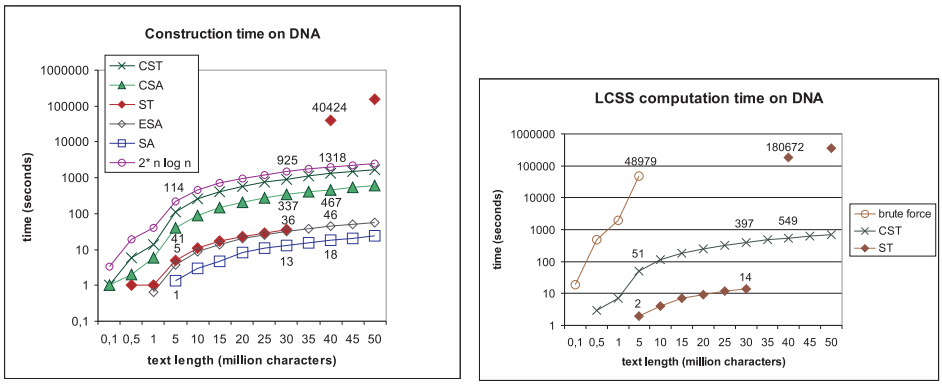

Fig. 4. Comparison of time requirements. For LCSS, we treated the first half of the sequence as A, the second as B. We plotted the expected behaviour, $2n \log n$, for reference. Due to lack of space in the plot, the absolute values belonging to `CST` are shown on top of expected behaviour curve. The more dense sampling of x-values is to illustrate the brute-force algorithm behaviour. After 30MB, suffix tree did not fit into main memory. This constitutes a huge slowdown because of swapping to disk; see the two separate dots in the right-most corners of the plots.

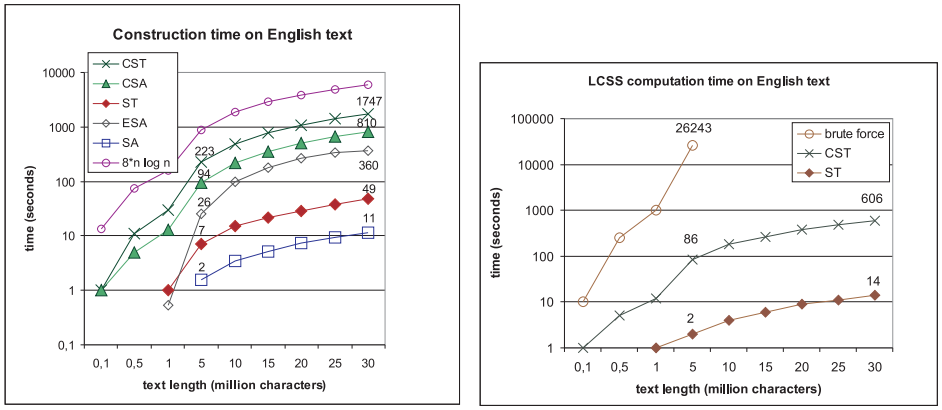

Fig. 5. See Fig. 4. A notable difference to the DNA case is that `ESA` construction seems to have a linear dependency on the alphabet size.

The behaviour of the space requirement and the construction time of `CSA` and `CST` on varying alphabet size is illustrated in Fig. 6.

Finally, both the space and the time requirements of `CST` are heavily dependent on the sample rate used in the underlying compressed suffix array implementation. This effect is illustrated in Fig. 7.

9. FINAL REMARKS

As one can see, our implementation of the compressed suffix tree follows very closely the theoretical proposals. We made only couple of choices towards practical efficiency and very few towards reducing the implementation work; we used previous

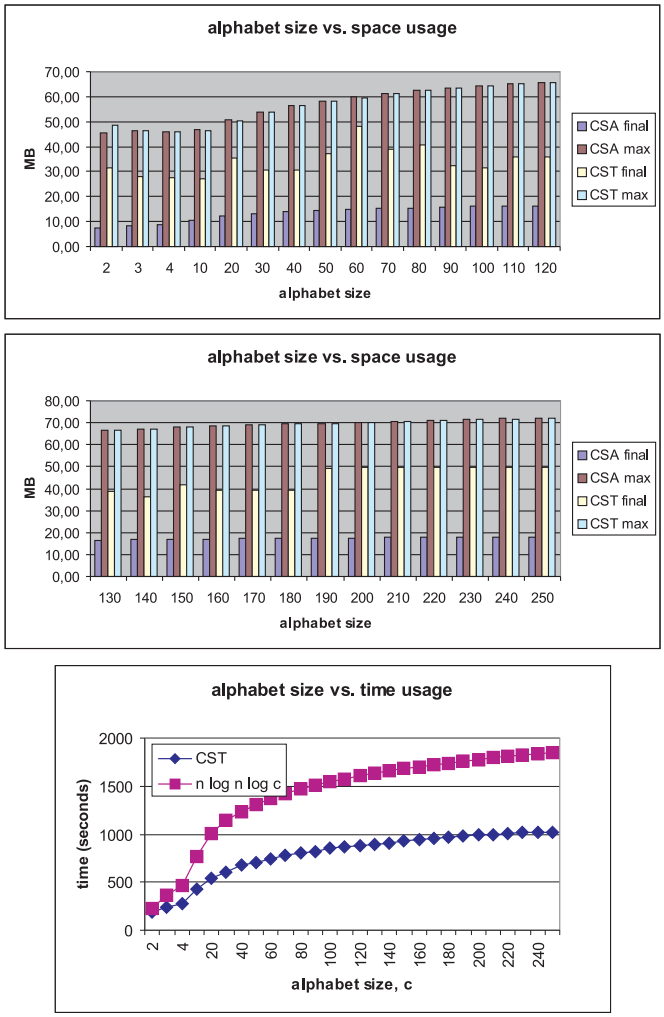

Fig. 6. (Top) Space requirement (final and maximum) of `CST` and `CSA` on randomly generated 10MB files with varying alphabet size. Interestingly, all the other space requirements follow the expected logarithmic dependency of the alphabet size, but the final size of `CST`. A possible explanation is that the number of nodes in the suffix tree varies irregularly on the alphabet size. (Bottom) Time requirement of construction algorithm for `CST` on randomly generated 10MB files with varying alphabet size. One can clearly see that the dependency is logarithmic as stated.

implementations as basis as much as possible. Many of these are also implementations of the best algorithms for the particular tasks.

We are currently working on a finetuned 64-bit version of the compressed suffix tree that can hold the *human genome* indexed in main memory. Preliminary experiments show that this is possible using a computer with 32 GB main memory (which is a significant saving, as for classical suffix tree one would need at least 200 GB main memory). The construction takes currently about 4 days, the final index

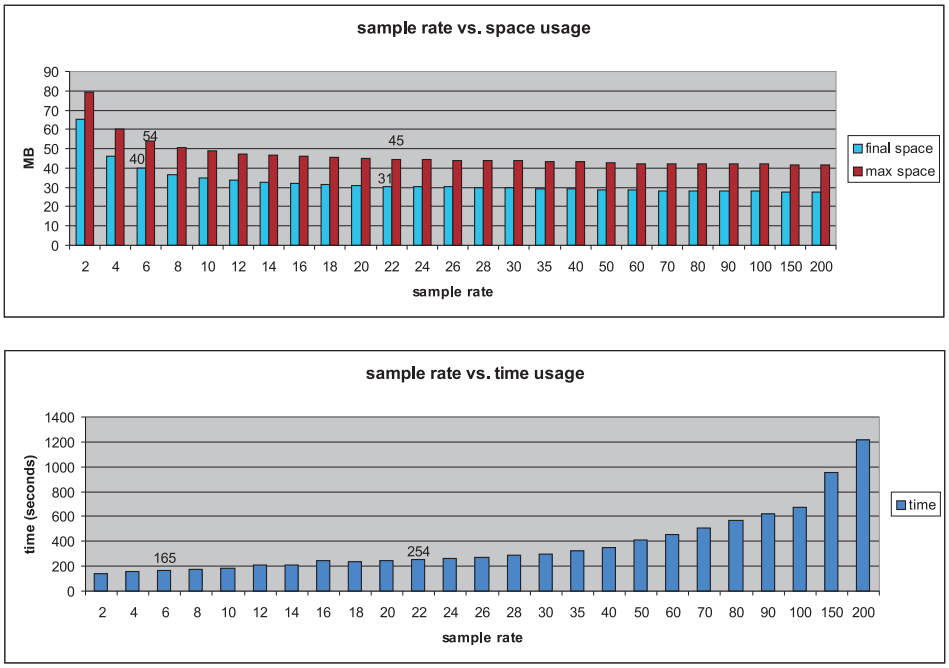

Fig. 7. Space requirement and time requirement of construction of `CST` on varying sample rates used for the underlying compressed suffix array. One can see that with a little compromise in space requirement one can achieve a considerable speed-up in construction algorithm.

occupies about 8.5 GB, and the peak memory usage is 24 GB. We plan to extend the functionalities of the implementation to cover the many useful sequence analysis algorithms developed for suffix trees over the years. We expect to be able to speed up the construction by plugging in some new faster construction algorithms and by taking use of the multiple processor environment. However, a more feasible approach is probably to store the index to disk and recover it back to main memory once accessed (which currently takes about 15 minutes for human genome); this makes possible the maintainance of a genome database, on which more complicated analyses can be executed than what is nowadays possible.

The idea of storing the index to disk leads to the appealing alternative approach; the disk space is not that limited as main memory space, so once can think of storing and using suffix trees on disk. Although our experiments have shown that a typical pointer-based suffix tree is hopelessly slow when running out of main memory, there are ways to adjust the tree to work better on disk, see e.g. [Cheung et al. 2005; Ko and Aluru 2006]. Comparison to these secondary memory suffix trees is left for future work.

REFERENCES

Abouelhoda, M., Kurtz, S., and Ohlebusch, E. 2004. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms 2*, 53–86.

Altschul, S. F., Gish, W., Miller, W., Myers, E. W., and Lipman, D. J. 1990. Basic local alignment search tool. *Journal of Molecular Biology 215,* 3, 403–410.

Apostolico, A. 1985. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*. NATO ISI Series. Springer-Verlag, 85–96.

Burrows, M. and Wheeler, D. 1994. A block sorting lossless data compression algorithm. Tech. Rep. Technical Report 124, Digital Equipment Corporation.

Chan, W.-L., Hon, W.-K., and Lam, T.-W. 2004. Compressed index for a dynamic collection of texts. In *Proc. CPM'04*. LNCS 3109. 445–456.

Cheung, C.-F., Yu, J. X., and Lu, H. 2005. Constructing suffix tree for gigabyte sequences with megabyte memory. *IEEE Transactions on Knowledge and Data Engineering 17,* 1, 90–105.

Clark, D. 1996. Compact pat trees. Ph.D. thesis, University of Waterloo.

Cole, R., Gottlieb, L. A., and Lewenstein, M. 2004. Dictionary matching and indexing with errors and don't cares. In *Symposium on Theory of Computing (STOC)*. 91–100.

Crochemore, M. and Rytter, W. 2002. *Jewels of Stringology*. World Scientific.

Elias, P. 1975. Universal codeword sets and representation of the integers. *IEEE Transactions on Information Theory 21,* 2, 194–20.

Farach-Colton, M. and Bender, M. A. 2000. The lca problem revisited. In *Proc. LATIN'00*. 88–94.

Ferragina, P. and Manzini, G. 2005. Indexing compressed texts. *Journal of the ACM 52,* 4, 552–581.

González, R., Grabowski, S., Mäkinen, V., and Navarro, G. 2005. Practical implementation of rank and select queries. In *Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA'05)*. CTI Press and Ellinika Grammata, Greece, 27–38.

Grossi, R., Gupta, A., and Vitter, J. 2003. High-order entropy-compressed text indexes. In *Proc. SODA'03*. 841–850.

Grossi, R. and Vitter, J. 2006. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing 35,* 2, 378–407.

Gusfield, D. 1997. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press.

Hon, W.-K. 2004. On the construction and application of compressed text indexes. Ph.D. thesis, University of Hong Kong.

Hon, W.-K. and Sadakane, K. 2002. Space-economical algorithms for finding maximal unique matches. In *Proc. CPM'02*. 144–152.

Hon, W.-K., Sadakane, K., and Sung, W.-K. 2003a. Breaking a time-and-space barrier in constructing full-text indices. In *Proc. FOCS'03*. 251.

Hon, W.-K., Sadakane, K., and Sung, W.-K. 2003b. Succinct data structures for searchable partial sums. In *Proc. ISAAC'03*. LNCS 2906. 505–516.

Jacobson, G. 1989. Space-efficient static trees and graphs. In *Proc. 30th IEEE Symp. Foundations of Computer Science (FOCS'89)*. 549–554.

Kärkkäinen, J. 2006. personal communication.

Kasai, T., Lee, G., Arimura, H., Arikawa, S., and Park, K. 2001. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. CPM'01*. Springer Verlag LNCS 2089. 181–192.

Ko, P. and Aluru, S. 2006. Obtaining provably good performance from suffix trees in secondary storage. In *CPM*, M. Lewenstein and G. Valiente, Eds. Lecture Notes in Computer Science, vol. 4009. Springer, 72–83.

Mäkinen, V. and Navarro, G. 2005. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing 12,* 1, 40–66.

Mäkinen, V. and Navarro, G. 2006. Dynamic entropy compressed sequences and full-text indexes. In *Proc. CPM'06*. LNCS 4009. 306–317.

Mäkinen, V. and Navarro, G. 2007a. Implicit compression boosting with applications to self-indexing. In *Proc. SPIRE'07*. LNCS. To appear.

Mäkinen, V. and Navarro, G. 2007b. Rank and select revisited and extended. *Theoretical Computer Science*. To appear.

Manber, U. and Myers, G. 1993. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 935–948.

Manzini, G. 2001. An analysis of the Burrows-Wheeler transform. *Journal of the ACM 48,* 3, 407–430.

McCreight, E. 1976. A space-economical suffix tree construction algorithm. *Journal of the ACM 23,* 2, 262–272.

Munro, I. 1996. Tables. In *Proc. 16th Foundations of Software Technology and Theoretical Computer Science (FSTTCS'96)*. LNCS 1180. 37–42.

Munro, I., Raman, V., and Rao, S. 2001. Space efficient suffix trees. *Journal of Algorithms 39,* 2, 205–222.

Navarro, G. 2004. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms (JDA) 2,* 1, 87–114.

Navarro, G. and Mäkinen, V. 2007. Compressed full-text indexes. *ACM Computing Surveys 39,* 1, Article 2.

Sadakane, K. 2007. Compressed suffix trees with full functionality. *Theory of Computing Systems*. In press, preliminary version available at `http://tcslab.csce.kyushu-u.ac.jp/~sada/papers/cst.ps`.

Schürmann, K.-B. and Stoye, J. 2005. An incomplex algorithm for fast suffix array construction. In *Proc. ALENEX/ANALCO*. 77–85.

Ukkonen, E. 1995. On-line construction of suffix trees. *Algorithmica 14,* 3, 249–260.

Weiner, P. 1973. Linear pattern matching algorithms. In *14th IEEE Annual Symp. on Switching and Automata Theory*. 1–11.

...