Engineering a Compressed Suffix Tree Implementation

Niko Välimäki[1], Wolfgang Gerlach[2], Kashyap Dixit[3] ⋆, and Veli Mäkinen[1] ⋆

[1] Department of Computer Science, University of Helsinki, Finland.
{nvalimak,vmakinen}@cs.helsinki.fi
[2] Technische Fakultät, Universität Bielefeld, Germany.
wgerlach@cebitec.uni-bielefeld.de
[3] Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur, India
kdixit@iitk.ac.in

Abstract. 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 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 1.4 times the final product size.

1 Introduction

Myriad non-trivial combinatorial questions concerning strings turn out to have efficient solutions via extensive use of *suffix trees* [2]. As a theoretical tool, suffix trees have a fundamental role in plethora of algorithmic

⋆ Funded by the Academy of Finland under grant 108219.

results in the area of string matching and sequence analysis. *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. [13]. 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. A bottleneck to the wide-spread use of suffix trees in Bioinformatics 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* [20]. 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 [1].

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* [12, 9] (see [24] 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 slow-

[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/

downs. 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 [25] 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 [22]. 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 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 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 [25]. In addition, we considered the issue of space-efficient construction, studying the following subtasks: (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 [15] and later improved in [16] 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 [9], and is tackled in this paper. Also for task (3) our solution variates slightly from [15] as we build on top of the suffixes-insertion algorithm [6] and they build on top of the post-order traversal algorithm of [17]. The final time-requirement of our implementation is $O(n \log n \log |\Sigma|)$, being reasonably close to the best current theoretical result [16].

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 a solutions to task (2). Section 4 describes the solution hinted in [15] 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. We conclude with some illustrative experimental results in Sect. 7.

The software package can be downloaded from `http://www.cs.helsinki.fi/group/suds/cst/`. Also a technical report is available there that contains the full implementation details that are omitted here for the lack of space.

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 $T_{i \dots j} = t_i t_{i+1} \cdots t_j$, where $1 \leq i \leq j \leq n$. A *suffix* of T is any substring $T_{i \dots n}$, where $1 \leq i \leq n$. A *prefix* of T is any substring $T_{1 \dots j}$, where $1 \leq j \leq n$.

Definition 1. *(Adopted from [13]) The* keyword trie *for 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. *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 $T_{l \dots r}$ 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 arbitrary two suffixes $T_{i \dots n}$ and $T_{j \dots n}$: Let v and w be the two leaves of suffix tree have path labels $T_{i \dots n}$ and $T_{j \dots n}$, 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 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.*

2.1 Overview of Compressed Suffix Tree

Sadakane [25] shows how to implement each operation listed in Def. 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 $SA[1 \ldots n]$ 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 [24]. The following captures typical suffix array operations on an abstract level.

[2] Sadakane [25] uses name *Height*-array.

Definition 5. *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 $T[SA[i] \dots SA[i]+l-1]$.*

3.1 Our Implementation

We used *Succinct Suffix Array* (SSA) of [18] to implement the abstract suffix array operations. The base structure is the *wavelet tree* [11] build on the *Burrows-Wheeler transform* [3]. 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: $LF(i) = i'$ such that $SA[i'] = SA[i]-1$.

It can be shown [9] that LF-mapping can computed by the means of T^{bwt}:

Lemma 1 ([9]). *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 $C[1 \dots |\Sigma|]$ 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 [18, 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.

Rest of the abstract suffix array operations can be supported by storing sampled suffix array values (and sampled inverse suffix array values) and using LF-mapping to simulate those operations. These additional structures occupy together $\frac{2n}{R} \log n + n(1+o(1)$ bits, where R is the sample rate used. See the full version of this article for details or consult [24], where these are explained for *lookup*() and *substring*(); Ψ() and *inverse*() are analogous.

In our implementation, we use the Huffman-tree shape as advised in [18], 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 $O(R \cdot \log n)$.) 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.

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.

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 [19]. 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 [19, Sect. 3.2]). Using this inside the dynamic wavelet trees of [19], one obtains the claimed result (see the paragraph just before Sect. 6 in [19]). 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 [18].

The rest of the structures to support abstract suffix array operations can be constructed afterward in $O(n \log |\Sigma|)$ time using LF-mapping. We leave the details for the full version.

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 [25] 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.

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 [21].

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

Kasai et al. [17] gave a linear time algorithm to construct lcp-array given SA. One can easily modify Kasai et al. algorithm to directly give H [15]. 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 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 [22]. 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, Raman, and Rao [22] 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 [25] 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.

5.1 Our Implementation

We used the existing *rank* and *select* implementations that are explained and experimented in [10]. We leave for the full version the explanation how we modified these solutions 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 [23] that are based on [22].

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. Very similar algorithm is already given in [15], and hence we only sketch the main idea and differences.

Recall from [6, 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 $T_{SA[1]...n}, T_{SA[2]...n}, \ldots, T_{SA[i]...n}$ 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.

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 [4, 19]. 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. It happens that the string and node depths are monotonic on the rightmost path, and during the algorithm one only needs to modify them from the tail. Such monotonic sequences of numbers, whose sum is $O(n)$, can be stored in $O(n)$ bits using integer codes like Elias δ-code [7]. We leave the details to the full version. Hence we can construct the balanced parentheses sequence in $O(n \log n)$ time using $O(n)$ bits working space.

The difference to Hon and Sadakane algorithm [15] is mainly on the conceptual level. They build on top of an algorithm in [17] that simulates the post-order traversal of suffix tree given the *lcp*-values. When making that algorithm space-efficient, the end result is very close to ours.[3]

Implementation remark. 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 [8] 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 [25] 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.

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 $O(K)$.

[3] The handling of P is not quite complete in [15]: in some extreme cases, their algorithm might need $O(n \log n)$ bits space. This can be fixed by adding a similar handling of node depths as in our algorithm (their algorithm already has very similar handling of string depths). Alternatively, Hon [14, page 59] describes another solution that goes around this problem.

7 Experimental Results

We report some illustrative experimental results on a 50 MB DNA sequence [4]. 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` [1]; we used the implementation that is plugged into the `Vmatch` software package[5]. For suffix array construction, we used the `bpr` algorithm [26] that is the currently the fastest construction algorithm in practice.

Figure 1 reports the space requirements on varying length prefixes of the text. 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 construction is only 1.4 times larger than the final space.

For the time requirement comparison, we measured both the construction time and the usage time (see Fig. 2). 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.

[4] `http://pizzachili.dcc.uchile.cl/texts/dna/dna.50MB.gz`

[5] http://www.vmatch.de

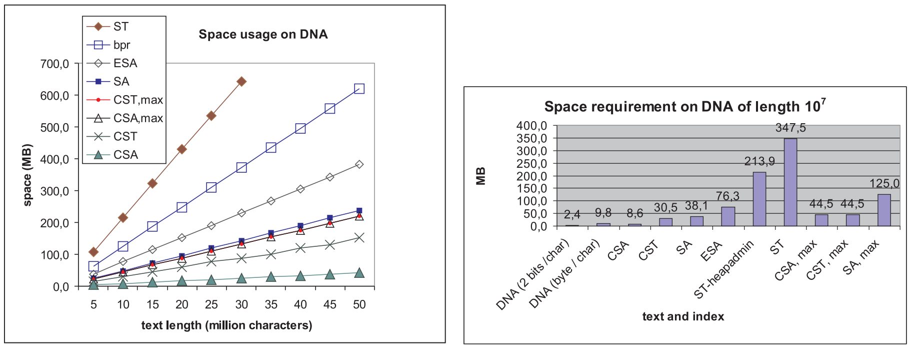

Fig. 1. Comparison of space requirements. We have added the text size to the `SA` and `ST` sizes, as they need the text to function as indexes, whereas `CSA` and `CST` work without. Here `ST-heapadmin` is the space used by suffix tree without the overhead of heap; this large overhead is caused due to 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).

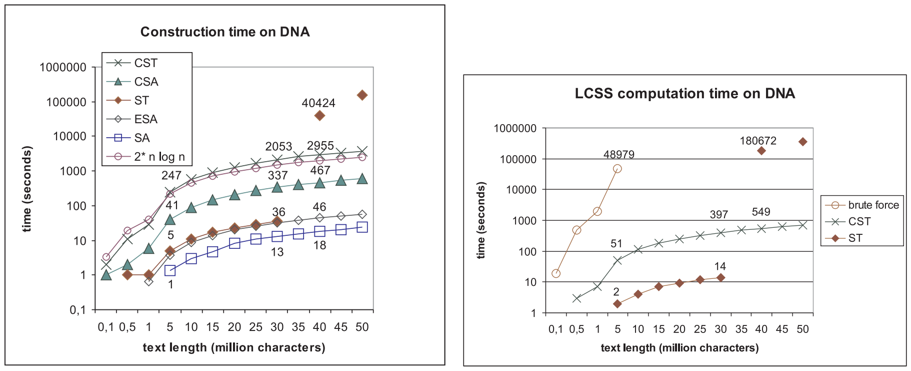

Fig. 2. Comparison of time requirements. For LCSS, we treated the first half of the sequence as A, the second as B. We plotted the expected behaviour, $2n \log n$, for reference. The more dense sampling of x-values is to illustrate the brute-force algorithm behaviour. After 30MB, suffix tree did not fit into main memory. This constitutes a huge slowdown because of swapping to disk.

Finally, 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 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 1.

Table 1. Average running times (in microseconds) for operations of `ST` and `CST`.

tree operation	edge(*,1)	sl()	isleaf()	parent()	depth()	lca()
ST	$0,14$	$0,09$	$0,09$	-	-	-
CST	$13,12$	$11,07$	$0,05$	$0,11$	$4,56$	$6,66$

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. 1.

These experiments show that even though the compressed suffix tree is significantly slower than a classical suffix tree, it has an important application domain on genome-scale analysis tasks; when memory is the bottleneck for using classical suffix trees and brute-force solutions are too slow, compressed suffix trees can provide a new opportunity to solve the problem at hand without running out of space or time. [6]

References

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

[6] Notice that our suffix tree implementation is not tuned for secondary memory use. When such tuning is done, the slowdown gets very close to that of compressed suffix trees in main memory [5]. Comparison between these two approaches is left for future work.

5. C.-F. Cheung, J. X. Yu, and H. Lu. Constructing suffix tree for gigabyte sequences with megabyte memory. *IEEE Transactions on Knowledge and Data Engineering*, 17(1):90–105, 2005.
6. M. Crochemore and W. Rytter. *Jewels of Stringology.* World Scientific, 2002.
7. P. Elias. Universal codeword sets and representation of the integers. *IEEE Transactions on Information Theory*, 21(2):194–20, 1975.
8. M. Farach-Colton and M. A. Bender. The lca problem revisited. In *Proc. LATIN'00*, pages 88–94, 2000.
9. P. Ferragina and G. Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005.
10. R. González, Sz. Grabowski, V. Mäkinen, and G. 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.
11. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA'03*, pages 841–850, 2003.
12. 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.
13. D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.
14. W.-K. Hon. *On the Construction and Application of Compressed Text Indexes.* PhD thesis, University of Hong Kong, 2004.
15. W.-K. Hon and K. Sadakane. Space-economical algorithms for finding maximal unique matches. In *Proc. CPM'02*, pages 144–152, 2002.
16. W.-K. Hon, K. Sadakane, and W.-K. Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proc. FOCS'03*, page 251, 2003.
17. 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. CPM'01*, Springer Verlag LNCS 2089, pages 181–192, 2001.
18. V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.
19. V. Mäkinen and G. Navarro. Dynamic entropy compressed sequences and full-text indexes. In *Proc. CPM'06*, LNCS 4009, pages 306–317, 2006.
20. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, pages 935–948, 1993.
21. I. Munro. Tables. In *Proc. 16th Foundations of Software Technology and Theoretical Computer Science (FSTTCS'96)*, LNCS 1180, pages 37–42, 1996.
22. I. Munro, V. Raman, and S. Rao. Space efficient suffix trees. *Journal of Algorithms*, 39(2):205–222, 2001.
23. G. Navarro. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms (JDA)*, 2(1):87–114, 2004.
24. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 2007. To appear, preliminary version available at `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/survcompr2.ps.gz`.
25. K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 2007. To appear, preliminary version available at `http://tcslab.csce.kyushu-u.ac.jp/~sada/papers/cst.ps`.
26. K.-B. Schürmann and J. Stoye. An incomplex algorithm for fast suffix array construction. In *Proc. ALENEX/ANALCO*, pages 77–85, 2005.