Department of Computer Science
Series of Publications C
Report C-2006-37

Engineering a Compressed Suffix Tree Implementation

Kashyap Dixit, Wolfgang Gerlach, Veli Mäkinen, and Niko Välimäki

University of Helsinki
Finland

Engineering a Compressed Suffix Tree Implementation

Kashyap Dixit, Wolfgang Gerlach, Veli Mäkinen, and Niko Välimäki

Department of Computer Science

P.O. Box 68, FIN-00014 University of Helsinki, Finland

kdixit@iitk.ac.in, wgerlach@cebitec.uni-bielefeld.de,
{vmakinen,nvalimak}@cs.helsinki.fi

Technical report, Series of Publications C, Report C-2006-37

Helsinki, September 2006, 18 pages

Abstract

Suffix tree is one of the most important data structures in string algorithms. Especially in biological sequence analysis literature, suffix tree has a central role. 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. We followed the original proposal in spirit, but tailored some internal parts 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. As by-products, we develop a method to create Succinct Suffix Arrays (Mäkinen & Navarro, CPM 2005) directly from Burrows-Wheeler transform and a space-efficient version of *suffixes-insertion* algorithm to build balanced parantheses representation of suffix tree from LCP information.

Computing Reviews (1998) Categories and Subject Descriptors:

E.1,	Data: Data Structures — trees
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, sorting and searching

General Terms:
Algorithms, Data structures

Additional Key Words and Phrases:
Combinatorial pattern matching, Data compression, Full-text indexing, Algorithm engineering

1 Introduction

Myriad non-trivial combinatorial questions concerning strings turn out to have efficient solutions via extensive use of *suffix trees* [2]. 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 [35, 27, 34].

Bioinformatics is a field where suffix trees would seem to have the strongest potential; unlike the natural language texts formed by words and delimiters, 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. [15]).

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 [6]). In practice, the highly popular software tools like BLAST [1] 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 $100MB$, its suffix tree may require $5GB$ 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. In the sequel, we express all space requirements in bits.

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* [26]. 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.

A recent twist in the development is the rise of *abstract data structures*; the operations supported by a data structure are identified and the best possible implementation is seeked for that supports those operations. This line of development has led to *compressed suffix arrays* [14, 10, 32, 13, 11]. These data structures take, in essense, $n \log |\Sigma|(1+o(1))$ 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. See [31] for a survey on these structures.

Very recently Sadakane [33] 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, Raman, and Rao [29]. In addition to these navigational operations, suffix trees have several other useful operations such as suffix links, constant time lowest common ancestor (lca) queries, and pattern search capabilites. 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 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 [33]. 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 reasonable space-efficiency (see Sect. 8 for some numbers).

A problem related to practical implementation is how to construct the compressed suffix tree without using too much extra space at construction time. There are many other 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 parantheses representation of suffix tree directly from compressed suffix array. Tasks (1), (3) and (4) have been considered in [18] and later improved in [19] so as to obtain an $O(n \log^{\epsilon} n)$ time algorithm to construct compressed suffix trees. Task (2) is related to our choice of implementing compressed suffix arrays using structures evolved from FM-index [10], and is tackled in this paper. Also for task (3) our solution variates slightly from [18] as we build on top of the suffixes-insertion algorithm [7] and they build on top of the post-order traversal algorithm of [22]. The final time-requirement of our implementation is $O(n \log n \log |\Sigma|)$, being reasobaly close to the best current theoretical result [19].

The outline of the article is as follows. Section 2 gives the basic definitions and a very cursory overview of Sadakane's structure. [1] Section 3 explains how

[1]Although it is technically possible to follow this paper without understanding Sadakane's structure, we encourage the reader to study [33] to obtain a deeper understanding of the

we implemented compressed suffix arrays (related to task (1)) and provides a solutions to task (2). Section 4 extrapolates the solution mentioned in [18] for task (3). Section 5 gives an overview of balanced parantheses 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. Some final remarks are given in Sect. 8.

The software package is available at `http://www.cs.helsinki.fi/group/suds/`.

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 $T_{i \ldots j} = t_i t_{i+1} \cdots t_j$, where $1 \leq i \leq j \leq n$. A *suffix* of T is any substring $T_{i \ldots n}$, where $1 \leq i \leq n$. A *prefix* of T is any substring $T_{1 \ldots j}$, 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 1 *(Adopted from [15]) The* keyword trie *for set $\mathcal{P}$ of patterns 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 pattern in $\mathcal{P}$.*

Definition 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 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 $T_{l \ldots r}$ gives the label.*

The definition for a *keyword tree* is analogous.

Definition 4 *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.

Definition 5 *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.*

context.

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 arbitrary two suffixes $T_{i\ldots n}$ and $T_{j\ldots n}$: Let v and w be the two leaves of suffix tree have path labels $T_{i\ldots n}$ and $T_{j\ldots n}$, 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 6 *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 [33] shows how to implement each operation listed in Def. 6 by means of a sequence of operations on (1) compressed suffix array, (2) *lcp*-array [2], (3) balanced parantheses representation of suffix tree hierarchy, and (4) a structure for *lca*-queries. In the following sections we explain how we implemented those structures.

[2]Sadakane [33] uses name *Height*-array

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 $SA[1 \ldots n]$ be a table such that $T_{SA[i] \ldots n}$ gives the i-th smallest suffix in lexicographic order. Notice that this table can be filled by a depth-first traversal on suffix tree following its edges in 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 e.g. [14, 10, 32]. The following captures typical suffix arrays operations on an abstract level.

Definition 7 *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* $T[SA[i] \ldots SA[i]+l-1]$*.*

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

Definition 8

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

3.1 Our Implementation

We used *Succinct Suffix Array* (SSA) of [23] to implement the abstract suffix array operations. The base structure is the *wavelet tree* [13] build on the *Burrows-Wheeler transform* [3].[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.

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 9

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

It can be shown [10] that LF-mapping can computed by the means of T^{bwt}:

[3]For background on these techniques, see a recent survey [31].

Lemma 1 ([10]) *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 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 [23, 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 $i' \times R$ its location in suffix array explicitly: `sampledSAinverse`$[i'] = j$ such that $SA[j] = i' \times R$. Now, the *substring*(i, l)-query can be supported as follows. We compute the smallest integer i' such that $i + l \leq i' \times R$. Then substring $T_{i \ldots i' \times R-1}$ is retrieved in reverse order by applying LF-mapping repeatedly: $t_{i' \times R-1} = T^{bwt}[j]$, $t_{i' \times R-2} = T^{bwt}[LF[j]]$, $t_{i' \times R-3} = T^{bwt}[LF[LF[j]]]$, $\ldots$. 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 $i' \times R - i$ times starting from `sampledSAinverse`$[i']$. The index j reached in the end has the desidered 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 `sampledSA` such that `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*$(i) =$ `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 [20].

Finally, to answer $\Psi(i)$, we first apply $j =$ *lookup*(i), then apply LF-mapping starting from `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 [23], so that the structure takes $\frac{2n}{R} \log n + n(H_0 + 2)(1 + o(1))$ bits of space and supports all the abstract suffix array operations in $O(R * H_0)$ average time. (Worst case $O(R * \log n)$. Use $R + l$ instead of R for *substring*(i, l) function time requirement.) Here H_0 is the *zeroth order entropy* of T. Recall that $H_0 \leq \log |\Sigma|$. Fixing any $R = \Omega(\frac{\log n}{\log |\Sigma|})$, the structure takes $O(n \log |\Sigma|)$ bits.

3.2 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 $nH_0 + o(n \log |\Sigma|)$ bits of space by using the dynamic self-index explained in [24]. 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 [4] (see also [24, Sect. 3.2]). Using this inside the dynamic wavelet trees of [24], one obtains the claimed result (see the paragraph just before Sect. 6 in [24]). 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 [23].

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 1 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 $i \times R$, 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` corrrectly. 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 $i \times R$, 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 [17] 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 $x_1 = y_1$, $x_2 = y_2$, $\ldots$, $x_j = y_j$, but $x_{j+1} \neq y_{j+1}$. Sadakane [33] 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. That is, 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 extra space [28, 5].

Computing $lcp[i]$ can now be done as follows. Compute $k = SA[i]$ using $lookup(i)$ of compressed suffix array. Value $lcp[i]$ equals $select_1(H,k) - k$.

4.1 Space-efficient Construction via Kasai et al. Algorithm

Kasai et al. [22] gave a linear time algorithm to construct *lcp*-array given suffix array 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 encoding H [18].

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 $lcp[SA^{-1}[i+1]] \geq lcp[SA^{-1}[i]] - 1$ [22]. 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 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 text, 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 Parantheses

The *balanced paranthesis* 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 [29]. 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 paranthesis representation takes at most $4n$ bits.

Munro, Raman, and Rao [29] explain how to simulate tree traversal by means of P. After building several structures of sublinear size, one can go e.g. from node to its first child, from node to its next sibling, and from node to its parent, each in constant time. Sadakane [33] 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 paranthesis 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 paranthesis for the open paranthesis at position i. Function $enclose(P, i)$ returns the open paranthesis position of the parent of the node whose open paranthesis 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 children in the subtree of v is easily calculated: As pattern $p = 10$ represents an open-close paranthesis pair, i.e. a child node, the amount of children 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 [12]. There *rank* is the constant time solution of Clark [5], but *select* is implemented by binary search on *rank* values (the constant time solution [5] is inferior to this on practical inputs [12]). 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 [30] that are based on [29]; these are faster in practice than the original, but worst case is raised from constant to $O(\log \log n)$.

5.2 Space-efficient Construction via LCP Information

To build balanced parantheses sequence of suffix tree space-efficiently one can not 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 parantheses sequence incrementally. Very similar algorithm is given in [18]; we will consider the differences in the end of the section.

Recall from [7, Theorem 7.5, p. 97] the *suffixes-insertion* algorithm to construct suffix tree from LCP information: The algorithm adds suffixes in lexicographic order into a tree, having the keyword tree of suffixes $T_{SA[1]\ldots n}, T_{SA[2]\ldots n}, \ldots, T_{SA[i]\ldots n}$ ready after i-th step. Suffix $T_{SA[i+1]\ldots n}$ 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 $T_{SA[i]\ldots n}$) whose string depth is smaller or equal to $lcp[i]$ is seeked for. If the depth is equal, then a new leaf (corresponding to suffix $T_{SA[i+1]\ldots n}$) is created as its child. Otherwise, its outgoing rightmost edge is splitted, a new internal node is inserted in between, and the leaf corresponding to suffix $T_{SA[i+1]\ldots n}$ 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.

The idea is that at each step of the algorithm we have the balanced parantheses representation of the current tree ready. Unfortunately, the parantheses structure does not change sequentially, so we need to maintain it using a dynamic bitvector allowing insertions of bits (open/close parantheses) 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 [4, 24]. In addition to the balanced parantheses 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 sequences $E(i)$ during the algorithm in 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 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 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 (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 new internal node and 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.

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 [8]. 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. [25]). 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 reading by decoding the first code in constant time and shifting the starting

```
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 $sdepth = n - E.Sum(j) \leq lcp$; {$E.Sum(j) = \sum_{j'=1}^{j} e_{j'}$ }
        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 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;
```

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

position to the right accordingly. Identically an insertion is done by shifting the starting position to the left to make 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 working space.

Improving running time to linear. Finally, the time requirement can be improved to linear by replacing the dynamic bit vector by a patching technique [21]: 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.

Comparison to Hon and Sadakane solution. Hon and Sadakane [18] describe a very similar algorithm. They build on top of an algorithm in [22] 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 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 take $O(n \log n)$ 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. The 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 [16, 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 $P.Insert(\texttt{(}, r)$ 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 $P' =$ `()())()()))` and $P'' =$ `((()()(()()` 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, $P =$ `((()())(()()))`. Setting the parantheses in the other order between the second and third leaf would yield another leaf, which is not allowed.

Our implementation. In our implementation, we do not yet 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 balanced parantheses construction remains $O(n \log n)$ even after applying one of the speed ups.

6 Lowest Common Ancestor Structure

Farach-Colton and Bender [9] 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 [33] 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 parantheses representation of previous section, adding lookup tables taking $o(n)$ bits.

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 using `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. A novel 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 $I(P,p) = I[1\dots|P|]$ 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 $I(\alpha,p) = I[1\dots|\alpha|]$, 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 of [20, 28, 5]. 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).

7.1 Correctness.

The most difficult goal to achieve in the implementation (in general) is the correctness. We wanted to avoid the typical setting of doing one month implementation and three months bug fixing. We adopted a strategy often associated with *Extreme Programming*, namely, we produced each week a working release. In this particular case this strategy suited our purposes perfectly.

We started the project by taking an already existing implementation of a suffix tree. We implemented the first version of the abstract suffix tree (Def. 6) by supporting the functions via the classical suffix tree. Functions such as lca were first implemented by trivial scanning. This was our first release. The work continued by converting the suffix tree hierarchy into balanced parantheses form, and implementing the required traversal operations by trivial scanning. Abstract suffix array was implemented first by normal suffix array. This refinement continued gradually, so that each week we had a new release whose correctness could be compared to the previous release. After two months, we had a fully working implementation of compressed suffix tree ready. One more month was used in implementing the space-efficient construction algorithms. The complete work required equals about three months from two undergraduate students.

8 Final remarks

As one can see, our implementation of 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. The former is mainly because we do not yet have executed extensive experimentation to guide the choice of practical alternatives. For the latter matter, we used previous implementations as basis as much as possible. Many of these are also implementations of the best algorithms for the particular tasks. Yet there exists some algorithms that could be used to speed up our construction algorithm. Plugging in those algorithms is left for future work.

The most important future task is to experiment the new space/time tradeoffs achieved with respect to classical suffix trees. Preliminary experiments show that the space-efficiency is very appealing: The compressed suffix tree for a 10MB DNA sequence requires 32MB. This is already less than what a suffix array takes: 40MB, not to talk about a standard suffix tree that takes at least 160MB at the same input. Measuring in bits, the DNA sequence could be stored in 20M bits (2 bits / character). The compressed suffix tree takes 256M bits, that is, about 13 times more than the DNA sequence. The theoretical bound of our implementation is $n \log |\Sigma| + 8n = 10n$ plus sublinear terms. Assymptotically we should then be using about 5 times more space than a succinctly encoded DNA sequence. This shows that the sublinear structures take over the half of the space with files of size 10MB. A major research question of practical interest is thus to improve the constants in the sublinear structures.

References

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

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

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

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

[5] D. Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, 1996.

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

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

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

[9] M. Farach-Colton and M. A. Bender. The lca problem revisited. In *Proc. Latin American Theoretical Informatics (LATIN)*, pages 88–94, 2000.

[10] P. Ferragina and G. Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005.

[11] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representation of sequences and full-text indexes. *ACM Transactions on Algorithms*, 2006. To appear. Preliminary versions in *Proc. SPIRE 2004* and Tech. Rep. TR/DCC-2004-5, Dept. of Computer Science Univ. of Chile, `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/sequences.ps.gz`.

[12] Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. Practical implementation of rank and select queries. In *Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA'05)*, pages 27–38, Greece, 2005. CTI Press and Ellinika Grammata.

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

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

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

[16] W.-K. Hon. *On the Construction and Application of Compressed Text Indexes*. PhD thesis, University of Hong Kong, 2004.

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

[18] Wing-Kai Hon and Kunihiko Sadakane. Space-economical algorithms for finding maximal unique matches. In Alberto Apostolico and Masayuki Takeda, editors, *CPM*, volume 2373 of *Lecture Notes in Computer Science*, pages 144–152. Springer, 2002.

[19] Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Breaking a time-and-space barrier in constructing full-text indices. In *FOCS '03: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, page 251, Washington, DC, USA, 2003. IEEE Computer Society.

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

[21] J. Kärkkäinen. personal communication, 2006.

[22] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. Annual Symposium on Combinatorial Pattern Matching (CPM)*, Springer Verlag LNCS 2089, pages 181–192, 2001.

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

[24] V. Mäkinen and G. Navarro. Dynamic entropy compressed sequences and full-text indexes. In *Proc. Annual Symposium on Combinatorial Pattern Matching (CPM)*, Springer Verlag LNCS 4009, pages 306–317, 2006.

[25] V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 2006. To appear.

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

[27] E. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

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

[29] I. Munro, V. Raman, and S. Rao. Space efficient suffix trees. *Journal of Algorithms*, 39(2):205–222, 2001.

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

[31] G. Navarro and V. Mäkinen. Compressed full-text indexes. Technical Report TR/DCC-2006-6, Department of Computer Science, University of Chile, Chile, April 2006. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/survcompr2.ps.gz`. Submitted to a journal.

[32] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

[33] K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 2006. To appear, preliminary version available at `http://tcslab.csce.kyushu-u.ac.jp/~sada/papers/cst.ps`.

[34] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

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