Relative Compressed Suffix Trees*

Travis Gagie[1], Gonzalo Navarro[2], Simon J. Puglisi[1], and Jouni Sirén[3]

[1] Department of Computer Science, University of Helsinki, Finland
{gagie,puglisi}@cs.helsinki.fi

[2] Center for Biotechnology and Bioengineering, Department of Computer Science, University of Chile, Chile
gnavarro@dcc.uchile.cl

[3] Wellcome Trust Sanger Institute, United Kingdom
jouni.siren@sanger.ac.uk

Abstract. This work investigates the use of mutual information between data structures for similar datasets to represent the structures in less space. If two data structures are similar to each other, one of them can probably be represented by its differences to the other, while still supporting efficient queries. Such relative data structures may find use in bioinformatics, where the genomes of individuals of the same species are very similar to each other. More formally, assume that we have similar datasets R and S. If we build data structure D for the datasets, we will likely see that D(R) and D(S) have low relative entropy. Given D(R), we can probably represent D(S — R) (denoting D(S) relative to dataset R) in small space, while still supporting the functionality of D efficiently. Then, given D(R) and D(S — R), we can either simulate D(S) directly, or decompress it for faster queries. A similar approach may also allow the construction of D(S) and D(S — R) efficiently, given D(R), R, and the differences between datasets S and R. Our work clearly has links to persistent data structures and can be thought of as a special case where only the initial state and the final state are preserved, the final state being the net result of potentially many individual modifications, all of which would be represented by a persistent data structure.

1 Introduction

The main topic of the paper will be the relative CST, but we also have the RLZ bitvector and the relative FM-index for read collections. The RLZ bitvector is a nice idea that works well in practice, if we can just find applications for it. Sorting suffixes in lexicographic order amplifies the differences between the sequences, so it's probably going to be something unrelated to suffix trees and suffix arrays.

Relative data compression is a well-established topic. Version control systems store revisions of files as insertions and deletions to earlier revisions. In bioinformatics, individual genomes are often represented by listing their differences to the reference genome of the same species. More generally, we can use relative Lempel-Ziv (RLZ) parsing [10] to represent a text as a concatenation of substrings of a related text.

Compressed data structures for repetitive data. Given similar datasets S1,...,Sr, the data structure D(S1,...,Sr) is often repetitive. If we compress these repetitions, we can represent and use the data structure in much smaller space. Compressed data structures achieve better compression than relative data structures, because they can take advantage of the redundancy between all datasets, instead of just between the current dataset and the reference dataset. The price is less flexibility, as the encoding of each dataset may depend on all the other datasets. While the construction of compressed data structures for multiple datasets requires dedicated algorithms and often also significant computational resources, we can easily distribute the construction of relative data structures to multiple systems, as well as add and remove datasets.

Persistent data structures preserve the state of the data structure before each operation. Relative data structures can be seen as a special case of persistent data structures that preserves

* This work is funded in part by: by Fondecyt Project 1-140796; Basal Funds FB0001, Conicyt, Chile; by Academy of Finland grants 258308 and 250345 (CoECGR); and by the Wellcome Trust grant [098051].

only the initial and the final state, with more emphasis on space-e"ciency. Also, while research on persistent data structures concentrates on structures that can be dynamically updated, my emphasis is on static data structures that are smaller and faster to use.

- motivation for indexing, suffix trees
- repetitive data
- related work

2 Background

A *string* $S[1,n] = s_1 \dots s_n$ is a sequence of *characters* over an *alphabet* $\Sigma = \{1, \dots, \sigma\}$. *Binary* sequences are sequences over alphabet $\{0,1\}$. For indexing purposes, we often consider *text* strings $T[1,n]$ that are terminated by an *endmarker* $T[n] = \$ = 0$ not occurring elsewhere in the text. For any subset $A \subseteq [1,n]$, we define the *subsequence* S_A of string S as the concatenation of characters $\{s_i \mid i \in A\}$ in the same order as in the original string. Contiguous subsequences $S[i,j]$ are called *substrings*. Substrings of type $S[1,j]$ and $S[i,n]$ are called *prefixes* and *suffixes*, respectively. We define the *lexicographic order* among strings in the usual way.

2.1 Full-text indexes

The *suffix tree* (ST) [24] of text T is a trie containing the suffixes of T, with unary paths compacted into single edges. As there are no unary internal nodes in the suffix tree, there can be at most $2n-1$ nodes, and the suffix tree can be stored in $\mathsf{O}(n \log n)$ bits. In practice, this is at least $10n$ bytes for small texts [14], and more for large texts as the pointers grow larger. If v is a node of a suffix tree, we write $\pi(v)$ to denote the label of the path from the root to node v.

Suffix arrays (SA) [16] were introduced as a space-efficient alternative to suffix trees. The suffix array $\mathsf{SA}[1,n]$ of text T is an array of pointers to the suffixes of the text in lexicographic order. In its basic form, the suffix array requires $n \log n$ bits in addition to the text, but its functionalities are more limited than those of the suffix tree. In addition to the suffix array, many algorithms also use the *inverse suffix array* $\mathsf{ISA}[1,n]$, with $\mathsf{SA}[\mathsf{ISA}[i]] = i$ for all i.

Let $\mathsf{lcp}(S_1, S_2)$ be the length of the longest common prefix of strings S_1 and S_2. The *longest-common-prefix* (LCP) *array* [16] $\mathsf{LCP}[1,n]$ of text T stores the lengths of lexicographically adjacent suffixes of T as $\mathsf{LCP}[i] = \mathsf{lcp}(T[\mathsf{SA}[i-1],n], T[\mathsf{SA}[i],n])$. Let v be an internal node of the suffix tree, $\ell = |\pi(v)|$ the *string depth* of node v, and $\mathsf{SA}[sp,ep]$ the suffix array interval corresponding to node v. The following properties hold for the LCP *interval* $\mathsf{LCP}[sp,ep]$: i) $\mathsf{LCP}[sp] < \ell$; ii) $\mathsf{LCP}[i] \geq \ell$ for all $sp < i \leq ep$; iii) $\mathsf{LCP}[i] = \ell$ for at least one $sp < i \leq ep$; and iv) $\mathsf{LCP}[ep+1] < \ell$ [2].

Abouelhoda et al. [2] simulated the suffix tree by using the suffix array, the LCP array, and a representation of the suffix tree topology based on the LCP intervals, paving way for more space-efficient suffix tree representations.

2.2 Compressed text indexes

Sequences supporting rank and select queries are the main building block of compressed text indexes. If S is a sequence, we define $\mathsf{rank}_c(S,i)$ to be the number of occurrences of character c in the prefix $S[1,i]$, and $\mathsf{select}_c(S,j)$ is the position of the occurrence of rank j in sequence S. A *bitvector* is a representation of a binary sequence B supporting fast rank and select queries. *Wavelet trees* (WT) [12] use bitvectors to support rank and select on strings.

The *Burrows-Wheeler transform* (BWT) [6] is a reversible permutation $\mathsf{BWT}[1,n]$ of text T. It is defined as $\mathsf{BWT}[i] = T[\mathsf{SA}[i]-1]$ (with $\mathsf{BWT}[i] = T[n]$, if $\mathsf{SA}[i] = 1$). Originally intended

for data compression, the Burrows-Wheeler transform has been widely used in space-efficient text indexes, because it shares the combinatorial structure of the suffix tree and the suffix array.

Let LF be a function such that $\mathsf{SA}[\mathsf{LF}(i)] = \mathsf{SA}[i] - 1$ (with $\mathsf{SA}[\mathsf{LF}(i)] = n$, if $\mathsf{SA}[i] = 1$). We can compute LF as $\mathsf{LF}(i) = \mathsf{C}[\mathsf{BWT}[i]] + \mathsf{rank}_{\mathsf{BWT}[i]}(\mathsf{BWT}, i)$, where $\mathsf{C}[c]$ is the number of occurrences of characters with lexicographical values smaller than c in BWT. The inverse function of LF is Ψ, with $\Psi(i) = \mathsf{select}_c(\mathsf{BWT}, i - \mathsf{C}[c])$, where c is the largest character value with $\mathsf{C}[c] < i$. With functions LF and Ψ, we can move forward and backward in the text, while maintaining the lexicographic rank of the current suffix.

Compressed suffix arrays (CSA) [9, 13] are text indexes supporting similar functionality as the suffix array. This includes the following queries: i) $\mathsf{find}(P) = [sp, ep]$ finds the lexicographic range of suffixes starting with *pattern* $P[1, \ell]$; ii) $\mathsf{locate}(sp, ep) = \mathsf{SA}[sp, ep]$ locates these suffixes in the text; and iii) $\mathsf{extract}(i, j) = T[i, j]$ extracts substrings of the text. In practice, the find performance of compressed suffix arrays can be competitive with suffix arrays, while locate queries are orders of magnitude slower [8]. Typical index sizes are less than the size of the uncompressed text.

The *FM-index* (FMI) [9] is a common type of compressed suffix arrays. A typical implementation stores the BWT in a wavelet tree. find queries are supported by *backward searching*. Let $[sp, ep]$ be the lexicographic range of suffixes starting with the suffix $P[i + 1, \ell]$ of the pattern. We can find the range matching the suffix $P[i, \ell]$ with a generalization of function LF as

$$\mathsf{LF}([sp, ep], P[i]) = [\mathsf{C}[P[i]] + \mathsf{rank}_{P[i]}(\mathsf{BWT}, sp - 1) + 1, \mathsf{C}[P[i]] + \mathsf{rank}_{P[i]}(\mathsf{BWT}, ep)].$$

We support locate queries by *sampling* some suffix array pointers. If we want to determine $\mathsf{SA}[i]$ that has not been sampled, we can compute it as $\mathsf{SA}[i] = \mathsf{SA}[j] + k$, where $\mathsf{SA}[j]$ is a sampled pointer found by iterating LF k times, starting from i. The samples can be chosen in *suffix order*, sampling $\mathsf{SA}[i]$ at regular intervals, or in *text order*, sampling $T[i]$ at regular intervals and marking the sampled SA positions in a bitvector. Suffix order sampling requires less space, often resulting in better time/space trade-offs, while text order sampling guarantees better worst-case performance. extract queries are supported by sampling some ISA pointers. To extract $T[i, j]$, we find the nearest samples pointer after $\mathsf{ISA}[j]$, and traverse backwards to $T[i]$ with function LF.

Compressed suffix trees (CST) [23] are compressed text indexes supporting the full functionality of a suffix tree (see Table 1). They combine a compressed suffix array, a compressed representation of the LCP array, and a compressed representation of suffix tree topology. For the LCP array, there are several common representations:

- LCP-byte [2] stores the LCP array as a byte array. If $\mathsf{LCP}[i] < 255$, the LCP value is stored in the byte array. Larger values are marked with a 255 in the byte array and stored separately. Because most LCP values are typically small, LCP-byte usually requires n to $1.5n$ bytes of space.
- We can store the LCP array by using variable-length codes. LCP-dac uses *directly addressable codes* [5] for the purpose, resulting in a structure that is typically somewhat smaller and somewhat slower than LCP-byte.
- The *permuted* LCP (PLCP) *array* [23] $\mathsf{PLCP}[1, n]$ is the LCP array stored in text order and used as $\mathsf{LCP}[i] = \mathsf{PLCP}[\mathsf{SA}[i]]$. Because $\mathsf{PLCP}[i + 1] \geq \mathsf{PLCP}[i] - 1$, the array can be stored as a bitvector of length $2n$ in $2n + o(n)$ bits. If the text is repetitive, run-length encoding can be used to compress the bitvector to even less space [10]. Because accessing PLCP uses locate, it is much slower than the other common encodings.

Tree topology representations are the main differences between the various CST proposals. While various compressed suffix arrays and LCP arrays are interchangeable, tree topology de-

Table 1. Typical compressed suffix tree operations.

Operation	Description
$\mathsf{Root}()$	The root of the tree.
$\mathsf{Leaf}(v)$	Tells whether node v a leaf.
$\mathsf{Ancestor}(v, w)$	Tells whether node v is an ancestor of node w.
$\mathsf{Count}(v)$	Number of leaves in the subtree with v as the root.
$\mathsf{Locate}(v)$	Pointer to the suffix corresponding to leaf v.
$\mathsf{Parent}(v)$	The parent of node v.
$\mathsf{FChild}(v)$	The first child of node v in alphabetic order.
$\mathsf{NSibling}(v)$	The next sibling of node v in alphabetic order.
$\mathsf{LCA}(v, w)$	The lowest common ancestor of nodes v and w.
$\mathsf{SDepth}(v)$	String depth: Length $\ell = \lvert\pi(v)\rvert$ of the label from the root to node v.
$\mathsf{TDepth}(v)$	Tree depth: The depth of node v in the suffix tree.
$\mathsf{LAQ}_S(v, d)$	The highest ancestor of node v with string depth at least d.
$\mathsf{LAQ}_T(v, d)$	The ancestor of node v with tree depth d.
$\mathsf{SLink}(v)$	Suffix link: Node w such that $\pi(v) = c\pi(w)$ for a character $c \in \Sigma$.
$\mathsf{SLink}^k(v)$	Suffix link iterated k times.
$\mathsf{Child}(v, c)$	The child of node v with edge label starting with character c.
$\mathsf{Letter}(v, i)$	The character $\pi(v)[i]$.

termines how various suffix tree operations are implemented. There are three main families of compressed suffix trees:

- *Sadakane's compressed suffix tree* (CST-Sada) [23] uses a *balanced parentheses* representation for the tree. Each node is encoded as an opening parenthesis, followed by the encodings of its children and finally a closing parenthesis. This can be encoded as a bitvector of length $2n'$ for a tree with n' nodes, requiring up to $4n + \mathsf{o}(n)$ bits. CST-Sada tends to be larger and faster than the other compressed suffix trees [11, 1].
- The *fully compressed suffix tree* (FCST) of Russo et al. [22, 18] aims to use as little space as possible. It does not require an LCP array at all, and stores a balanced parentheses representation for Navarro2014a sampled subset of suffix tree nodes in $\mathsf{o}(n)$ bits. Unsampled nodes are retrieved by following suffix links. FCST is smaller and much slower than the other compressed suffix trees [22, 1].
- Fischer et al. [10, 19, 11, 1] proposed an intermediate representation, CST-NPR, based on LCP intervals. Tree navigation is handled by searching for the values defining the LCP intervals. *Range minimum queries* $\mathsf{rmq}(sp, ep)$ find the leftmost minimal value in $\mathsf{LCP}[sp, ep]$, while *next/previous smaller value* queries $\mathsf{nsv}(i)$ and $\mathsf{psv}(i)$ find the next/previous LCP value smaller than $\mathsf{LCP}[i]$.

For typical texts and component choices, the size of compressed suffix trees ranges from the $1.5n$ to $3n$ bytes of CST-Sada to the $0.5n$ to n bytes of FCST [11, 1]. There are also some CST variants for repetitive texts, such as versioned document collections and collections of individual genomes. Abeliuk et al. [1] developed a variant of CST-NPR that can be smaller than n bits, while achieving similar performance as the FCST. Navarro and Ordóñez [17] used grammar-based compression for the tree representation of CST-Sada, resulting in a compressed suffix tree that requires slightly more space than the CST-NPR of Abeliuk et al., while being closer to the non-repetitive CST-Sada and CST-NPR in performance.

2.3 Relative Lempel-Ziv

Relative Lempel-Ziv (RLZ) parsing [15] compresses *target* sequence S relative to *reference* sequence R. The target sequence is represented as a concatenation of z *phrases* $w_i = (p_i, \ell_i, c_i)$,

where p_i is the starting position of the phrase in the reference, ℓ_i is the length of the copied substring, and c_i is the *mismatching* character. If phrase w_i starts from position p' in the target, then $S[p', p' + \ell_i - 1] = R[p_i, p_i + \ell_i - 1]$ and $S[p' + \ell_i] = c_i$.

The shortest RLZ parsing of the target sequence can be found in linear time [**?**]. The algorithm builds a CSA for the reverse of the reference sequence, and then parses the target sequence greedily by using backward searching. If the edit distance between the reference and the target is s, we need at most s phrases to represent the target sequence. On the other hand, because the relative order of the phrases can be different in sequences R and S, the edit distance can be much higher than the number of phrases in the shortest RLZ parsing.

In a straightforward implementation, the *phrase pointers* p_i and the mismatching characters c_i can be stored in arrays W_p and W_c. These arrays take $z \log |S|$ bits and $z \log \sigma$ bits, respectively. To support random access in the target sequence, we can encode phrase lengths as bitvector W_ℓ of length $|S|$ [15]. We set $W_\ell[j] = 1$, if $S[j]$ is the first character of a phrase. The bitvector requires $z \log \frac{n}{z} + \mathsf{O}(z)$ bits, if we use the sdarray representation [20]. To extract $S[j]$, we first determine the phrase w_i, with $i = \mathsf{rank}_1(W_\ell, j)$. If $W_\ell[j+1] = 1$, we return the mismatching character $W_c[i]$. Otherwise we determine the phrase offset with a select query, and return character $R[W_p[i] + j - \mathsf{select}_1(W_\ell, i)]$.

The select query can be avoided by using *relative pointers* instead of absolute pointers [7]. By setting $W_p[i] = p_i - \mathsf{select}_1(W_\ell, i)$, the general case simplifies to $S[j] = R[W_p[i]+j]$. If most of the differences between the reference and the target sequence are single-character *substitutions*, p_{i+1} will often be $p_i + \ell_i + 1$. This corresponds to $A_p[i+1] = W_p[i]$ with relative pointers, making *run-length encoding* the pointer array worthwhile.

3 Relative FM-index

The *relative FM-index* (RFM) [3] is compressed suffix array of a sequence relative to the CSA of another sequence. We write $\mathsf{RFM}(S \mid R)$ to denote the relative FM-index of target sequence S relative to reference sequence R. The index is based on approximating the *longest common subsequence* (LCS) of $\mathsf{BWT}(R)$ and $\mathsf{BWT}(S)$, and storing several structures based on the common subsequence. Given a representation of $\mathsf{BWT}(R)$ supporting rank and select, we can use the relative index $\mathsf{RFM}(S \mid R)$ to simulate rank and select on $\mathsf{BWT}(S)$.

3.1 Basic index

Assume that we have found a long common subsequence of sequences X and Y. We call positions $X[i]$ and $Y[j]$ *lcs-positions*, if they are in the common subsequence. If we mark the lcs-positions with 1-bits in bitvectors B_X and B_Y, we can map between the corresponding lcs-positions in the two sequences with rank and select operations. If $X[i]$ is an lcs-position, the corresponding position in sequence Y is $Y[\mathsf{select}_1(B_Y, \mathsf{rank}_1(B_X, i))]$. We denote this pair of *lcs-bitvectors* $\mathsf{LCS}(X, Y)$.

In its most basic form, relative FM-index $\mathsf{RFM}(S \mid R)$ only supports find queries by simulating rank queries on $\mathsf{BWT}(S)$. It does this by storing $\mathsf{LCS}(\mathsf{BWT}(S), \mathsf{BWT}(R))$ and the *complementary subsequences* $\mathsf{CS}(S)$ and $\mathsf{CS}(R)$ that consist of the characters of $\mathsf{BWT}(S)$ and $\mathsf{BWT}(R)$ not in the common subsequence. The lcs-bitvectors are compressed using *entropy-based compression* [21], while the complementary subsequences are stored in similar structures as $\mathsf{BWT}(R)$.

To compute $\mathsf{rank}_c(\mathsf{BWT}(S), i)$, we start by determining the number of lcs-positions in $\mathsf{BWT}(S)$ up to position $S[i]$ as $k = \mathsf{rank}_1(B_{\mathsf{BWT}(S)}, i)$. Then we find the lcs-position k in $\mathsf{BWT}(R)$ as $j = \mathsf{select}_1(B_{\mathsf{BWT}(R)}, k)$. With these positions, we can compute

$$\mathsf{rank}_c(\mathsf{BWT}(S), i) = \mathsf{rank}_c(\mathsf{BWT}(R), j) - \mathsf{rank}_c(\mathsf{CS}(R), j - k) + \mathsf{rank}_c(\mathsf{CS}(S), i - k).$$

3.2 Relative select

We can implement the entire functionality of a compressed suffix array with rank queries on the BWT. However, if we use the CSA in a compressed suffix tree, we also need select queries to support *forward searching* with Ψ and Child queries. We can always implement select queries by binary searching with rank queries, but the result will be much slower than rank queries.

The faster alternative to support select queries in the relative FM-index is to build a *relative select* structure [4]. Let $\mathsf{F} = \mathsf{F}(X)$ be a sequence consisting of the characters of sequence X in sorted order. Alternatively, F is a sequence such that $\mathsf{F}[i] = \mathsf{BWT}[\Psi(i)]$. The relative select structure consists of bitvectors $\mathsf{LCS}(\mathsf{F}(S), \mathsf{F}(R))$, where $B_{\mathsf{F}(S)}[i] = B_{\mathsf{BWT}(S)}[\Psi(i)]$ and $B_{\mathsf{F}(R)}[i] = B_{\mathsf{BWT}(R)}[\Psi(i)]$, as well as the C array $\mathsf{C}(\mathsf{LCS})$ for the common subsequence.

To compute $\mathsf{select}_c(\mathsf{BWT}(S), i)$, we start by determining how many of the first i occurrences of character c are lcs-positions as $k = \mathsf{rank}_1(B_{\mathsf{F}(S)}, \mathsf{C}(\mathsf{BWT}(S))[c] + i) - \mathsf{C}(\mathsf{LCS})[c]$. Then we check from bit $B_{\mathsf{F}(S)}[\mathsf{C}(\mathsf{BWT}(S))[c] + i]$ whether the occurrence we are looking for is an lcs-position or not. If the occurrence is an lcs-position, we find it in $\mathsf{BWT}(R)$ as $j = \mathsf{select}_c(\mathsf{BWT}(R), \mathsf{select}_1(B_{\mathsf{F}(R)}, \mathsf{C}(\mathsf{LCS})[c] + k))$, and then map j to $\mathsf{select}_c(\mathsf{BWT}(S), i)$ by using $\mathsf{LCS}(\mathsf{BWT}(S), \mathsf{BWT}(R))$. Otherwise we find the occurrence in $\mathsf{CS}(S)$ as $j = \mathsf{select}_c(\mathsf{CS}(S), i-k)$, and then return $\mathsf{select}_c(\mathsf{BWT}(S), i) = \mathsf{select}_0(B_{\mathsf{BWT}(S)}, j)$.

3.3 Full functionality

- BWT-invariant subsequence, locate/extract
- finding a BWT-invariant subsequence

4 Relative compressed suffix tree

- CST operations
- RLZ compression of the DLCP array
- supporting RMQ/PSV/NSV (Abeliuk2013)

5 Experiments

- environment
- datasets
- construction time, space (inexact)
- component sizes; comparison to the old RFM
- basic queries: LF, Psi, RMQ/PSV/NSV
- locate() performance
- RCST size vs. mutation rate
- RCST vs. CST for repetitive collections
- CST traversal
- maximal matches

6 Conclusions

- conclusions
- other ideas: RLZ bitvectors, RLZ pointer compression

References

1. Andrés Abeliuk, Rodrigo Cánovas, and Gonzalo Navarro. Practical compressed suffix trees. *Algorithms*, 6(2):319–351, 2013.
2. Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.
3. Djamal Belazzougui, Travis Gagie, Simon Gog, Giovanni Manzini, and Jouni Sirén. Relative FM-indexes. In *Proceedings of the 21st Symposium on String Processing and Information Retrieval (SPIRE 2014)*, volume 8799 of *LNCS*, pages 52–64. Springer, 2014.
4. Christina Boucher, Alexander Bowe, Travis Gagie, Giovanni Manzini, and Jouni Sirén. Relative select. Unpublished manuscript, 2015.
5. Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. Directly adressable variable-length codes. In *Proceedings of the 16th Symposium on String Processing and Information Retrieval (SPIRE 2009)*, volume 5721 of *LNCS*, pages 122–130. Springer, 2009.
6. Michael Burrows and David J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
7. Héctor Ferrada, Travis Gagie, Simon Gog, and Simon J. Puglisi. Relative Lempel-Ziv with constant-time random access. In *Proceedings of the 21st Symposium on String Processing and Information Retrieval (SPIRE 2014)*, volume 8799 of *LNCS*, pages 13–17. Springer, 2014.
8. Paolo Ferragina, Rodrigo González, Gonzalo Navarro, and Rossano Venturini. Compressed text indexes: From theory to practice. *Journal of Experimental Algorithmics*, 13:1.12, 2009.
9. Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.
10. Johannes Fischer, Veli Mäkinen, and Gonzalo Navarro. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science*, 410(51):5354–5364, 2009.
11. Simon Gog. *Compressed Suffix Trees: Design, Construction, and Applications*. PhD thesis, Ulm University, 2011.
12. Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms (SODA 2003)*, pages 841–850. SIAM, 2003.
13. Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.
14. Stefan Kurtz. Reducing the space requirement of suffix trees. *Software: Practice and Experience*, 29(13):1149–1171, 1999.
15. Shanika Kuruppu, Simon J. Puglisi, and Justin Zobel. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In *Proceedings of the 17th Symposium on String Processing and Information Retrieval (SPIRE 2010)*, volume 6393 of *LNCS*, pages 201–206. Springer, 2010.
16. Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
17. Gonzalo Navarro and Alberto Ordóñez. Faster compressed suffix trees for repetitive text collections. In *Proceedings of the 13th International Symposium on Experimental and Efficient Algorithms (SEA 2014)*, volume 8504 of *LNCS*, pages 424–435. Springer, 2014.
18. Gonzalo Navarro and Luis M.S. Russo. Fast fully-compressed suffix trees. In *Proceedings of the 2014 IEEE Data Compression Conference (DCC 2014)*, pages 283–291. IEEE, 2014.
19. Enno Ohlebusch, Johannes Fischer, and Simon Gog. CST++. In *Proceedings of the 17th Symposium on String Processing and Information Retrieval (SPIRE 2010)*, volume 6393 of *LNCS*, pages 322–333. Springer, 2010.
20. Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX 2007)*, pages 60–70. SIAM, 2007.
21. Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):43, 2007.
22. Luis M.S. Russo, Gonzalo Navarro, and Arlindo L. Oliveira. Fully compressed suffix trees. *ACM Transactions on Algorithms*, 7(4):4, 2011.
23. Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.
24. Peter Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory (FOCS 1973)*, pages 1–11. IEEE, 1973.