Faster Repetition-Aware Compressed Suffix Trees based on Block Trees*

Manuel Cáceres[1] and Gonzalo Navarro[1]

CeBiB — Center for Biotechnology and Bioengineering, Department of Computer Science, University of Chile, Chile. {mcaceres, gnavarro}@dcc.uchile.cl

Abstract. Suffix trees are a fundamental data structure in stringology, but their space usage, though linear, is an important problem in applications. We design and implement a new compressed suffix tree targeted to highly repetitive texts, such as large genomic collections of the same species. Our suffix tree builds on Block Trees, a recent Lempel-Ziv-bounded data structure that captures the repetitiveness of its input. We use Block Trees to compress the topology of the suffix tree, and augment the Block Tree nodes with data that speeds up suffix tree navigation. Our compressed suffix tree is slightly larger than previous repetition-aware suffix trees based on grammars, but outperforms them in time, often by orders of magnitude. The component that represents the tree topology achieves a speed comparable to that of general-purpose compressed trees, while using 2–10 times less space, and might be of independent interest.

1 Introduction

Suffix trees [37, 22, 36] are one of the most appreciated data structures in Stringology [3] and in application areas like Bioinformatics [13], enabling efficient solutions to complex problems such as (approximate) pattern matching, pattern discovery, finding repeated substrings, computing matching statistics, computing maximal matches, and many others. In other collections, like natural language and software repositories, suffix trees are useful for plagiarism detection [23], authorship attribution [38], document retrieval [14], and others.

While their linear space complexity is regarded as acceptable in classical terms, their actual space usage brings serious problems in application areas. From an Information Theory standpoint, on a text of length n over alphabet $[1, \sigma]$, classical suffix tree representations use $\Theta(n \lg n)$ bits, whereas the information contained in the text is, in the worst case, just $n \lg \sigma$ bits. From a practical point of view, even carefully engineered implementations [17] require at least 10 bytes per symbol, which forces many applications to run the suffix tree on (orders of magnitude slower) secondary memory.

Consider for example Bioinformatics, where various complex analyses require the use of sophisticated data structures, suffix trees being among the most important ones. DNA sequences range over $\sigma = 4$ different nucleotides represented with

* Funded by Fondecyt Grant 1-170048 and by Basal Funds FB0001, Conicyt. Chile.

$\lg 4 = 2$ bits each, whereas the suffix tree uses at least 10 bytes = 80 bits per base, that is, 4000% of the text size. A human genome fits in approximately 715 MB, whereas its suffix tree requires about 30 GB. The space problem becomes daunting when we consider the DNA analysis of large groups of individuals; consider for example the 100,000-human-genomes project (`www.genomicsengland.co.uk`).

One solution to the problem is to build suffix trees on secondary memory [7, 9]. Most suffix tree algorithms, however, require traversing them across arbitrary access paths, which makes secondary memory solutions many orders of magnitude slower than in main memory. Another approach replaces the suffix trees with suffix arrays [21], which decreases space usage to 4 bytes (32 bits) per character but loses some functionality like the suffix links, which are essential to solve various complex problems. This functionality can be recovered [2] by raising the space to about 6 bytes (48 bits) per character.

A promising line of research is the construction of compact representations of suffix trees, named *Compressed Suffix Trees (CSTs)*, which simulate all the suffix tree functionality within space bounded not only by $O(n \lg \sigma)$ bits, but by the information content (or text entropy) of the sequence. An important theoretical achievement was a CST using $O(n)$ bits on top of the text entropy that supports all the operations within an $O(\text{polylog } n)$ time penalty factor [34]. A recent implementation [28] uses, on DNA, about 10 bits per base and supports the operations in a few microseconds. While even smaller CSTs have been proposed, reaching as little as 5 bits per base [32], their operation times raise to milliseconds, thus becoming nearly as slow as a secondary-memory deployment.

Still, further space reductions are desirable when facing large genome repositories. Fortunately many of the largest text collections are highly repetitive; for example DNA sequences of two humans differ by less than 0.5% [35]. This repetitiveness is not well captured by statistical based compression methods [16], on which most of the CSTs are based. Lempel-Ziv [19] and grammar [15] based compression techniques, among others, do better in this scenario [24], but only recently we have seen CSTs building on them, both in theory [11, 5] and in practice [1, 26]. The most successful CSTs in practice on repetitive collections are the grammar-compressed suffix trees (GCSTs), which on DNA use about 2 bits per base and support the operations in tens to hundreds of microseconds.

GCSTs use grammar compression on the parentheses sequence that represents the suffix tree topology [31], which inherits the repetitiveness of the text collection. While Lempel-Ziv compression is stronger, it does not support easy access to the sequence. In this paper we explore an alternative to grammar compression called Block Trees [6, 29], which offer similar approximation ratios to Lempel-Ziv compression, but promise faster access.

Our main contribution is the BT-CT, a Block-Tree-based representation of tree topologies, which enriches Block Trees to support the required navigation operations. Although we are unable to prove useful upper bounds on the operation times, the BT-CT performs very well in practice: while using 0.3–1.5 bits per node in our repetitive suffix trees, it implements the navigation operations in a few microseconds, becoming very close to the performance of plain 2.8-bit-per-

Operation	**Description**
root()	The root of the suffix tree
is-leaf(v)	True if v is a leaf node
first-child(v)	The first child of v in lexicographical order
tree-depth(v)	The number of edges from root() to v
next-sibling(v)	The next sibling of v in lexicographical order
previous-sibling(v)	The previous sibling of v in lexicographical order
parent(v)	The parent of v
is-ancestor(v,u)	True if v is ancestor of u
level-ancestor(v,d)	The ancestor of v at tree depth d
lca(v,u)	The lowest common ancestor between v and u
letter(v, i)	str(v)[i]
string-depth(v)	|str(v)|
suffix-link(v)	The node u s.t. str(u) = str(v)[2,string-depth(v)]
string-ancestor(v,d)	The highest ancestor u of v s.t. string-depth(u) $\geq d$
child(v,c)	The child u of v s.t. str(u)[string-depth(v)+1] = c

Table 1: List of typical operations implemented by suffix trees; str(v) represents the concatenation of the strings in the root-to-v path.

node representations that are blind to repetitiveness [27]. We use the BT-CT to represent suffix tree topologies in this paper, but it might also be useful in other scenarios, such as representing the topology of repetitive XML collections [4].

As said, our new suffix tree, BT-CST, uses the BT-CT to represent the suffix tree topology. Although larger than the GCST, it still requires about 3 bits per base in highly repetitive DNA collections. In exchange, it is faster than the GCST, often by an order of magnitude. This owes to the BT-CT directly, but also indirectly: Its faster navigation enables the binary search for the "child by letter" operation in suffix trees, which is by far the slowest one. While with the GCST a linear traversal of the children is advisable [26], a binary search pays off in the BT-CST, making it faster especially on large alphabets.

2 Preliminaries and Related Work

A text $T[1,n] = T[1] \dots T[n]$ is a sequence of symbols over an alphabet $\Sigma = [1,\sigma]$, terminated by a special symbol \$ that is lexicographically smaller than any symbol of Σ. A substring of T is denoted $T[i,j] = T[i] \dots T[j]$. A substring $T[i,j]$ is a prefix if $i = 1$ and a suffix if $j = n$.

The *suffix tree* [37, 22, 36] of a text T is a trie of its suffixes in which unary paths are collapsed into a single edge. The tree then has less than $2n$ nodes. The suffix tree supports a set of operations (see Table 1) that suffices to solve a large number of problems in Stringology [3] and Bioinformatics [13].

The *suffix array* [21] $A[1,n]$ of a text $T[1,n]$ is a permutation of $[1,n]$ such that $A[i]$ is the starting position of the ith suffix in increasing lexicographical order. The leaves descending from a suffix tree node span a range of suffixes in A.

The function $lcp(X, Y)$ is the length of the longest common prefix (lcp) of strings X and Y. The *LCP array* [21], $LCP[1, n]$, is defined as $LCP[1] = 0$ and $LCP[i] = lcp(T[A[i-1], n], T[A[i], n])$ for all $i > 1$, that is, it stores the lengths of the lcps between lexicographically consecutive suffixes of $T[1, n]$.

2.1 Succinct tree representations

A balanced parentheses (BP) representation (there are others [31]) of the topology of an ordinal tree $\mathcal{T}$ of t nodes is a binary sequence (or bitvector) $P[1, 2t]$ built as follows: we traverse $\mathcal{T}$ in preorder, writing an opening parenthesis (a bit 1) when we first arrive at a node, and a closing one (a bit 0) when we leave its subtree. For example, a leaf looks like "10". The following primitives can be defined on P:

- $access(i) = P[i]$
- $rank_{0|1}(i) = |\{1 \leq j \leq i; P[j] = 0|1\}|$
- $excess(i) = rank_1(i) - rank_0(i)$
- $select_{0|1}(i) = \min(\{j; rank_{0|1}(j) = i\} \cup \{\infty\})$
- $leaf\text{-}rank(i) = rank_{10}(i) = |\{1 \leq j \leq i-1; P[j] = 1 \wedge P[j+1] = 0\}|$
- $leaf\text{-}select(i) = select_{10}(i) = \min(\{j; leaf\text{-}rank(j+1) = i\} \cup \{\infty\})$
- $fwd\text{-}search(i, d) = \min(\{j > i; excess(j) = excess(i) + d)\} \cup \{\infty\})$
- $bwd\text{-}search(i, d) = \max(\{j < i; excess(j) = excess(i) + d)\} \cup \{-\infty\})$
- $min\text{-}excess(i, j) = \min(\{excess(k) - excess(i-1); i \leq k \leq j\} \cup \{\infty\})$

These primitives suffice to implement a large number of tree navigation operations, and can all be supported in constant time using $o(t)$ bits on top of P [27]. These include the operations needed by suffix trees. For example, interpreting nodes as the position of their opening parenthesis in P, it holds that $parent(v) = bwd\text{-}search(i, -2)+1$, $next\text{-}sibling(v) = fwd\text{-}search(v, -1)+1$ and the lowest common ancestor of two nodes $v \leq u$ is $lca(v, u) = parent(fwd\text{-}search(v - 1, min\text{-}excess(v, u)) + 1)$.

2.2 Compressed Suffix Arrays

A milestone in the area was the emergence of Compressed Suffix Arrays (CSAs) [25], which using space proportional to that of the compressed sequence managed to answer access queries to the original suffix array and its inverse (i.e., return any $A[i]$ and $A^{-1}[j]$), to the indexed sequence (i.e., return any $T[i..j]$), and access to a novel array, $\Psi[i] = A^{-1}[(A[i] \bmod n) + 1]$, which lets us move from a text suffix $T[j, n]$ to the next one, $T[j+1, n]$, yet indexing the suffixes by their lexicographic rank, $A^{-1}[j]$. This function plays a key role in the design of CSTs, as seen next.

2.3 Compressed Suffix Trees

Sadakane [34] designed the first CST, on top of a CSA, using $|\text{CSA}| + O(n)$ bits and solving all the suffix tree operations in time $O(\text{polylog}\ n)$. He makes up a CST from three components: a CSA, for which he uses his own proposal [33];

a BP representation of the suffix tree topology, using at most $4n + o(n)$ bits; and a compressed representation of LCP, which is a bitvector $H[1, 2n]$ encoding the array $PLCP[i] = LCP[A^{-1}[i]]$ (i.e., the LCP array in text order). A recent implementation [28] of this index requires about 10 bits per character and takes a few microseconds per operation.

Russo et al. [32] managed to use just $o(n)$ bits on top of the CSA, by storing only a sample of the suffix tree nodes. An implementation of this index [32] uses as little as 5 bits per character, but the operations take milliseconds, as slow as running in secondary storage.

Yet another approach [10] also obtains $o(n)$ on top of a CSA by getting rid of the tree topology and expressing the tree operations on the corresponding suffix array intervals. The operations now use primitives on the LCP array: find the previous/next smaller value (psv/nsv) and find minima in ranges (rmq). They also noted that bitvector H contains $2r$ runs, where r is the number of runs of consecutive increasing values in Ψ, and used this fact to run-length compress H. Abeliuk et al. [1] designed a practical version of this idea, obtaining about 8 bits per character and getting a time performance of hundreds of microseconds per operation, an interesting tradeoff between the other two options.

Engineered adaptations of these three ideas were implemented in the SDSL library [12], and are named `cst_sada`, `cst_fully`, and `cst_sct3`, respectively. We will use and adapt them in our experimental comparison.

2.4 Repetition-aware Compressed Suffix Trees

Abeliuk et. al [1] also presented the first CST for repetitive collections. They built on the third approach above [10], so they do not represent the tree topology. They use the RLCSA [20], a repetition-aware CSA with size proportional to r, which is very low on repetitive texts. They use grammar compression on the *differential* LCP array, $DLCP[i] = LCP[i] - LCP[i-1]$. The nodes of the parsing tree (obtained with Re-Pair [18]) are enriched with further data to support the operations psv/nsv and rmq. To speed up simple LCP accesses, the bitvector H is also stored, whose size is also proportional to r. Their index uses 1–2 bits per character on repetitive collections. It is rather slow, however, operating within (many) milliseconds.

Navarro and Ordóñez [26] include again the tree topology. Since text repetitiveness induces isomorphic subtrees in the suffix tree, they grammar-compressed the BP representation. The nonterminals are enriched to support the tree navigation operations enumerated in Section 2.1. Since they do not need psv/nsv/rmq operations on LCP, they just use the bitvector H, which has a few runs and thus is very small. Their index uses slightly more space, closer to 2 bits per character, but it is up to 3 orders of magnitude faster than that of Abeliuk et al. [1]: their structure operates in tens to hundreds of microseconds per operation, getting closer to the times of general-purpose CSTs.

Less related or theoretical work [8, 11, 5] is not discussed for lack of space.

3 Block Trees

A Block Tree [6] is a full r-ary tree that represents a (repetitive) sequence $P[1,p]$ in compressed space while offering access and other operations in logarithmic time. The nodes at depth d (the root being depth 0) represent blocks of P of length $b = |P|/r^d$, where we pad P to ensure these numbers are integers. Such a node v, representing some block $v.blk = P[i, i+b-1]$, can be of three types:

LeafBlock: If $b \leq mll$, where mll is a parameter, then v is a leaf of the Block Tree, and it stores the string $v.blk$ explicitly.

BackBlock: Otherwise, if $P[i-b, i+b-1]$ and $P[i, i+2b-1]$ are not their leftmost occurrences in P, then the block is replaced by its leftmost occurrence in P: node v stores a pointer $v.ptr = u$ to the node u such that the first occurrence of $v.blk$ starts inside $u.blk = P[j, j+b-1]$, more precisely it occurs in $P[j+o, j+o+b-1]$. This offset inside $u.blk$ is stored at $v.\mathit{off} = o$. Node v is not considered at deeper levels.

InternalBlock: Otherwise, the block is split into r equal parts, handled in the next level by the children of v. The node v then stores a pointer to its children.

The Block Tree can return any $P[i]$ in logarithmic time, by starting at position i in the root block. Recursively, the position i is translated in constant time into an offset inside a child node (for InternalBlocks), or inside a leftward node in the same level (for BackBlocks, at most once per level). At leaves, the symbol is stored explicitly.

If we augment the nodes of the Block Tree with rank information for the σ symbols of the alphabet, the Block Tree answers rank and select queries on P in logarithmic time as well. Specifically, for every $c \in [1, \sigma]$, we store in every node v the number $v.c$ of cs in $v.blk$. Further, every BackBlock node v pointing to u stores the number of cs in $u.blk[1, v.\mathit{off}-1]$.

Our new repetition-aware CST will represent the BP topology with a Block Tree. The basic structure supports operations $\mathit{access}(i)$, $\mathit{rank}_{0|1}(i)$, $\mathit{excess}(i)$ and $\mathit{select}_{0|1}(i)$. In the next section we show how to solve the remaining operations.

4 Our Repetition-Aware Compressed Suffix Tree

Following the scheme of Sadakane [34] we propose a three-component structure to implement a new CST tailored to highly repetitive inputs. We use the RLCSA [20] as our CSA. For the LCP, we use the compressed version of the bitvector H [10]. For the topology, we use BP and represent the sequence with a Block Tree, adding new fields to the Block Tree nodes to efficiently answer all the queries we need (Section 2.1). We call this representation Block Tree CST (BT-CST). Section 4.1 describes BT-CT, our extension to Block Trees, and Section 4.2 our improved operation child(v, a) for the BT-CST.

4.1 Block Tree Compressed Topology (BT-CT)

We describe our main data structure, *Block Tree Compressed Topology (BT-CT)*, which compresses a parentheses sequence and supports navigation on it.

Stored fields We augment the nodes of the Block Tree with the following fields:

- For every node v that represents the block $v.\mathit{blk} = P[i, i+b-1]$:
  - rank_1, the number of 1s in $v.\mathit{blk}$, i.e., $\mathit{rank}_1(i+b-1) - \mathit{rank}_1(i-1)$ in P.
  - *lrank* (leaf rank), the number of 10s (i.e., leaves in BP) that finish inside $v.\mathit{blk}$, i.e., $\mathit{leaf\text{-}rank}(i+b-1) - \mathit{leaf\text{-}rank}(i-1)$ in P.
  - *lbreaker* (leaf breaker), a bit telling whether the first symbol of $v.\mathit{blk}$ is a 0 and the preceding symbol in P is a 1, i.e., whether $P[i-1, i] = 10$.
  - *mexcess*, the minimum excess in $v.\mathit{blk}$, i.e., $\mathit{min\text{-}excess}(i, i+b-1)$ in P.
- For every BackBlock node v that represents $v.\mathit{blk} = P[i, i+b-1]$ and points to its first occurrence $O = P[j+o, j+o+b-1]$ inside $u.\mathit{blk} = P[j, j+b-1]$ with offset $v.\mathit{off} = o$:
  - $\mathit{fb\text{-}rank}_1$, the number of 1s in the prefix of O contained in $u.\mathit{blk}$ ($O \cap u.\mathit{blk}$, the 1st block spanned by O), i.e., $\mathit{rank}_1(j+b-1) - \mathit{rank}_1(j+o-1)$ in P.
  - *fb-lrank*, the number of 10s that finish in $O \cap u.\mathit{blk}$, i.e., $\mathit{leaf\text{-}rank}(j+b-1) - \mathit{leaf\text{-}rank}(j+o-1)$ in P.
  - *fb-lbreaker*, a bit telling whether the first symbol of O is a 0 and the preceding symbol is a 1, i.e., whether $P[j+o-1, j+o] = 10$.
  - *fb-mexcess*, the minimum excess reached in $O \cap u.\mathit{blk}$, i.e., $\mathit{min\text{-}excess}(j+o, j+b-1)$.
  - *m-fb*, a bit telling whether the minimum excess of $u.\mathit{blk}$ is reached in $O \cap u.\mathit{blk}$, i.e., whether $\mathit{min\text{-}excess}(i, i+b-1) = \mathit{min\text{-}excess}(j+o, j+b-1)$.

Fields computed on the fly In the description of the operations we will use other fields that are computed in constant time from those we already store:

- For every node v that represents $v.\mathit{blk} = P[i, i+b-1]$
  - rank_0, the number of 0s in $v.\mathit{blk}$, i.e., $b - v.\mathit{rank}_1$.
  - *excess*, the excess of 1s over 0s in $v.\mathit{blk}$, i.e., $v.\mathit{rank}_1 - v.\mathit{rank}_0 = 2 \cdot v.\mathit{rank}_1 - b$.
- For every BackBlock node v that represents $v.\mathit{blk} = P[i, i+b-1]$ and points to its first occurrence $O = P[j+o, j+o+b-1]$ inside $u.\mathit{blk} = P[j, j+b-1]$ with offset $v.\mathit{off} = o$:
  - $\mathit{fb\text{-}rank}_0$, the number of 0s in $O \cap v.\mathit{blk}$, i.e., $(b-o) - v.\mathit{fb\text{-}rank}_1$.
  - $\mathit{pfb\text{-}rank}_{0|1}$, the number of 0s|1s in the prefix of $u.\mathit{blk}$ that precedes O ($u.\mathit{blk} - O$), i.e., $u.\mathit{rank}_{0|1} - v.\mathit{fb\text{-}rank}_{0|1}$.
  - *fb-excess*, the excess in $O \cap u.\mathit{blk}$, i.e., $v.\mathit{fb\text{-}rank}_1 - v.\mathit{fb\text{-}rank}_0$.
  - *sb-excess*, the excess in $O - u.\mathit{blk}$ (2nd block spanned by O), i.e., $v.\mathit{excess} - v.\mathit{fb\text{-}excess}$.
  - *pfb-lrank*, the number of 10s that finish in $u.\mathit{blk} - O$, i.e., $u.\mathit{lrank} - v.\mathit{fb\text{-}lrank}$.

- *sb-mexcess*, the minimum excess in $O - u.blk$, i.e., *min-excess*$(j+b, j+b+o-1)$ in P. We store either v.*fb-mexcess* or v.*sb-mexcess*, the one that differs from $v.mexcess$. To deduce the non-stored field we use *mexcess*, *fb-excess* and *m-fb*.

Complex operations Apart from the basic operations solved in the original Block Tree we need, as described in Section 2.1, more sophisticated ones to support navigation in the parentheses sequence.

leaf-rank(i) ***and leaf-select***(i)***.*** The implementations of these operations are analogous to those for $rank_c(i)$ and $select_c(i)$ respectively, in the base Block Tree. The only two differences are that in LeafBlocks we consider the *lbreaker* field to check whether the block starts with a leaf, and in BackBlocks we consider fields *lbreaker* and *fb-lbreaker* to check whether we have to add or remove one leaf when moving to a leftward node. Like $rank_c(i)$ and $select_c(i)$, our operations work $O(1)$ per level, and then have their same time complexity, given in Section 3.

fwd-search(i, d) ***and bwd-search***(i, d)***.*** We only show how to solve *fwd-search*(i, d) with $d < 0$; the other cases are similar (some combinations not needed for our CST require further fields). Thus we aim to find the smallest position $j > i$ where the excess of $P[i+1..j]$ is d.

We describe our solution as a recursive procedure *fwd-search*(i, j) with two global variables: d from the input, and e. Variables i and j are the limits of the search for the currently processed node, and e is the accumulated excess of the part of the range that has already been processed. The procedure is initially called at the Block Tree root with *fwd-search*(i, n) and with $e = 0$. If at some point e reaches d, we have found the answer to the search. The general idea is to traverse the range of the current node v left to right, using the fields $v.mexcess$, v.*fb-mexcess* and v.*sb-mexcess* to speed up the procedure:

- If the search range spans the entire block $v.blk$ (i.e., $j - i = b$) and the answer is not reached inside v (i.e., $e + v.mexcess > d$), then we increase e by $v.excess$ and return ∞.
- If v is a LeafBlock we scan $v.blk$ bitwise, increasing/decreasing e for each 1/0. If e reaches d at some index k, we return k; otherwise we return ∞.
- If v is an InternalBlock, we identify the k-th child of v, which contains position $i+1$, and the m-th, which contains position j (it could be that $k = m$). We then call *fwd-search* recursively on the k-th to the m-th children, intersecting the query range with the extent of each child (the search range will completely cover the children after the k-th and before the m-th). As soon as any of these calls returns a non-∞ value, we adjust (i.e., shift) and return it. If all of them return ∞, we also return ∞.
- If v is a BackBlock we must translate the query to the original block O, which starts at offset $v.\mathit{off}$ in $u.blk$, where $u = v.ptr$. We first check whether the query covers the prefix of $v.blk$ contained in $u.blk$, $O \cap u.blk$ (i.e., if $i = 0$ and $j \geq b - v.\mathit{off}$). If so, we check whether we can skip $O \cap u.blk$, namely if

$e + v.\mathit{fb\text{-}mexcess} > d$. If we can skip it, we just update e to $e + v.\mathit{fb\text{-}excess}$, otherwise we call *fwd-search* recursively on the intersection of $u.\mathit{blk}$ and the translated query range. If the answer is not ∞, we adjust and return it. Otherwise, we turn our attention to the node u' next to u. Again, we check whether the query covers the suffix of $v.\mathit{blk}$ contained in $u'.\mathit{blk}$, $O - u.\mathit{blk}$ (i.e., $j = b$ and $i \leq b - v.\mathit{off}$). If so, we check whether we can skip $O - u.\mathit{blk}$, namely if $e + v.\mathit{sb\text{-}mexcess} > d$. If we can skip it, we just update e to $e + v.\mathit{sb\text{-}excess}$, otherwise we call *fwd-search* recursively on the intersection of $u'.\mathit{blk}$ and the translated query range. If the answer is not ∞, we adjust and return it. Otherwise, we return ∞.

min-excess(i, j)**.** We will also start at the root with the global variable e set to zero. A local variable m will keep track of the minimum excess seen in the current node, and will be initialized at $m = 1$ in each recursive call. The idea is the same as for *fwd-search*: traverse the node left to right and use the fields $v.\mathit{mexcess}$, $v.\mathit{fb\text{-}mexcess}$ and $v.\mathit{sb\text{-}mexcess}$ to speed up the traversal.

- If the query covers the entire block $v.\mathit{blk}$ (i.e., $j - i + 1 = b$), we increase e by $v.\mathit{excess}$ and return $v.\mathit{mexcess}$.
- If v is a LeafBlock we record the initial excess in $e' = e$ and scan $v.\mathit{blk}$ bitwise, updating e for each bit read as in operation *fwd-search*. Every time we have $e - e' < m$, we update $m = e - e'$. At the end of the scan we return m.
- If v is an InternalBlock, we identify the k-th child of v, which contains position i, and the m-th, which contains position j (it could be that $k = m$). We then call *min-excess* recursively on the k-th to the m-th children, intersecting the query range with the extent of each child (the search range will completely cover the children after the k-th and before the m-th, so these will take constant time). We return the minimum between all their answers (composed with their correspondent prefix excesses).
- If v is a BackBlock we translate the query to the original block O, which starts at offset $v.\mathit{off}$ in $u.\mathit{blk}$, where $u = v.\mathit{ptr}$. We first check whether the query covers the prefix of $v.\mathit{blk}$ contained in $u.\mathit{blk}$, $O \cap u.\mathit{blk}$ (i.e., if $i = 1$ and $j \geq b - v.\mathit{off} - 1$). If so, we simply set $m = v.\mathit{fb\text{-}mexcess}$ and update e to $e + v.\mathit{fb\text{-}excess}$. Otherwise we call *min-excess* recursively on the intersection of $u.\mathit{blk}$ and the translated query range, and record its answer in m. We now consider the block u' next to u and again check whether the query covers the suffix of $v.\mathit{blk}$ contained in $u'.\mathit{blk}$, $O - u.\mathit{blk}$ (i.e., if $j = b$ and $i \leq b - v.\mathit{off} + 1$). If so, we just set $m = \min(m, v.\mathit{fb\text{-}excess} + v.\mathit{sb\text{-}mexcess})$ and update e to $e + v.\mathit{sb\text{-}excess}$. Otherwise, we call *min-excess* on the intersection of $u'.\mathit{blk}$ and the translated query range, record its answer in m', and set $m = \min(m, v.\mathit{fb\text{-}excess} + m')$. Finally, we return m.

Note that, although we look for various opportunities to use the precomputed data to skip parts of the query range, the operations *fwd-search*, *bwd-search*, and *min-excess* are not guaranteed to work proportionally to the height of the Block Tree. The instances we built that break this time complexity, however, are unlikely to occur. Our experiments will show that the algorithms perform well in practice.

4.2 Operation child

The fast operations enabled by our BT-CT structure give space for an improved algorithm to solve operation child(v, a). Most previous CSTs first compute $d =$ string-depth(v) and then linearly traverse the children of v from $u =$ first-child(v) with operation next-sibling, checking for each child u whether letter$(u, d+1) = a$, and stopping as soon as we find or exceed a. Since computing letter is significantly more expensive than our next-sibling, we consider the variant of first identifying all the children u of v, and then binary searching them for a, using letter. We then perform $O(\sigma)$ next-sibling operations, but only $O(\lg \sigma)$ letter operations.

5 Experiments and Results

We measured the time/space performance of our new BT-CST and compared it with the state of the art. Our code and testbed is available at
https://github.com/elarielcl/BT-CST.

5.1 Experimental setup

Compared CSTs. We compare the following CST implementations.

BT-CST. Our new Compressed Suffix Tree with the described components. For the BT-CT component we vary $r \in \{2, 4, 8\}$ and $mll \in \{4, 8, 16, 32, 64, 128, 256\}$.

GCST. The Grammar-based Compressed Suffix Tree [26]. We vary parameters *rule-sampling* and *C-sampling* as they suggest.

CST_SADA ,CST_SCT3, CST_FULLY. Adaptation and improvements from the SDSL library[1] on the indexes of Sadakane [34], Fischer et al. [10] and Russo et al. [32], respectively. CST_SADA maximizes speed using Sadakane's CSA [33] and a non-compressed version of bitvector H. CST_SCT3 uses instead a Huffman-shaped wavelet tree of the BWT as the suffix array, and a compressed representation [30] for bitvector H and those of the wavelet tree. This bitvector representation exploits the runs and makes the space sensitive to repetitiveness, but it is slower. CST_FULLY uses the same BWT representation. For all these suffix arrays we set *sa-sampling* $= 32$ and *isa-sampling* $= 64$.

CST_SADA_RLCSA, CST_SCT3_RLCSA. Same as the preceding implementations but (further) adapted to repetitive collections: We replace the suffix array by the RLCSA [20] and use a run-length-compressed representation of bitvector H [10].

[1] Succinct data structures library (SDSL), https://github.com/simongog/sdsl-lite

For the CSTs using the RLCSA, we fix their parameters to 32 for the sampling of Ψ and 128 for the text sampling. We only show the Pareto-optimal results of each structure. Note that we do not include the CST of Abeliuk et al. [1] in the comparison because it was already outperformed by several orders of magnitude by GCST [26].

Text collection and queries. Our input sequences come from the Repetitive Corpus of *Pizza&Chili* (http://pizzachili.dcc.uchile.cl/repcorpus). We selected `einstein`, containing all the versions (up to January 12, 2010) of the German Wikipedia Article of *Albert Einstein* (89MB, compressible by p7zip to 0.11%); `influenza`, a collection of 78,041 H. influenzae genomes (148MB, compressible by p7zip to 1.69%); and `kernel`, a set of 36 versions of the Linux Kernel (247MB, compressible by p7zip to 2.56%).

Data points are the average of 100,000 random queries, similar to the scheme used in previous work on Compressed Suffix Trees [1, 26] to choose the nodes on which the operations are called: For *next-sibling* and *parent* we collect the nodes in leaf-to-root paths starting from random leaves. For *lca* we choose random leaf pairs. For *suffix-link* we collect the nodes on traversals starting from random leaves, and taking suffix-links until reaching the root. For *child* we choose random leaves and collect the nodes in the traversals to the root, discarding the nodes with less than 3 children, and we choose the initial letter of a random child of the node.

Computer. The experiments ran on an isolated Intel(R) Xeon(R) CPU E5-2407 @ 2.40GHz with 256GB of RAM and 10MB of L3 cache. The operating system is GNU/Linux, Debian 2, with kernel 4.9.0-8-amd64. The implementations use a single thread and all of them are coded in C++. The compiler is gcc version 4.6.3, with -O9 optimization flag set (except CST_SADA, CST_SCT3 and CST_FULLY, which use their own set of optimization flags).

Operations. We implemented all the suffix tree operations of Table 1. From those, for lack of space, we present the performance comparison with other CSTs on five important operations: next-sibling, parent, child, suffix-link, and lca. To test our suffix tree in more complex scenarios we implemented the suffix-tree-based algorithm to solve the "maximal substrings" problem [26] on all of the above implementations except for CST_FULLY (because of its poor time performance). We use their same setup [26], that is, `influenza` from *Pizza&Chili* as our larger sequence and a substring of size m (m = 3000 and m = 2MB) of another `influenza` sequence taken from https://ftp.ncbi.nih.gov/genomes/INFLUENZA. BT-CST uses $r = 2$ and $mll = 128$ and GCST uses $\textit{rule-sampling} = 1$ and $\textit{C-sampling} = 2^{10}$. The tradeoffs refer to $\textit{sa-sampling} \in \{64, 128, 256\}$ for the RLCSAs.

5.2 Results and discussion

Figures 1 to 3 show the space and time for all the indexes and all the operations. The smallest structure is GCST, which takes as little as 0.5–2 bits per symbol (bps). The next smallest indexes are BT-CST, using 1–3 bps, and CST_FULLY, using 2.0–2.5 bps. The compressed indexes not designed for repetitive collections use 4–7 bps if combined with a RLCSA, and 6–10.5 bps in their original versions (though we also adapted the bitvectors of CST_SCT3).

From the BT-CST space, component H takes just 2%–9%, the RLCSA takes 23%–47%, and the rest is the BT-CT (using a sweetpoint configuration). This component takes 0.30 bits per node (bpn) on `einstein`, 1.06 bpn on `influenza`, and 1.50 bpn on `kernel`. The grammar-compressed topology of GCST takes, respectively, 0.05, 0.81, and 0.39 bpn.

In operations next-sibling and parent, which rely most heavily on the suffix tree topology, our BT-CT component building on Block Trees makes BT-CST excel in time: The operations take nearly one microsecond (μsec), at least 10 times less than the grammar-based topology representation of GCST. CST_FULLY is three orders of magnitude slower on this operation, taking over a millisecond (msec). Interestingly, the larger representations, including those where the tree topology is represented using 2.79 bits per node (CST_SADA[_RLCSA]), are only marginally faster than BT-CST, whereas the indexes CST_SCT3[_RLCSA] are a bit slower than CST_SADA[_RLCSA] because they do not store an explicit tree topology. Note that these operations, in BT-CT, make use of the operations *fwd-search* and *bwd-search*, thereby showing that they are fast although we cannot prove worst-case upper bounds on their time.

Operation lca, which on BT-CST involves essentially the primitive *min-excess*, is costlier, taking around 10 μsec in almost all the indexes, including ours. This includes again those where the tree topology is represented using 2.79 bits per node (CST_SADA[_RLCSA]). Thus, although we cannot prove upper bounds on the time of *min-excess*, it is in practice as fast as on perfectly balanced structures, where it can be proved to be logarithmic-time. The variants CST_SCT3[_RLCSA] also require an operation very similar to *min-excess*, so they perform almost like CST_SADA[_RLCSA]. For this operation, CST_FULLY is equally fast, owing to the fact that operation lca is a basic primitive in this representation. Only GCST is several times slower than BT-CST, taking several tens of μsec.

Operation suffix-link involves *min-excess* and several other operations on the topology, but also the operation Ψ on the corresponding CSA. Since the latter is relatively fast, BT-CST also takes nearly 10 μsec, whereas the additional operations on the topology drive GCST over 100 μsec, and CST_FULLY over the msec. This time the topology representations that are blind to repetitivess are several times faster than BT-CST, taking a few μsec, possibly because they take more advantage of the smaller ranges for *min-excess* involved when choosing random nodes (most nodes have small ranges). The CST_SCT3[_RLCSA] variants also solve this operation with a fast and simple formula.

Finally, operation child is the most expensive one, requiring one application of string-depth and several of next-sibling and letter, thereby heavily relying on the CSA. BT-CST-bin and CST_SCT3[_RLCSA] binary search the children; the others scan them linearly. The indexes using a CSA that adapts to repetitiveness require nearly 1 msec on large alphabets, whereas those using a larger and faster CSA are

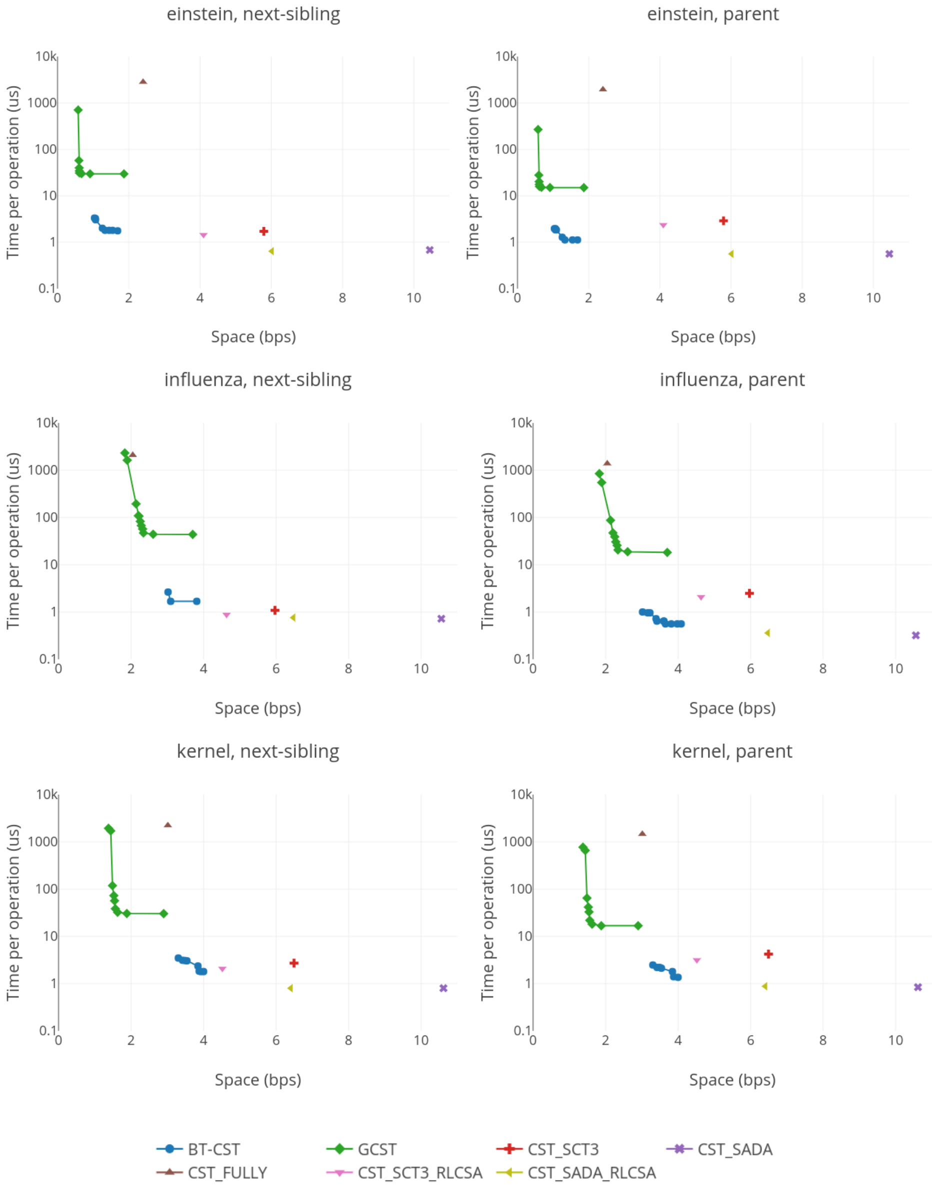

Fig. 1: Performance of CSTs for operations next-sibling and parent. The y-axis is in log-scale.

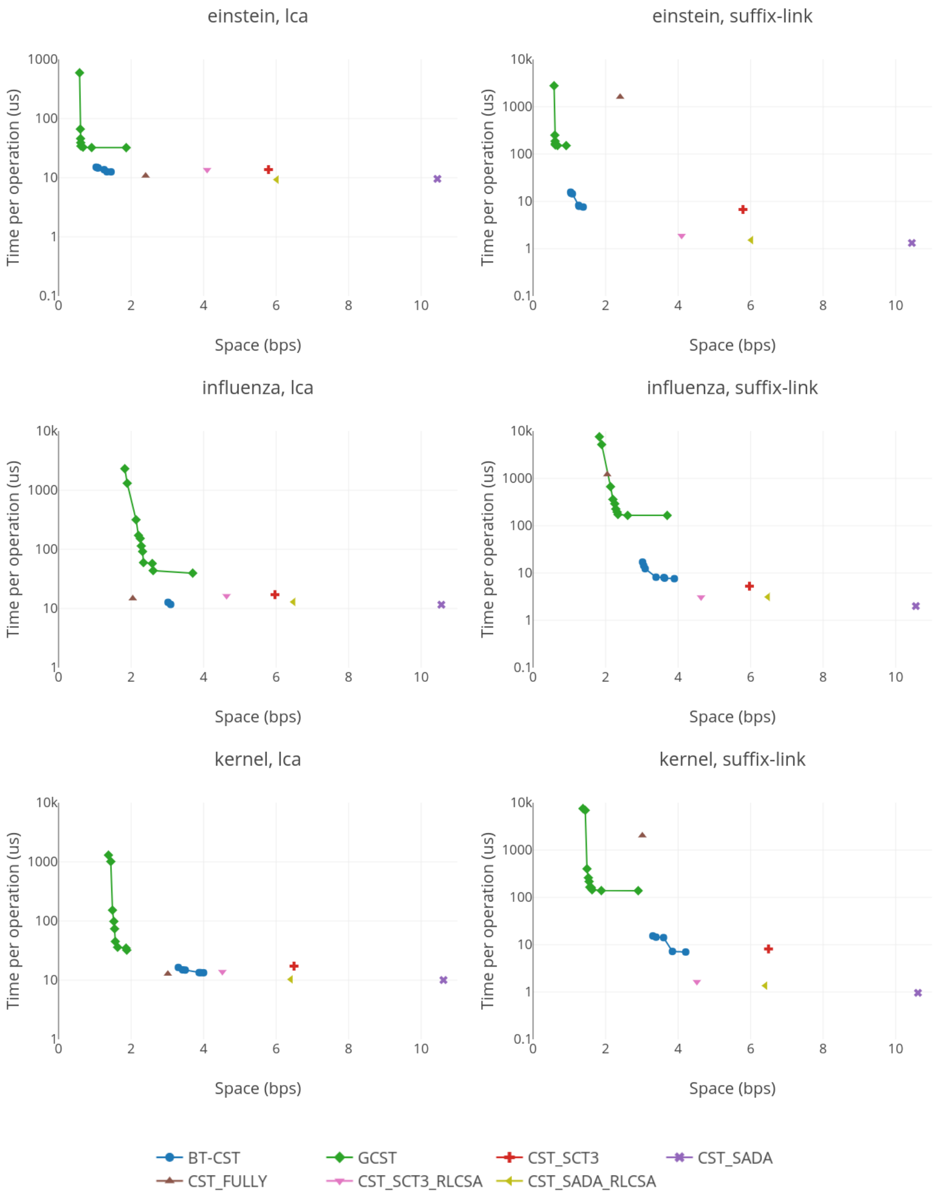

Fig. 2: Performance of CSTs for operations lca and suffix-link. The y-axis is in log-scale.

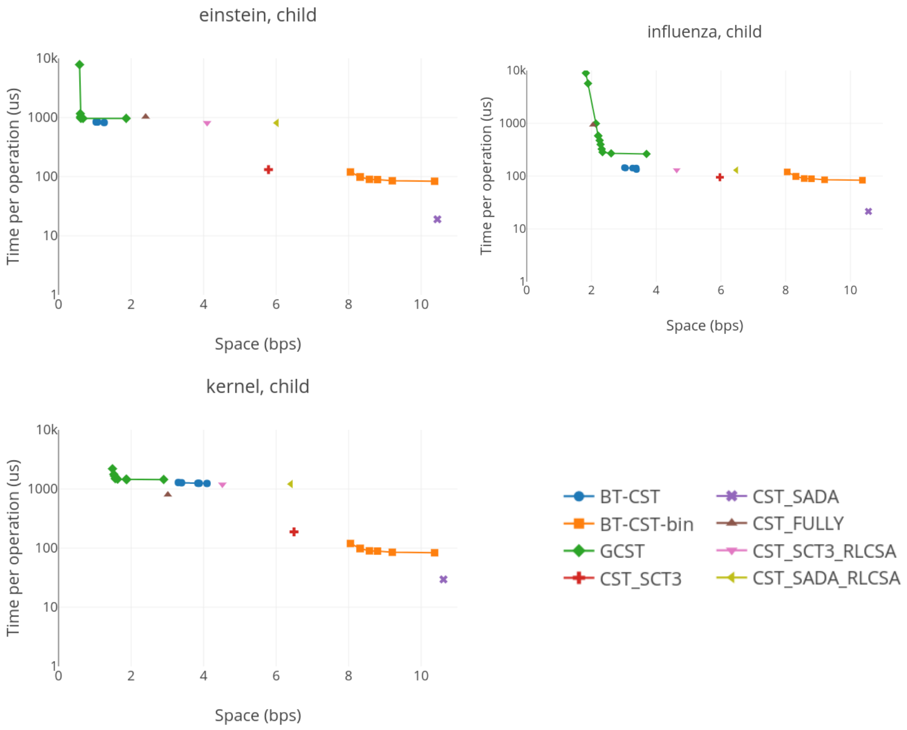

Fig. 3: Performance of CSTs for operation child. The y-axis is in log-scale. BT-CST-bin is BT-CST with binary search for child.

up to 10 (CST_SCT3) and 100 (CST_SADA) times faster. Our BT-CST-bin variant is faster than the base BT-CST by 15% on einstein and 18% on kernel, and outperforms the RLCSA-based indexes. On DNA, instead, most of the indexes take nearly 100 μsec, except for CST_SADA, which is several times faster; GCSA, which is a few times slower; and CST_FULLY, which stays in the msec.

Figure 4 shows the results for the maximal substrings problem. BT-CST sharply dominates an important part of the Pareto-curve, including the sweet point at 3.5 bps and 200-300 μsec per symbol. The other structures for repetitive collections take either much more time and slightly less space (GCSA, 1.5–2.5 times slower), or significantly more space and slightly less time (CST_SCT3, 45% more space and around 200 μsec). CST_SADA is around 10 times faster, the same as its CSA when solving the dominant operation, child.

6 Conclusions and Future Work

We have introduced the Block-Tree Compressed Suffix Tree (BT-CST), a new compressed suffix tree aimed at indexing highly repetitive text collections. Its

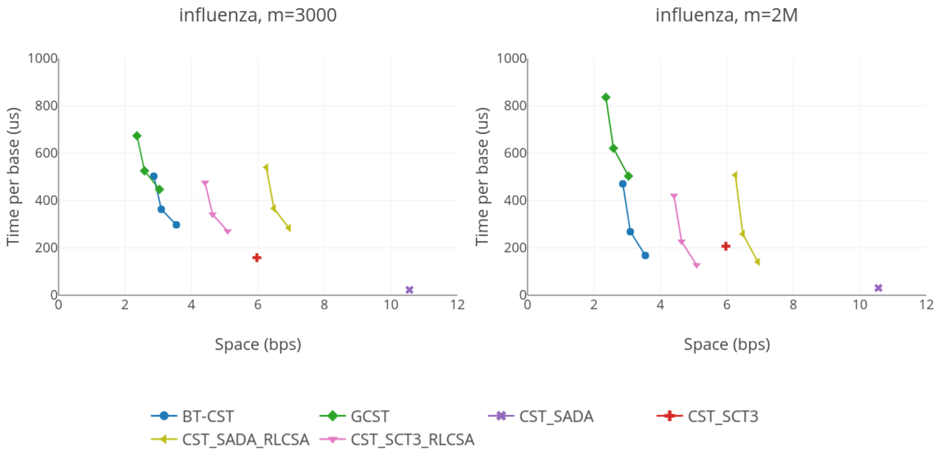

Fig. 4: Performance of CSTs when solving the maximal substrings problem. The y-axis is time in microseconds per base in the smaller sequence (of length m).

main feature is the BT-CT component, which uses Block Trees to represent the parentheses-based topology of the suffix tree and exploit the repetitiveness it inherits from the text collection. Block Trees [6] represent a sequence in space close to its Lempel-Ziv complexity (with a logarithmic-factor penalty), in a way that logarithmic-time access to any element is supported. The BT-CT enhances Block Trees with the more complex operations needed to simulate tree navigation on the parentheses sequence, as needed by the suffix tree operations.

Our experimental results show that the BT-CST requires 1–3 bits per symbol in highly repetitive text collections, which is slightly larger than the best previous alternatives [26], but also significantly faster (often by an order of magnitude). In particular, the BT-CT component uses 0.3–1.5 bits per node on these suffix trees and it takes a few microseconds to simulate the tree navigation operations, which is close to the time obtained by the classical 2.8-bit-per-node representation that is blind to repetitiveness [27]. This structure may be interesting to represent other repetitive trees beyond compressed suffix tree topologies, for example those arising in XML datasets, JSON repositories, and many others.

Although we have shown that in practice they perform as well as their classical counterpart [27], an interesting open problem is whether the operations *fwd-search*, *bwd-search*, and *min-excess* can be supported in polylogarithmic time on Block Trees. This was possible on perfectly balanced trees [27] and even on balanced-grammar parse trees [26], but the ability of Block Trees to refer to a prefix or a suffix of a block makes this more challenging. We note that the algorithm described by Belazzougui et al. [6] claiming logarithmic time for *min-excess* does not work (as checked with coauthor T. Gagie).

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. Alberto Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, pages 85–96. Springer, 1985.
4. Diego Arroyuelo, Francisco Claude, Sebastian Maneth, Veli Mäkinen, Gonzalo Navarro, Kim Nguyễn, Jouni Sirén, and Niko Välimäki. Fast in-memory xpath search using compressed indexes. *Software Practice and Experience*, 45(3):399–434, 2015.
5. Djamal Belazzougui and Fabio Cunial. Representing the suffix tree with the CDAWG. In *Proc. 28th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 7:1–7:13, 2017.
6. Djamal Belazzougui, Travis Gagie, Pawel Gawrychowski, Juha Kärkkäinen, Alberto Ordónez, Simon J. Puglisi, and Yasuo Tabei. Queries on LZ-bounded encodings. In *Proc. Data Compression Conference (DCC)*, pages 83–92, 2015.
7. David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 383–391, 1996.
8. Andrea Farruggia, Travis Gagie, Gonzalo Navarro, Simon J Puglisi, and Jouni Sirén. Relative suffix trees. *The Computer Journal*, 61(5):773–788, 2018.
9. Paolo Ferragina and Roberto Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.
10. Johannes Fischer, Veli Mäkinen, and Gonzalo Navarro. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science*, 410(51):5354–5364, 2009.
11. Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in BWT-runs bounded space. *CoRR*, 1705.10382, 2017. URL: arxiv.org/abs/1705.10382.
12. Simon Gog. *Compressed Suffix Trees: Design, Construction, and Applications.* PhD thesis, University of Ulm, Germany, 2011.
13. Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.
14. Wing-Kai Hon, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Space-efficient frameworks for top-k string retrieval. *Journal of the ACM*, 61(2):9:1–9:36, 2014.
15. John C. Kieffer and En-Hui Yang. Grammar-based codes: a new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000.
16. Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.
17. Stefan Kurtz. Reducing the space requirement of suffix trees. *Software Practice and Experience*, 29(13):1149–1171, 1999.
18. Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.
19. Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Transactions on information theory*, 22(1):75–81, 1976.

20. Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.
21. Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
22. Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
23. Maxim Mozgovoy, Kimmo Fredriksson, Daniel White, Mike Joy, and Erkki Sutinen. Fast plagiarism detection system. In *Proc. 12th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 267–270, 2005.
24. Gonzalo Navarro. Indexing highly repetitive collections. In *Proc. 23rd International Workshop on Combinatorial Algorithms (IWOCA)*, pages 274–279, 2012.
25. Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), 2007.
26. Gonzalo Navarro and Alberto Ordóñez. Faster compressed suffix trees for repetitive collections. *Journal of Experimental Algorithmics*, 21(1):1–8, 2016.
27. Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3):16, 2014.
28. Enno Ohlebusch, Johannes Fischer, and Simon Gog. CST++. In *Proc. 17th International Conference on String Processing and Information Retrieval (SPIRE)*, pages 322–333, 2010.
29. Alberto Ordóñez. *Statistical and repetition-based compressed data structures*. PhD thesis, Universidade da Coruña, 2016.
30. 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.
31. Rajeev Raman and S. Srinivasa Rao. Succinct representations of ordinal trees. In *Space-Efficient Data Structures, Streams, and Algorithms*, pages 319–332. Springer, 2013.
32. Luís M. S. Russo, Gonzalo Navarro, and Arlindo L. Oliveira. Fully compressed suffix trees. *ACM Transactions on Algorithms*, 7(4):53:1–53:34, 2011.
33. Kunihiko Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.
34. Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.
35. Sarah A. Tishkoff and Kenneth K. Kidd. Implications of biogeography of human populations for 'race' and medicine. *Nature Genetics*, 36:S21–S27, 2004.
36. Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
37. Peter Weiner. Linear pattern matching algorithms. In *Proc. 14th Annual Symposium on Switching and Automata Theory (FOCS)*, pages 1–11, 1973.
38. Dell Zhang and Wee Sun Lee. Extracting key-substring-group features for text classification. In *Proc. 12th Annual International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 474–483, 2006.