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

Manuel Cáceres[1,2] and Gonzalo Navarro[2,3]

[1]Department of Computer Science, University of Helsinki, Finland, manuel.caceresreyes@helsinki.fi
[2]CeBiB — Center for Biotechnology and Bioengieering, Chile
[3]Department of Computer Science, University of Chile, Chile, gnavarro@dcc.uchile.cl

Abstract

The suffix tree is a fundamental data structure in stringology, but its space usage, though linear, is an important problem in applications like Bioinformatics. We design and implement a new *compressed suffix tree (CST)* targeted to highly repetitive texts, such as large genomic collections of the same species. Our first contribution is to enhance the Block Tree, a data structure that captures the repetitiveness of its input sequence, to represent the topology of trees with large repeated subtrees. Our so-called Block-Tree Compressed Topology (BT-CT) data structure augments the Block Tree nodes with data that speeds up tree navigation. Our Block-Tree CST (BT-CST), in turn, uses the BT-CT to compress the topology of the suffix tree, and also replaces the sampling of the suffix array and its inverse with grammar- and/or Block-Tree-based representations of those arrays.

Our experimental results show that BT-CTs reach navigation speeds comparable to compact tree representations that are insensitive to repetitiveness, while using 2–10 times less space on the topologies of the suffix trees of repetitive collections. Our BT-CST is slightly larger than previous repetition-aware suffix trees based on grammar-compressed topologies, but outperforms them in time, often by orders of magnitude.

1 Introduction

Suffix trees [1, 2, 3] are one of the most appreciated data structures in Stringology [4] and application areas like Bioinformatics [5], 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 [6], authorship attribution [7], document retrieval [8], and others.

While their linear space complexity is regarded as acceptable in classical terms, the actual space usage of suffix trees 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 [9] require at least 10 bytes per symbol, which forces many applications to run the suffix tree on (the orders of magnitude slower) secondary memory.

[*]Funded by Fondecyt Grant 1-200038 and Basal Funds FB0001, ANID, Chile. A preliminary partial version of this paper appeared in Proc. SPIRE 2019.

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 $\lg 4 = 2$ 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 [10, 11]. 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 [12], 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 [13] by raising the space to about 6 bytes (48 bits) per character. Those numbers get even larger when the text collection is longer than 4 billion bases and 32-bit numbers do not suffice.

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 [14]. A recent implementation [15] uses, on DNA, about 10 bits per base and supports the operations in a few microseconds. Even smaller, though slower, CSTs have been proposed [16, 17], reaching the extreme of using as little as 5 bits per base [18], though in this case the operation times already raise to milliseconds, close to a secondary-memory deployment.

Still, further space reductions are possible (and necessary!) when facing large genome repositories, thanks to the fact that DNA sequences of two humans differ by less than 0.5% [19]. Many other large text collections are equally repetitive, for example versioned document collections and software repositories. This repetitiveness is not well captured by statistical compression methods [20], on which most of the CSTs are based. Lempel-Ziv [21] and grammar [22] based compression techniques, among others, do better in this scenario [23], but only recently we have seen CSTs building on them, both in theory [24, 25] and in practice [17, 26]. The most successful CST in practice for repetitive collections is the grammar-compressed suffix tree (GCST [26]), which on DNA uses about 2 bits per base and supports the operations in tens to hundreds of microseconds.

GCSTs use grammar compression on the parentheses sequence that represents the suffix tree topology [27], 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 [28, 29], which offer similar approximation ratios to Lempel-Ziv compression, but promise faster access.

Our first 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-node representations that are blind to repetitiveness [30]. 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 [31].

Our second contribution improves the performance of suffix tree operations that make heavy use of its compressed suffix array (CSA). Repetition-aware CSTs [17, 26] use the run-length CSA

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 lexicographic order
tree-depth(v)	The number of edges from root() to v
next-sibling(v)	The next sibling of v in lexicographic order
previous-sibling(v)	The previous sibling of v in lexicographic order
parent(v)	The parent of v
is-ancestor(v,u)	True if v is an ancestor of u
level-ancestor(v,d)	The ancestor of v at tree depth tree-depth(v) $- d$
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. The first group are tree topology operations, while the second is specific of suffix trees.

(RLCSA) [32], whose sampling of the suffix array and its inverse has been difficult to compress. We show that replacing these samplings by grammar-compressed or Block-Tree-compressed representations of those arrays achieves a time improvement of up to two orders of magnitude on such operations, and better compression when the repetitiveness is very high.

Our new suffix tree, BT-CST, uses the BT-CT to represent the suffix tree topology and some enhanced version of the RLCSA. Although larger than the GCST, it still requires about 3 bits per base in highly repetitive DNA collections. In exchange, it is considerably faster than the GCST, often by an order of magnitude.

2 Preliminaries and Related Work

A *text* $T[1,n] = T[1] \cdots T[n]$ 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 $T[i,j] = T[i] \cdots T[j]$. A substring $T[i,j]$ is a prefix if $i = 1$ and a suffix if $j = n$.

The *suffix tree* [1, 2, 3] of a text T is a trie of its suffixes in which unary paths are collapsed into a single edge and leaves representing the suffix $T[j,n]$ store the number j. The tree then has less than $2n$ nodes, thus a classical/pointer-based implementation uses $\Theta(n \lg n)$ bits. The suffix tree supports a set of operations (see Table 1) that suffices to solve a large number of problems in Stringology [4] and Bioinformatics [5].

The *suffix array* [12] $A[1,n]$ 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 lexicographic order. If the suffix tree stores the children in increasing lexicographic order of the corresponding suffixes, then the concatenation of the numbers stored at the leaves forms the suffix array. Further, the leaves descending from a suffix tree node span a range of suffixes in A.

One well-known functionality of the suffix array is to *count* the occurrences of a pattern $S[1,m]$

in T in $O(m \lg n)$ character comparisons, which is achieved with two binary searches on A.

The function $lcp(X, Y)$ is the length of the longest common prefix (lcp) of strings X and Y. The *LCP array* [12], $LCP[1, n]$, 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 $T[1, n]$. When the suffix array is enhanced with a variant of the LCP array, *count* is possible in time $O(m + \lg n)$ [12], and if even more information is added, it can achieve $O(m)$ time [13]. Note that the string-depth of the *lowest common ancestor* between two consecutive suffix tree leaves is precisely its corresponding LCP entry.

2.1 Succinct tree representations: BP

A balanced parentheses (BP) representation (there are others [27]) 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:

- $access(i) = P[i]$, the bit at position i.
- $rank_{0|1}(i) = |\{1 \leq j \leq i; P[j] = 0|1\}|$, the number of 0|1s up to position i.
- $excess(i) = rank_1(i) - rank_0(i)$, the number of open parentheses minus closing parentheses up to position i.
- $select_{0|1}(i) = \min(\{j; rank_{0|1}(j) = i\} \cup \{\infty\})$, the position of the i-th 0|1.
- $leaf\text{-}rank(i) = rank_{10}(i) = |\{1 \leq j \leq i-1; P[j] = 1 \wedge P[j+1] = 0\}|$, the number of leaves up to position i.
- $leaf\text{-}select(i) = select_{10}(i) = \min(\{j; leaf\text{-}rank(j+1) = i\} \cup \{\infty\})$, the position of the open parenthesis of the j-th leaf.
- $fwd\text{-}search(i, d) = \min(\{j > i; excess(j) = excess(i) + d)\} \cup \{\infty\})$, the least we have to move forward from i for the excess to grow by d units.
- $bwd\text{-}search(i, d) = \max(\{j < i; excess(j) = excess(i) + d)\} \cup \{-\infty\})$, the least we have to move backward from i for the excess to grow by d units.
- $min\text{-}excess(i, j) = \min(\{excess(k) - excess(i-1); i \leq k \leq j\})$, the minimum excess in $P[i, j]$.

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 [30]. These include the operations needed by suffix trees. For example, interpreting nodes v as the position of their opening parenthesis in P, it holds that tree-depth$(v) = excess(v)$, next-sibling$(v) = fwd\text{-}search(v, -1) + 1$, lca$(v, u) =$ parent$(fwd\text{-}search(v - 1, min\text{-}excess(v, u)) + 1)$, parent$(v) =$ level-ancestor$(v, 1)$, and level-ancestor$(v, d) = bwd\text{-}search(v, -d - 1) + 1$.

2.2 Compressed Suffix Arrays (CSAs)

Compressed Suffix Arrays (CSAs) [33] represent a text T and its suffix array A within $O(n \lg \sigma)$ bits, and often in space close to that of a compressed representation of T. They provide access to the suffix array and its inverse (i.e., return any $A[i]$ and $A^{-1}[j]$), to the text (i.e., return any $T[i, j]$), and often 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, $T[j + 1, n]$, working at their positions in A: if $A[i] = j$, then $A[\Psi[i]] = j + 1$. CSAs usually support queries $count(S)$ (returning the number of times S occurs as a substring in T) and $locate(S)$ (returning the positions of those occurrences).

The most effective of these indexes can be classified into two groups. The first group [34, 35] takes advantage of properties of the array Ψ to compress it. It also stores samples of A and A^{-1} taken at every s-th text position, so as to recover the original values in $O(s)$ time using their interplay with Ψ: for A we sample all values of the form $A[i] = s \cdot j$, so if we want to recover $A[i]$, we apply Ψ iteratively until we get to a sampled position $A[\Psi^k[i]]$, and then $A[i] = A[\Psi^k[i]] - k$. To compute $A^{-1}[j]$ we start from the sampled value $A^{-1}[j']$, for $j' = \lfloor j/s \rfloor \cdot s$ and then have $A^{-1}[j] = \Psi^{j'-j}[A^{-1}[j']]$.

The indexes of the second group [36] are called FM-indexes. One of the most successful implementations [37] of these indexes uses *access* and *rank* queries on the *Burrows Wheeler Transform* (BWT), a permutation T_{BWT} of the characters of T such that $T_{BWT}[i] = T[A[i] - 1]$ (and \$ if $A[i] = 1$), that is, the character preceding each suffix in lexicographic order. The index is built on T_{BWT} and the same samplings to compute A and A^{-1}. Instead of the function Ψ, it computes the function $LF(i) = \Psi^{-1}[i]$ using *rank* and *access* on T_{BWT}, and thus the sampling mechanism works analogously. Operations *access* and *rank* typically take time $O(\lg \sigma)$.

2.3 Repetition-aware CSAs

Both classes of CSAs use space bounded by the statistical entropy of T, which is however insensitive to the its repetitiveness. Repetitions in T generate long runs of equal letters in T_{BWT}, and also long runs of consecutive increasing values in Ψ. Considering these runs to further compress the indexes gives birth to *repetition-aware CSAs*. One is the *Run-Length CSA* (RLCSA) [32], which run-length compresses the runs of 1s in the differential Ψ array, $D\Psi[i] = \Psi[i] - \Psi[i - 1]$, and stores absolute values of Ψ at sampled runs. Another is the *Run-Length FM-Index* (RLFMI) [38], which run-length compresses T_{BWT} and stores additional structures to translate the *access* and *rank* queries on T_{BWT}.

Both structures use $O(r \lg n)$ bits, r being the number of runs, and count in time $O(m \lg n)$. Further, they can access A and A^{-1} in $O(s \lg n)$ time using $O((n/s) \lg n)$ extra bits for the sampling. When T is repetitive, r is small and the space for the sampling of A and A^{-1} becomes dominant.

An attempt to break the space/time barrier of the sampling is the *Locally Compressed Suffix Array* (LCSA) [39]. It uses Re-Pair [40] to grammar-compress the differential suffix array, $DA[i] = A[i] - A[i - 1]$. The compression succeeds on repetitive texts because, if $D\Psi[k + 1] = 1$ on a run $i \le k \le j$, then $DA[k + 1] = DA[\Psi[k] + 1]$ for all $i \le k \le j$ becomes a repetition that a grammar compressor can capture. With some care, one can obtain a grammar of size $O(r \lg(n/r))$ that extracts any symbol of A and A^{-1} in time $O(\lg(n/r))$ [24], though in practice Re-Pair produces much smaller grammars.

Another very recent attempt [41] compresses A using Relative Lempel-Ziv [42] on DA. They sample a suitable reference from DA and represent the array as a sequence of pointers to the reference, which ensures fast access and good compression if T is repetitive.

2.4 Compressed Suffix Trees (CSTs)

Sadakane [14] 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 [35]; 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 [15] of this index requires about 10 bits per character and takes a few microseconds per operation.

Russo et al. [18] 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 [18] uses as little as 5 bits per character, but the operations take milliseconds, nearly as slow as running in secondary storage.

Fischer et al. [16] also obtain $o(n)$ 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, and used this fact to run-length compress H. Abeliuk et al. [17] 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.

2.5 Repetition-aware CSTs

Abeliuk et. al [17] also presented the first CST for repetitive collections. They built on the third approach above [16], so they do not represent the tree topology. They use the RLCSA [32] and a grammar compression on the differential LCP array, $DLCP[i] = LCP[i] - LCP[i-1]$. The nodes of the parse tree (obtained with Re-Pair [40]) 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. Their index uses 1–2 bits per character on repetitive collections. It is rather slow, however, operating within (many) milliseconds. Gagie et al. [24] show that $O(r \lg(n/r))$ space can be guaranteed for this design if one uses a particular Run-Length Context-Free Grammars [43] on $DLCP$, DA and DA^{-1}, while supporting most operations in $O(\lg n)$ time. Still, heuristic grammar constructions such as Re-Pair work better in practice.

Navarro and Ordóñez [26] include again the tree topology. Since text repetitiveness induces isomorphic subtrees in the suffix tree, they grammar-compress the BP representation. The nonterminals are enriched to support the tree navigation operations. 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. [17]: their structure operates in tens to hundreds of microseconds per operation, getting closer to the times of general-purpose CSTs.

Recent work by Farruggia et al. [44] builds on Relative Lempel-Ziv [42] to compress the suffix trees of the individual sequences (instead of that of the whole collection). They showed to be time- and space-competitive against the CSTs mentioned, but their structure offers a different functionality (useful for other problems). Belazzougui and Cunial's [25] CST based on the CDAWG [45] (a minimized automaton that recognizes all the substrings of T), also supports most operations in time $O(\lg n)$. However, experiments [24] using CDAWG show that it uses significantly more space than other repetition-aware techniques.

Very recently, Boucher et al. [46] build a CST for repetitive collections based on *prefix-free parsing*, which aims to parse the text consistently so as to obtain a small list of different factors and represent the text as a concatenation of those. This allows them to build the CST on very

Algorithm 1 Accessing $P[i]$ with the Block Tree of P, invoked as $access(root, i, |P|)$.

```
Function access(v, i, b)
if $v$ is a LeafBlock then return $v.blk[i]$ ;
if $v$ is a BackBlock then
    if $v.off + i \leq b$ then return $access(v.ptr, i + v.off, b)$ ;
    else return $access(v.ptr.next, i + v.off - b, b)$ ;
return $access(v.child[\lfloor (i-1)/\kappa \rfloor + 1], ((i-1) \bmod \kappa) + 1, b/\kappa)$
```

large text collections fast and using little extra space, which is their focus and a weakness of many other CSTs. The resulting CST, however, is typically 5–10 times larger than the CST we present in this article, and also significantly slower for most operations, according to their experiments.

3 Block Trees

A Block Tree [28] is a full κ-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), left to right, represent a partition of P into blocks of length $b = |P|/\kappa^d$ (we pad P to ensure these numbers are integers). A node v, representing some block $v.blk = P[i, i + b - 1]$, can be of three types:

LeafBlock: If $b \leq ll$, where ll 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 then stores a pointer $v.ptr = u$ to the node u such that the first occurrence of $v.blk$ starts inside $u.blk = P[j, j+b-1]$. If it occurs at $P[j + o, j + o + b - 1]$, this offset inside $u.blk$ is stored in $v.off = o$. Node u also stores a pointer $u.next$ to its following node, which is necessary to access the content of $v.blk$. Node v, and its block, are not anymore considered at deeper levels.

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

Figure 1 illustrates a Block Tree (some field names are introduced later). The key property of the Block Tree is that, if a BackBlock v points leftwards to u, then node u is an InternalBlock. The Block Tree can then return any symbol $P[i]$ in time proportional to the height of the Block Tree, 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. Algorithm 1 gives the pseudocode.

If the sequence P is parsed by the Lempel-Ziv algorithm into z phrases, then the Block Tree has $O(\kappa z \lg_\kappa(n/z))$ nodes, which for constant κ matches the best space guarantee offered by grammar-based compressors. Further, the Block Tree was shown to be of size $O(\kappa\delta \lg_\kappa(n/\delta))$, where $\delta \leq z$ is a stricter measure of repetitiveness for P; this bound cannot be reached by context-free grammars [47]. A root-to-leaf traversal in the Block Tree (e.g., to access $P[i]$ with Algorithm 1) takes time $O(\lg_\kappa(n/ll))$.

More recently [48], the Block Tree construction was optimized without affecting its worst-case time and space guarantees: some InternalBlocks are converted into BackBlocks even if they do

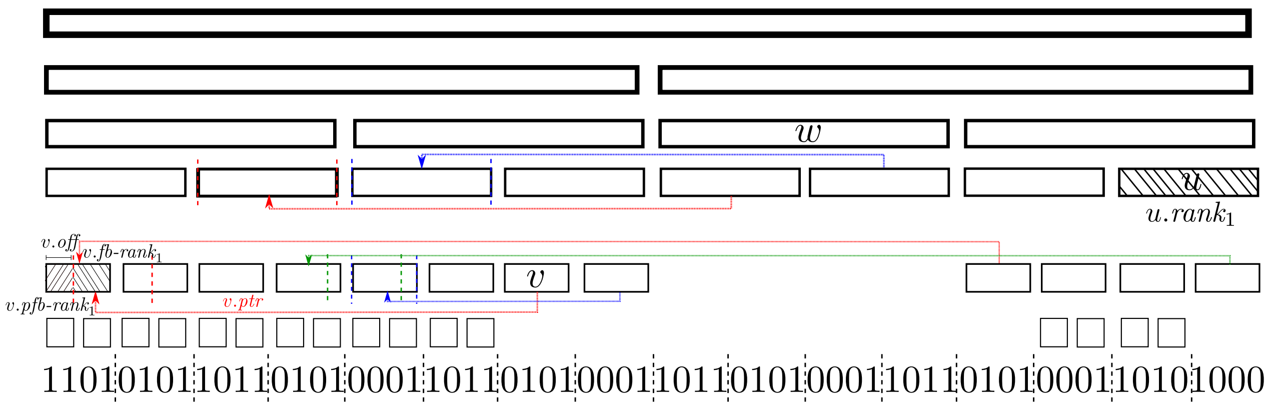


Figure 1: Block Tree for a bit sequence representing the BP of a tree topology. Nodes have been put on top of the substrings they represent, and the child pointers were eliminated for clarity. We represent various fields of BackBlock v, which occurs between the red dashed lines, and of node u. Note that node w is not a BackBlock even when $w.blk$ is not a leftmost occurrence, because its sibling contains a leftmost occurrence. On the other hand, the children of w are BackBlocks because, concatenated with their neighbors, they do not contain leftmost occurrences.

not satisfy the stated conditions, as long as they are not referenced by other BackBlocks to the right. In the same work, the author studies a number of compact representations of Block Trees; we use the one recommended in there for our implementation. In particular, all the first levels not containing BackBlocks are removed to speed up traversals, and various pointers are eliminated by using a levelwise deployment and bitvectors to mark the block types.

4 Our Block-Tree Compressed Topology (BT-CT)

This section describes our main data structure, the *Block-Tree Compressed Topology (BT-CT)*, which compresses a parentheses sequence and supports navigation on the tree topology it represents.

4.1 Block Tree augmentation

To support the primitive operations we start from a simple Block Tree representing a BP sequence P, supporting only *access*, and introduce a number of additional fields. Many of those fields can be computed from others, so we divide the description into *stored fields*, which are actually stored in the compact representation of BT-CT and *fields computed on the fly*, which are inferred from the stored fields.

Stored fields

- For every node v that represents the block $v.blk = P[i, i+b-1]$:
  - $rank_1$, the number of 1s in $v.blk$, i.e., $rank_1(P, i+b-1) - rank_1(P, i-1)$.
  - *lrank* (leaf rank), the number of 10s (i.e., leaves in BP) that finish inside $v.blk$, i.e., $leaf\text{-}rank(P, i+b-1) - leaf\text{-}rank(P, i-1)$.
  - *lbreaker* (leaf breaker), a bit telling whether the first symbol of $v.blk$ 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 $v.blk$, i.e., $\mathit{min\text{-}excess}(P, i, i+b-1)$.

- For every BackBlock node v that represents $v.blk = P[i, i+b-1]$ and points to its first occurrence $O = P[j+o, j+o+b-1]$ inside $u.blk = P[j, j+b-1]$ with offset $v.\mathit{off} = o$:

  - *fb-rank*$_1$, the number of 1s in the prefix of O contained in $u.blk$ ($O \cap u.blk$, the 1st block spanned by O), i.e., $\mathit{rank}_1(P, j+b-1) - \mathit{rank}_1(P, j+o-1)$.
  - *fb-lrank*, the number of 10s that finish in $O \cap u.blk$, i.e., $\mathit{leaf\text{-}rank}(P, j+b-1) - \mathit{leaf\text{-}rank}(P, j+o-1)$.
  - *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 $O \cap u.blk$, i.e., $\mathit{min\text{-}excess}(P, j+o, j+b-1)$.
  - *m-fb*, a bit telling whether the minimum excess of $u.blk$ is reached in $O \cap u.blk$, i.e., whether $\mathit{min\text{-}excess}(P, i, i+b-1) = \mathit{min\text{-}excess}(P, j+o, j+b-1)$.

Fields computed on the fly

- For every node v that represents $v.blk = P[i, i+b-1]$:

  - *rank*$_0$, the number of 0s in $v.blk$, i.e., $b - v.\mathit{rank}_1$.
  - *excess*, the excess of 1s over 0s in $v.blk$, i.e., $v.\mathit{rank}_1 - v.\mathit{rank}_0 = 2 \cdot v.\mathit{rank}_1 - b$.

- For every BackBlock node v that represents $v.blk = P[i, i+b-1]$ and points to its first occurrence $O = P[j+o, j+o+b-1]$ inside $u.blk = P[j, j+b-1]$ with offset $v.\mathit{off} = o$:

  - *fb-rank*$_0$, the number of 0s in $O \cap u.blk$, i.e., $(b-o) - v.\mathit{fb\text{-}rank}_1$.
  - *pfb-rank*$_{0|1}$, the number of 0s|1s in the prefix of $u.blk$ that precedes O ($u.blk - O$), i.e., $u.\mathit{rank}_{0|1} - v.\mathit{fb\text{-}rank}_{0|1}$.
  - *fb-excess*, the excess in $O \cap u.blk$, i.e., $v.\mathit{fb\text{-}rank}_1 - v.\mathit{fb\text{-}rank}_0$.
  - *sb-excess*, the excess in $O - u.blk$ (2nd block spanned by O), i.e., $v.\mathit{excess} - v.\mathit{fb\text{-}excess}$.
  - *pfb-lrank*, the number of 10s that finish in $u.blk - O$, i.e., $u.\mathit{lrank} - v.\mathit{fb\text{-}lrank}$.
  - *sb-mexcess*, the minimum excess in $O - u.blk$, i.e., $\mathit{min\text{-}excess}(P, j+b, j+b+o-1)$. We store either $v.\mathit{fb\text{-}mexcess}$ or $v.\mathit{sb\text{-}mexcess}$, the one that differs from $v.\mathit{mexcess}$. To deduce the non-stored field we use *mexcess*, *fb-excess* and *m-fb*.
  - *Mexcess-suf*, *fb-Mexcess-suf*, and *sb-Mexcess-suf*. They are the analogous to the *mexcess* fields but considering the maximum reached in a right-to-left scan or suffix of the corresponding zone. Note that these fields can be computed from the corresponding *excess* and *mexcess* fields, because the maximum in a suffix is reached exactly next to the position where the minimum in a prefix is reached: $v.\mathit{Mexcess\text{-}suf} = v.\mathit{excess} - v.\mathit{mexcess}$, $v.\mathit{fb\text{-}Mexcess\text{-}suf} = v.\mathit{fb\text{-}excess} - v.\mathit{fb\text{-}mexcess}$, and $v.\mathit{sb\text{-}Mexcess\text{-}suf} = v.\mathit{sb\text{-}excess} - v.\mathit{sb\text{-}mexcess}$.

4.2 Operations

We now describe how the operations described in Section 2.1 are implemented on our augmented Block Trees.

Algorithm 2 Computing $rank_c(P, i)$ on the Block Tree of P, invoked as $rank(root, i, c, |P|)$.

```
Function rank(v, i, c, b)
if $v$ is a LeafBlock then return the number of $c$s in $v.blk[1, i]$ (by brute force) ;
if $v$ is a BackBlock then
    if $v.off + i \leq b$ then return $rank(v.ptr, i + v.off, c, b) - v.pfb\text{-}rank_c$ ;
    else return $rank(v.ptr.next, i + v.off - b, c, b) + v.fb\text{-}rank_c$ ;
$d \leftarrow \lfloor (i-1)/\kappa \rfloor$
$p \leftarrow 0$
for $k \leftarrow 1$ to $d$ do $p \leftarrow p + v.child[k].rank_c$ ;
return $p + rank(v.child[d+1], ((i-1) \bmod \kappa) + 1, c, b/\kappa)$
```

Algorithm 3 Computing $select_c(P, j)$ on the Block Tree of P, invoked as $select(root, j, c, |P|)$.

```
Function select(v, j, c, b)
if $v$ is a LeafBlock then return the position of the $j$-th $c$ in $v.blk$ (by brute force) ;
if $v$ is a BackBlock then
    if $j \leq v.fb\text{-}rank_c$ then return $select(v.ptr, j + v.pfb\text{-}rank_c, c, b) - v.off$ ;
    else return $select(v.ptr.next, j - v.fb\text{-}rank_c, c, b) + b - v.off$ ;
$k \leftarrow 1$
while $j > v.child[k].rank_c$ do
    $j \leftarrow j - v.child[k].rank_c$
    $k \leftarrow k + 1$
return $(k-1) \cdot (b/\kappa) + select(v.child[k], j, c, b/\kappa)$
```

4.2.1 Rank and select of bits and leaves

Algorithms 2 and 3 show how operations $rank_c$ and $select_c$ are implemented, for $c = 0$ or 1, in time $O(\kappa \lg_\kappa(n/ll) + ll)$. The pseudocode is self-explanatory (see Figure 1 again).

The implementations of *leaf-rank* and *leaf-select* are analogous to those of $rank_c(i)$ and $select_c(i)$, respectively, using the field *lrank* instead of $rank_c$. The other difference is that we must correct the counts to consider 10s at block borders: (1) in LeafBlocks, we use the field *lbreaker* to check whether the block starts with a leaf (so as to increase the leaf count); (2) in BackBlocks we consider fields *lbreaker* and *fb-lbreaker* to check whether we have to increment or decrement the leaf count when moving to a leftward node; and (3) in InternalBlocks we use the *lbreaker* of the children to correct the cumulative sum of leaves.

4.2.2 Forward and backward searches

We describe in detail how to solve *fwd-search*(i, d) for $d \leq 0$; the case $d > 0$, as well as *bwd-search*(i, d), are solved analogously. We then 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*(v, i, j, b, d) on the current node v and $b = |v.blk|$, with a global excess variable e that is updated as we traverse $v.blk[i, j]$. The procedure is initially called with *fwd-search*$(root, i+1, n, n, d)$ and $e \leftarrow 0$. The general idea is to traverse the range of the current node v left to right, using the fields $v.mexcess$, $v.fb\text{-}mexcess$ and $v.sb\text{-}mexcess$ to speed up the procedure when possible. If at some point e becomes d, we have found the answer to the search.

Algorithm 4 shows the pseudocode for *fwd-search*(v, i, j, b). Some observations are in order:

– In general, we can skip a block if its minimum excess exceeds d. For example, we first test if

Algorithm 4 Computing *fwd-search*(P, i, d) on the Block Tree of P, invoked as *fwd-search*$(root, i+1, |P|, |P|, d)$ with global variable $e \leftarrow 0$.

```
Function fwd-search(v, i, j, b, d)
if $j - i = b$ and $e + v.mexcess > d$ then
    $e \leftarrow e + v.excess$ ; return $\infty$
if v is a LeafBlock then
    for $k \leftarrow i$ to $j$ do
        if $v.blk[k] = 0$ then  $e \leftarrow e - 1$ ;
        else  $e \leftarrow e + 1$ ;
        if $e = d$ then  return $k$;
if v is a BackBlock then
    if $v.off + i \leq b$ then
        if $i = 1$ and $j \geq b - v.off$ and $e + v.fb\text{-}mexcess > d$ then
            $e \leftarrow e + v.fb\text{-}excess$
        else
            $f \leftarrow fwd\text{-}search(v.ptr, v.off + i, \min(v.off + j, b), b, d)$
            if $f \neq \infty$ then  return $f - v.off$ ;
    if $v.off + j > b$ then
        if $j = b$ and $i \leq b - v.off$ and $e + v.sb\text{-}mexcess > d$ then
            $e \leftarrow e + v.sb\text{-}excess$
        else
            $f \leftarrow fwd\text{-}search(v.ptr.next, \max(v.off + i - b, 1), v.off + j - b, b, d)$
            if $f \neq \infty$ then  return $f - v.off + b$;
if v is an InternalBlock then
    $k_1 \leftarrow \lfloor (i-1)/\kappa \rfloor$; $k_2 \leftarrow \lfloor (j-1)/\kappa \rfloor$
    $p \leftarrow (k_1 - 1) \cdot (b/\kappa)$
    for $k \leftarrow k_1$ to $k_2$ do
        $f \leftarrow fwd\text{-}search(v.child[k], \max(i, p+1) - p, \min(j, p + b/\kappa) - p, b/\kappa, d)$
        if $f \neq \infty$ then return $p + f$;
        $p \leftarrow p + b/\kappa$
return $\infty$
```

the search range spans the entire block $v.blk$ (i.e., $j-i = b$) and the answer cannot be reached inside v (i.e., $e + v.mexcess > d$). If so, we simply increase e by $v.excess$ and return ∞.

– If v is a BackBlock we translate the query to its original block O, which starts at offset $v.off$ in $u.blk$, where $u = v.ptr$. We first consider $O \cap u.blk$ (if not empty). We attempt to skip it, asking if the translated query covers $O \cap u.blk$ (i.e., if $i = 1$ and $j \geq b - v.off$) and if $e + v.fb\text{-}mexcess > d$. If not, we recursively call *fwd-search* on $O \cap u.blk$. If this does not yield an answer, we try on $O \cap u.next.blk$, if it is not empty. We also try to skip this second block, if it is contained in the translated query (i.e., $j = b$ and $i \leq b - v.off$) and $e + v.sb\text{-}mexcess > d$. If we cannot skip it, we recursively call *fwd-search* again on $O \cap u.next.blk$.

– If v is an InternalBlock, we identify the children of v that contain $v.blk[i, j]$ and scan them left to right until finding the answer. Note that, although we call *fwd-search* recursively on the children, those that are completely contained in the query and do not contain the answer are processed in $O(1)$ time due to the guard in the first line of the pseudocode. Thus, at most two of those recursive calls are nontrivial.

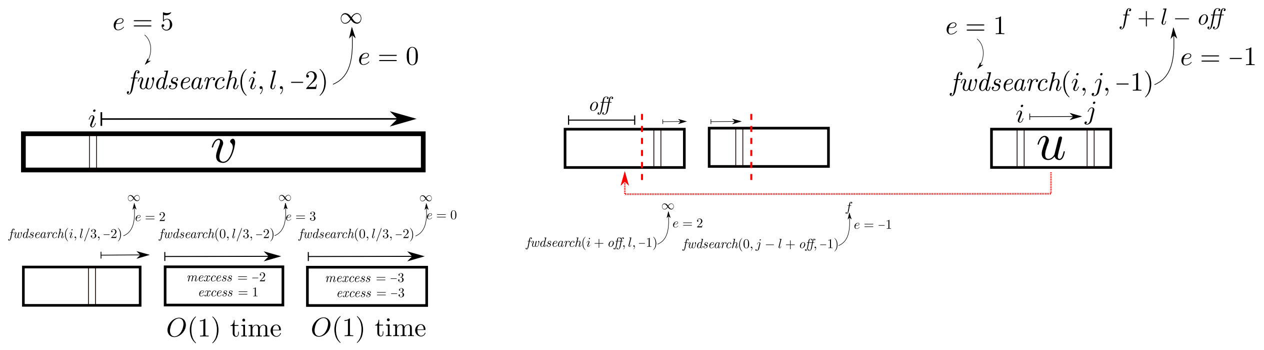


Figure 2: On the left, *fwd-search* is queried on a suffix of an InternalBlock v with a cumulative excess of $e = 5$. The query is translated to a suffix of the first child of v, which returns no answer and changes the cumulative excess to $e = 2$. Then the query continues, covering the second and third children of v completely, where it instantly (using the fields *mexcess* and *excess*) returns no answer and updates the corresponding cumulative excess. Finally, the original query returns no answer. On the right, *fwd-search* is queried on a substring of a BackBlock u with a cumulative excess of $e = 1$. The query is translated to a suffix of the first pointed block, where it returns no answer and changes the cumulative excess to $e = 2$. Then the query continues to a prefix of the second pointed block, where the answer is found and relocated to the original node u.

Figure 2 exemplifies the execution of the *fwd-search* algorithm.

Note that, although we look for various opportunities to use precomputed data to skip parts of the query, the operation *fwd-search* (as well as *bwd-search* and *min-excess*) is 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; see Figure 3. Our experiments will show that the algorithms perform well in practice.

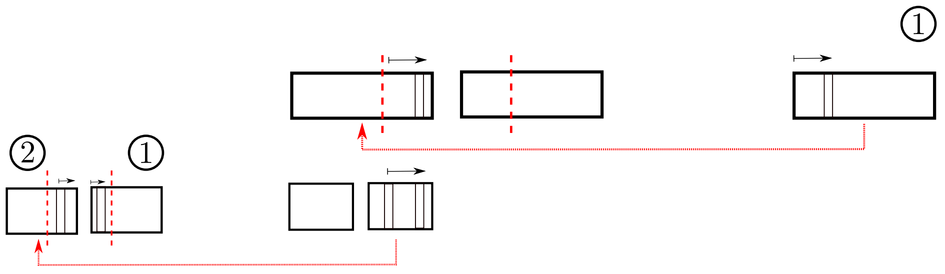
For *bwd-search*(i, d) we aim to find the largest $j < i$ where the excess of $P[j+1..i]$ is $-d \geq 0$. The idea is to traverse the range of the current node v right to left, updating e with the negative excess differences we scan, and looking for the position where $e = d$, while using the fields $v.$*Mexcess-suf*, $v.$*fb-Mexcess-suf* and $v.$*sb-Mexcess-suf* to try to speed up the procedure. For example, we can skip the whole v if $j - i + 1 = b$ and $e + v.$*Mexcess-suf* $< -d$. Upon BackBlocks, we first try with $v.ptr.next$ (skipping it if $j = b$, $i \leq b - v.off$, and $e + v.$*sb-Mexcess-suf* $< -d$) and, if the answer is not found in there, with $v.ptr$ (skipping it if $i = 1$, $j \geq b - v.off$, and $e + v.$*fb-Mexcess-suf* $< -d$).

4.2.3 Finding the minimum excess

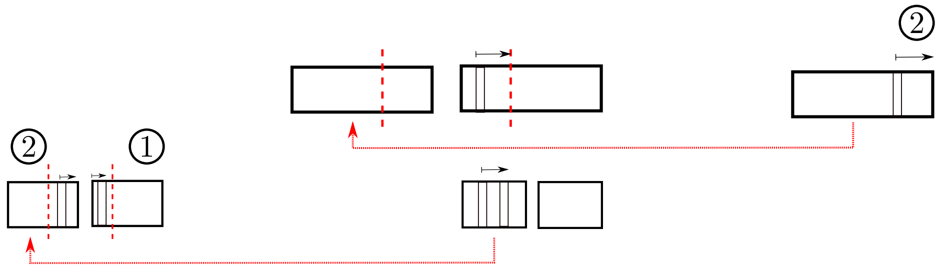
Operation *min-excess*(P, i, j) seeks the minimum excess in $P[i..j]$. It is implemented in Algorithm 5 with the recursive procedure *min-excess*$(root, i, j, |P|)$, with a global excess variable e. The idea is analogous to that of *fwd-search*: traverse the node left to right and use the *excess* and *mexcess* fields to speed up the traversal. Note that, again, only the first and last of the calls to the children of an InternalNode can take more than $O(1)$ time. Figure 4 shows two examples of the execution of the algorithm.

5 Our Repetition-Aware Compressed Suffix Tree

Following the scheme of Sadakane [14] we propose a three-component structure to implement a new CST tailored to highly repetitive inputs.



(a) Case 1: prefix query, starting at the top right node



(b) Case 2: suffix query, starting at the top right node

Figure 3: A schematic example of a bad instance for our *fwd-search* algorithm. Case 1 corresponds to a search covering a prefix of a block, while case 2 corresponds to a search covering a suffix of a block. Each of those cases is transformed into one instance of each case in the next level of the tree, generating an exponential blowup in the levels (thus linear in the number of nodes of the tree) by repeating the same argument. Note that, for this to happen, we first need to visit a BackBlock, transforming the query into a substring that it is not a prefix nor a suffix, and then we need to visit another BackBlock, which separates the query again into the corresponding cases.

– We represent the tree topology using BP, and use our BT-CT structure of Section 4 to exploit its repetitiveness.

– For the LCP, we use the compressed version of the bitvector H [16].

– We use the RLCSA [32] as our CSA, and replace its sampling by a grammar- or Block-Tree-compressed representation of the suffix array and its inverse.

We call Block-Tree CST (BT-CST) the resulting compressed suffix tree. Our choice for accessing A and A^{-1} give rise to variants we call BT-CST-X-Y, where X refers to the implementation of A and Y to A^{-1}. Their values are *NONE* if we retain the sampling mechanism of the RLCSA, *LCSA* if we use grammar compression, and *DABT* if we use Block-Tree compression, as detailed next. The variant BT-CST-NONE-NONE is called just BT-CST.

5.1 Enhanced RLCSA

Recall from Section 2.2 that the RLCSA uses $O(r \lg n)$ bits of space and can support access to the suffix array A with the help of a regular sampling of $O((n/s) \lg n)$ bits, where s is the sampling rate. If $A[i]$ is queried, we iteratively apply Ψ until we get that $\Psi^k[i]$ is a sampled position, and return the sample minus k. Access to the inverse array A^{-1} is solved in a similar way, using a sampling on A^{-1} and applying Ψ to reach the requested position. Finally, the iterated Ψ function,

Algorithm 5 Computing *min-excess*(P, i, j) on the Block Tree of P, invoked as *min-excess*$(root, i, j, |P|)$ with global variable $e \leftarrow 0$.

```
Function min-excess($v, i, j, b$)
if $j - i = b$ then
    $e \leftarrow e + v.excess$
    return $v.mexcess$
$m \leftarrow 1$
if $v$ is a LeafBlock then
    $e' \leftarrow e$
    for $k \leftarrow i$ to $j$ do
        if $v.blk[k] = 0$ then $e \leftarrow e - 1$ ;
        else $e \leftarrow e + 1$ ;
        if $e - e' < m$ then $m \leftarrow e - e'$;
if $v$ is a BackBlock then
    if $v.off + i \leq b$ then
        if $i = 1$ and $j \geq b - v.off$ then
            $e \leftarrow e + v.fb\text{-}excess$
            $m \leftarrow v.fb\text{-}mexcess$
        else
            $m \leftarrow min\text{-}excess(v.ptr, v.off + i, \min(v.off + j, b), b)$
    if $v.off + j > b$ then
        if $j = b$ and $i \leq b - v.off$ then
            $e \leftarrow e + v.sb\text{-}excess$
            $m' \leftarrow v.sb\text{-}mexcess$
        else
            $m' \leftarrow min\text{-}excess(v.ptr.next, \max(v.off + i - b, 1), v.off + j - b, b)$
        $m \leftarrow \min(m, v.fb\text{-}excess + m')$
if $v$ is an InternalBlock then
    $k_1 \leftarrow \lfloor (i-1)/\kappa \rfloor$; $k_2 \leftarrow \lfloor (j-1)/\kappa \rfloor$
    $e' \leftarrow e$
    $p \leftarrow (k_1 - 1) \cdot (b/\kappa)$
    for $k \leftarrow k_1$ to $k_2$ do
        $m' \leftarrow min\text{-}excess(v.child[k], \max(i, p+1) - p, \min(j, p + b/\kappa) - p, b/\kappa)$
        if $m' + (e - e') < m$ then $m \leftarrow m' + (e - e')$;
        $p \leftarrow p + b/\kappa$
return $m$
```

Ψ^d, required by the operation *letter*, is answered by applying Ψ d times, unless d is greater than $2s$, in which case the operation is solved by using A and A^{-1}, $\Psi^d[i] = A^{-1}[A[i] + d]$.

The samplings in the RLCSA have been very hard to compress [32], so we try replacing these samplings by compressed encodings of A and A^{-1}. Since their differential encodings DA and DA^{-1} inherit the repetitiveness of its input [16] we use structures based on grammar compression and Block Trees. For the grammar compression we make improvements on the *Locally Compressed Suffix Array* (LCSA) [39]. We note that this is a general representation of differential encodings (not specific to A), and we add fields to speed up its access queries. For Block Trees we show how to adapt the *rank* query for answering *access* to the original array in a differential encoding of it.

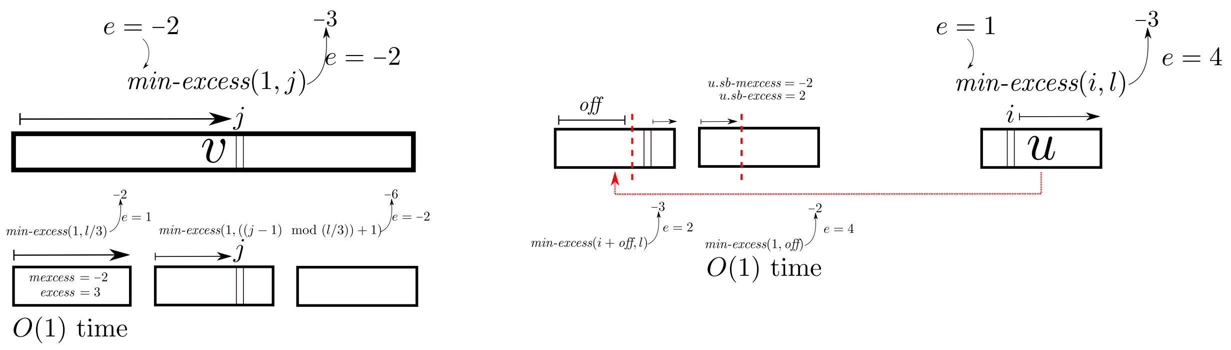

$e = -2$
-3
$e = -2$
$min\text{-}excess(1, j)$
j
v
-2
$e = 1$
$min\text{-}excess(1, l/3)$
$min\text{-}excess(1, ((j-1) \bmod (l/3)) + 1)$
-6
$e = -2$
j
$mexcess = -2$
$excess = 3$
$O(1)$ time
off
$u.sb\text{-}mexcess = -2$
$u.sb\text{-}excess = 2$
$e = 1$
-3
$e = 4$
$min\text{-}excess(i, l)$
i
u
-3
$e = 2$
$min\text{-}excess(i + off, l)$
-2
$e = 4$
$min\text{-}excess(1, off)$
$O(1)$ time

Figure 4: On the left, *min-excess* is queried on a prefix of an InternalBlock v with a cumulative excess of $e = -2$. The query is translated to the first child of v, which is completely covered, returning -2 instantly and changing the cumulative excess to $e = 1$ (using the fields *mexcess* and *excess*). Then the query continues to a prefix of the second child of v, where it returns -6 and changes the cumulative excess to $e = -2$. Finally, the minimum from the second child is chosen and readjusted to -3. On the right, *min-excess* is queried on a suffix of a BackBlock u with a cumulative excess of $e = 1$. The query is translated to a suffix of the first pointed block, where it returns -3 and changes the cumulative excess to $e = 2$. Then the query continues to the prefix of the second block corresponding to the second part of $u.blk$, where it instantly returns -2 and updates the cumulative excess to $e = 4$ (using the fields *sb-mexcess* and *sb-excess*). Finally, the minimum from the first pointed block is chosen and returned.

5.1.1 LCSA optimizations

The *Locally Compressed Suffix Array* (LCSA) [39] is a grammar-compressed representation of the suffix array A, where Re-Pair [40] is used to compress the differential encoding DA and a sampling of A with sampling rate r, $A'[1, \lceil n/r \rceil]$, is added on top. To recover a particular cell $A[i]$ they take the nearest sample to the left and add all the symbols between the sample and i, of the compressed differential representation DA, because $A[i] = A[s] + \sum_{j=s+1}^{i} DA[j]$. They also add a bitvector $L[1, n]$ indicating the positions in A where each symbol of the initial rule $S \to C[1] \cdots C[c]$ starts.

More formally, the process to recover a particular cell $A[i]$ of the suffix array is as follows:

1. Identify the nearest sample to the left of i: $A'[k = \lfloor (i-1)/r \rfloor + 1] = A[s = (k-1)r + 1]$.
2. Decompress $DA[s+1], \ldots, DA[i]$ from the grammar and add it to $A[s]$:
   a. Identify the symbols $C[x]$ and $C[y]$ containing $DA[s+1]$ and $DA[i]$, respectively. For this, *rank* on L is used.
   b. Expand $C[x], \ldots, C[y]$ and get the desired values from this expansion.

The authors note that, if $x \neq y$, it is not necessary to completely expand $C[y]$, because they do a left-to-right expansion of rules. That this, for the Re-Pair rules $A \to BC$ they first expand B and then C, so they do not expand C in case $DA[i]$ was already expanded by B. We note we could apply the same optimization on $C[x]$ if we do a right-to-left expansion. What we call LCSA in this paper already includes this optimization.

Still, it could happen that the sample and i fall in the same symbol $C[x = y]$, and in this case none of the optimizations apply. To handle this case, we propose our second improvement, which consists in attaching the length of the expansions at every rule. With these lengths we can decide beforehand when an expansion is necessary. We call LCSA-*lengths* this optimization.

Algorithm 6 Computing *daccess*(DA, i) on the Block Tree of P, invoked as *daccess*$(root, i, |P|)$.

```
Function daccess(v, i, b)
if $v$ is a LeafBlock then return the sum of the symbols in $v.blk[1, i]$ (by brute force) ;
if $v$ is a BackBlock then
    if $v.off + i \leq b$ then return $daccess(v.ptr, i + v.off, b) - v.pfb\text{-}sum$ ;
    else return $daccess(v.ptr.next, i + v.off - b, b) + v.fb\text{-}sum$ ;
$d \leftarrow \lfloor (i-1)/\kappa \rfloor$
$p \leftarrow 0$
for $k \leftarrow 1$ to $d$ do $p \leftarrow p + v.child[k].partial\text{-}sum$ ;
return $p + daccess(v.child[d+1], ((i-1) \bmod \kappa) + 1, b/\kappa)$
```

Note that, although these solutions were built to represent suffix arrays and their inverses, they can represent any array A whose consecutive differences are repetitive.

5.1.2 Block Tree adaptation (DABT)

We use Block Trees to compress the differential encoding of an array A and change the node fields used to answer *rank* by fields storing the sum of the sequence of differences they represent.

More formally, we compress the differential encoding of an array A with a Block Tree and in the nodes v of the Block Tree we replace:

– $v.rank_c$ by $v.partial\text{-}sum$, which is the sum of the symbols of $v.blk$.

– $v.fb\text{-}rank_c$ by $v.fb\text{-}sum$, which is the sum of the symbols in the maximal suffix of $v.ptr.blk$ overlapping with the first occurrence of $v.blk$.

– $v.pfb\text{-}rank_c$ by $v.pfb\text{-}sum$, which is the sum of the symbols in the maximal prefix of $v.ptr.blk$ not overlapping with the first occurrence of $v.blk$.

With these changes, an *access* query in the Block Tree corresponds to *access* on the differential array DA, and Algorithm 6, using these new fields, corresponds to an *access* to the original array A. This access takes time $O(\kappa \lg_\kappa(n/ll) + ll)$.

We call this adaptation *DABT* (Differential Array Block Tree).

6 Experiments and Results

We measured the time/space performance of our compressed topology BT-CT (Section 4), different strategies for replacing the sampling in the RLCSA (Section 5.1) of our new BT-CST (Section 5), and compared them with the state of the art. Our code is publicly available at `https://github.com/elarielcl/MinimalistBT-CST`.

6.1 Setup and Datasets

Our 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++ and use the *sdsl* library [49] for their internal components. The compiler is `gcc 6.3.0`, with `-O9` optimization.

Dataset	n	σ	Type	`p7zip` (bps)
dna0.001	106	5	Synthetic DNA	0.041
dna0.01	106	5	Synthetic DNA	0.044
dna0.1	106	5	Synthetic DNA	0.061
dna1.0	106	5	Synthetic DNA	0.188
`influenza`	155	15	Real DNA	0.135
`escherichia`	112	15	Real DNA	0.621
`einstein`	93	117	English text	0.009
`kernel`	259	160	Program code	0.205

Table 2: Summary table of the repetitive datasets. Length n is measured in millions (and rounded). Compression by `p7zip` is shown in bits per symbol (bps).

In our experiments we use repetitive datasets obtained from the Repetitive Corpus of the *Pizza&Chili* platform[1]. We artificially create repetitive dna sequences: *dna0.001*, *dna0.01*, *dna0.1*, and *dna1.0*, where *dnap* is built taking a 1MB prefix of the dna sequence in the corpus, copying it 100 times, and mutating each copied base with probability $p/100$. The 1MB prefix of dna used as base for the construction was obtained from the Gutenberg Project[2].

We also use two real DNA sequences: `influenza`, a collection composed of 78,041 sequences of *Haemophilus Influenzae*; and `escherichia`, a collection of sequences of different *Escherichia Coli* individuals. Both sequences come from the NCBI[3]. Besides, we use two non-DNA sequences, `einstein`, containing all the versions (up to January 12, 2010) of the German Wikipedia Article of *Albert Einstein*; and `kernel`, a set of 36 versions of the Linux Kernel. Table 2 gives their main characteristics, including an estimation of their repetitiveness through the Lempel-Ziv compression program `p7zip`[4]

For each of those collections, we build its suffix tree and represent its topology with parentheses (BP, see Section 2.1).

6.2 Compressed Topologies

6.2.1 Structures and Tests

We compare the following structures:

SADA. The classical BP implementation using the rmM-Tree [30]. We use the implementation of the *sdsl*, `cst_sada`, with its default configuration. We only consider the part of the implementation dedicated to answer tree topology operations.

GCT. The Grammar-Compressed Topology used as part of the Grammar-Compressed Suffix Tree [26]. We use the same implementation used in its original publication, but only considering the topology space. We vary parameters *rule-sampling* and *C-sampling* as the authors suggest.

[1]`http://pizzachili.dcc.uchile.cl/repcorpus`
[2]`http://www.gutenberg.org/`
[3]`https://www.ncbi.nlm.nih.gov/`
[4]`http://p7zip.sourceforge.net/`

BT-CT. Our Block Tree based topology. We vary the parameters $\kappa \in \{2, 4, 8\}$ (arity) and $ll \in \{16, 32, 64, 128, 256, 512\}$ (leaf length).

We run the queries *first-child*, *tree-depth*, *next-sibling*, *parent*, *level-ancestor* and *lca*. Data points are the average of 100,000 random queries, doing as in previous work on Compressed Suffix Trees [17, 26] to choose the nodes on which the operations are called: For *first-child*, *tree-depth*, *next-sibling* and *parent* we collect the nodes in leaf-to-root paths starting from random leaves. For *level-ancestor* we choose random leaves v whose $\mathit{tree\text{-}depth}(v) = td \geq 10$, and choose a random $d \in [1, td-1]$. For *lca* we choose random leaf pairs. We show only the Pareto-optimal results of each structure.

6.2.2 Results and Discussion

In the Appendix we show a graph for each operation on each input, the x-coordinate representing the space in bps (bits per symbol) and the y-coordinate the time per operation in microseconds; the input BP and the operation are indicated in the title.

Our experiments show that BT-CT is, in general, one order of magnitude faster than GCT, and similar to SADA. The latter uses about 1.4 bps independently of the repetitiveness of its input. GCT is the smallest structure, using from half to an order of magnitude less space than BT-CT depending on the input. BT-CT, in turn, uses from a half to an order of magnitude less space than SADA, depending on repetitiveness. For example, for *einstein* GCT uses 0.03–0.35 bps and BT-CT uses 0.11–0.25 bps.

Figures 7 and 8 show the operations *first-child* and *tree-depth*, which use the basic *access* and *rank* primitives. SADA solves these primitives using techniques for plain bitvectors, and thus it is an order of magnitude faster than BT-CT (which solves them on the Block Tree) and two orders faster than GCT (which solves them on grammars).

Figures 9, 10 and 11 show the operations *next-sibling*, *parent* and *level-ancestor*, which are solved by using the primitives *fwd-search* and *bwd-search*. Those are solved by SADA using the so-called rmM-tree, whose implementation offers logarithmic times. For these operations BT-CT obtains times very similar to SADA, both being an order of magnitude faster than GCT. That is, even when we could not offer good theoretical bounds for these primitives, BT-CT works in time similar to the rmM-tree while using repetitiveness-aware space.

Figure 12 shows the operation *lca*, which uses the primitives *min-excess*, *bwd-search* and *fwd-search*. In this case, the BT-CT obtains time very similar to SADA again, but this time GCT gets closer (less than one order of magnitude of difference) on its fastest versions.

To summarize, BT-CT offers a new topology representation, with times competitive with SADA, the fastest known compressed representation, while using space that decreases significantly with higher repetitiveness. GCT is able to use considerably less space, but it is typically an order of magnitude slower than BT-CT.

6.3 Differential Arrays

6.3.1 Structures and Tests

We run *access* experiments on the suffix array A and its inverse A^{-1}. Our experiments build the corresponding arrays for each of the inputs and average 100,000 random *access* queries on each of them. We compare the following structures:

LCSA, LCSA-lengths. Our adaptations of LCSA [39]. We vary $\mathit{sampling\text{-}rate} \in \{16, 32, 64, 128, 256, 512, 1024, 2048\}$, for the sampling of the absolute values.

Table 3: Recommended structures depending on the accessed array and the repetitiveness of the input sequence.

	Repetitive	Highly repetitive
A	RLCSA	LCSA-lengths
A^{-1}	RLCSA	DABT

DABT. Our adaptation of Block Trees to access differential arrays. We vary the parameters $\kappa \in \{2, 4, 8\}$ (arity) and $ll \in \{4, 8, 16, 32, 64, 128\}$ (leaf length).

RLCSA. The same RLCSA [32] implementation used by Ordóñez el al. [26]; *access* to A and A^{-1} is done with the help of a sampling on A. We vary its parameters *sa-sampling* $\in \{32, 64, 128, 256\}$ and *block-size* $\in \{16, 32, 64\}$.

We only show the Pareto-optimal results of each structure.

6.3.2 Results and Discussion

The Appendix shows the time/space tradeoffs obtained for accessing A and A^{-1}.

Figure 13 shows that DABT is the fastest alternative, but it is also the largest, never using less than 5 bps. RLCSA is the structure achieving the least space, except for the most repetitive inputs, where the sampling component is a barrier for further reducing space. LCSA can be up to two orders of magnitude slower than LCSA-lengths, which shows that the length field we added handles the bad cases of traditional LCSA. In general, LCSA-lengths can be nearly as fast as DABT while using considerably less space, and up to an order of magnitude faster than RLCSA. It is not clear why the Block-Tree-based DABT representation of the differential suffix array is considerably larger than the corresponding grammar-based representation.

Figure 14 shows that DABT uses much less space for representing the differential array A^{-1} than for A (the array A^{-1} is known to preserve the repetitiveness of the input better than A [24]). Since the LCSA variants and RLCSA use a space similar as for A, DABT becomes an interesting variant for highly repetitive inputs, where it is of comparable size and 1–2 orders of magnitude faster than the RLCSA and LCSA-lengths (LCSA, in turn, is considerably slower). When repetitiveness is lower, DABT becomes again too large to be of interest, and RLCSA becomes the best option.

Table 3 summarizes the recommended structures depending on the accessed array and the repetitiveness of the input sequence.

6.4 Compressed Suffix Trees

We now combine the previous structure to build various variants of our structure, BT-CST, and compare it with previous work.

6.4.1 Structures and Tests

We compare the following structures:

CST_SADA, CST_SCT3, CST_FULLY. Adaptation and improvements from the *sdsl* library on the indexes of Sadakane [14], Fischer et al. [16] and Russo et al. [18], respectively. CST_SADA maximizes speed using Sadakane's CSA [35] 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 [50] 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 [32] and use a run-length-compressed representation of bitvector H [16].

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

BT-CST-{LCSA, DABT, NONE}-{LCSA, DABT, NONE}. Our new Compressed Suffix Tree with the described components. For the BT-CT component we vary $\kappa \in \{2, 4, 8\}$ (arity) and $ll \in \{4, 8, 16, 32, 64, 128, 256\}$ (leaf length). For the versions NONE, the RLCSA uses a sampling of $s = 128$ for both A and A^{-1}. For the versions LCSA we use LCSA-lengths to get better time performance and set the sweet point *sampling-rate* = 128. For DABT, we use a low-space configuration, that is, $\kappa = 2$ and $ll = 16$. We only present the results for BT-CST-NONE-NONE (as BT-CST), BT-CST-LCSA-NONE (as BT-CST-LCSA), BT-CST-LCSA-LCSA, and BT-CST-LCSA-DABT, since the others are pretty similar or dominated by these versions.

For all 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. We do not include the CST of Abeliuk et al. [17] in the comparison because it was already outperformed by several orders of magnitude by GCST.

In addition to the six topology operations tested in Section 6.2 (which some CSTs simulate without storing the tree topology at all), we study four specific suffix tree operations: *suffix-link*, *string-depth*, *string-ancestor* and *child*. On those, data points are also averaged over 100,000 random queries, following the scheme used in previous work on Compressed Suffix Trees [17, 26] to choose the nodes on which the operations are called. For *suffix-link* we collect the nodes on traversal starting from random leaves, and taking suffix-links until reaching the root. For *string-ancestor* we choose random leaves v whose *string-depth*$(v) = sd \geq 10$, and choose a random $d \in [1, sd - 1]$. 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.

Maximal Exact Matches (MEMs) We also compare the suffix trees in solving a typical bioinformatic problem. We test all of the above implementations except CST_FULLY, because of its poor time performance.

The MEM problem is as follows. Given a pattern string $S[1, m]$, we want to find all its maximal substrings appearing in T, where maximal means that, if we extend them left or right by a single symbol, they do not appear anymore in T. This problem can be solved in $O(m)$ time using the suffix tree of T to find the requested substrings. We implement the same algorithm as in previous work [26]. The algorithm maintains two pointers, which indicate the limits of the current substring of S, $S[i, j]$, and a current suffix tree node. Initially we have $i = 1$, $j = 0$, and the root node. The algorithm works by iteratively applying the following two steps:

1. Try to increase j by descending in the suffix tree by the letter $S[j + 1]$. The algorithm descends from the current node as much as possible and then outputs the corresponding maximal substring, $S[i, j]$, if $i < j$.

2. Increase i by the minimum necessary amount. The algorithm takes successive suffix links from the current node until it can descend again from it by $S[j + 1]$. If $i > j$ during this process, it sets $j = i - 1$.

This iteration of two steps is repeated until j reaches m, the end of S.

We use the same setup of the GCST publication [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 BT-CT with $\kappa = 2$ and $ll = 128$ and GCST uses *rule-sampling* $= 1$ and *C-sampling* $= 2^{10}$. The tradeoffs refer to *sa-sampling* $\in \{64, 128, 256\}$ for the RLCSAs. Data points are the average of 100 executions of the algorithm.

Time and peak memory in construction We also measure the time and peak memory usage for the constructions on all of the above implementations, except for CST_FULLY, on the same sequence `influenza` and parameters used for the MEMs problem. Data points are the average of 10 executions of the constructions.

6.5 Results and Analysis

The Appendix shows the plots for the 10 suffix tree operations. The smallest structure, by a wide margin, is always GCST, except on *dna1.0* and `escherichia` (two of the least repetitive sequences), where CST_FULLY is smaller. The next smallest indexes are BT-CST, CST_FULLY and, on the less repetitive sequences, CST_SCT3_RLCSA. The compressed indexes not designed for repetitive collections use 1–4 bps less if combined with a RLCSA. The variant BT-CST-LCSA is larger than BT-CST, and in turn smaller than BT-CST-LCSA-DABT.

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).

In operations *first-child* and *tree-depth* (Figures 15 and 16), which use the basic *access* and *rank* primitives, CST_SADA[_RLCSA] use techniques of plain bitvectors, which yields an order of magnitude of advantage over BT-CST and two orders over GCST. CST_SCT3[_RLCSA] and CST_FULLY are orders of magnitude slower in these operations, because they do not explicitly store the topology.

In operations *next-sibling*, *parent* and *level-ancestor* (Figures 17, 18, and 19), which rely more 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. Interestingly, the larger representations, including those where the tree topology is represented using 2.8 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 the tree topology explicitly.

Operation *lca* (Figure 20), 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 2.8 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 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.

Note that for tree topology operations (*first-child*, *tree-depth*, *next-sibling*, *parent*, *level-ancestor*, and *lca*), the times of our BT-CST variants are the same, because they differ only in their underlying CSA, which is not used for those operations. The same occurs between CST_SADA and CST_SADA_RLCSA, and between CST_SCT3 and CST_SCT3_RLCSA.

Operation *suffix-link* (Figure 21) involves primitive *min-excess* and several others on the topology, but also the operation Ψ on the corresponding CSA. Since the latter is relatively fast on the RLCSA, all the BT-CST variants take nearly 10 μsec, whereas the additional operations on the topology drive GCST over 100 μsec, and CST_FULLY over the millisecond. This time the topology representations that are blind to repetitiveness 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.

Operations *string-depth* and *string ancestor* (Figures 22 and 23) are solved by combinations of topology operations and access to the suffix array A, being the latter the costliest for the indexes using the RLCSA. For this reason, the time difference between BT-CST and GCST is reduced in these operations. The impact in the use of A is also shown in the comparison between CST_SADA and CST_SADA_RLCSA, and between CST_SCT3 and CST_SCT3_RLCSA. CST_FULLY is an order of magnitude slower in these operations. Our variants using the LCSA encoding of A yield about an order of magnitude time improvement over the plain BT-CST, which uses the slower RLCSA sampling.

Finally, operation *child* (Figure 24) is the most expensive, requiring one application of *string-depth* and several of *next-sibling* and *letter*, thereby heavily relying on the CSA. CST_SCT3[_RLCSA] binary search the children; the others scan them linearly. The indexes using a CSA that adapts to repetitiveness require nearly one millisecond on large alphabets, whereas those using a larger and faster CSA are up to 10 (CST_SCT3) and 100 (CST_SADA) times faster. 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 near the millisecond. Our variants of BT-CST behave differently depending on the input sequence: on *dna* texts, the major time improvement (about one order of magnitude) is given by the presence of a differential encoding of A, because in this case the costliest operation is the *string-depth* done at the beginning. However, for inputs with large alphabets, such as *einstein* and *kernel*, the presence of a differential encoding of A^{-1} is more important (one order of magnitude improvement), as in this case nodes have more children, and then more applications of *letter* are required. Although we recommended the DABT over LCSA for A^{-1}, the difference in time or space between BT-CST-LCSA-LCSA and BT-CST-LCSA-DABT is very small. The former is likely to be preferable in general because it behaves better when the repetitiveness is not so high.

Maximal Exact Matches Figure 5 shows the results for the MEMs problem, giving the time per symbol of S. Our basic version, 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 (GCST, 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, just as its CSA for solving the dominant operation *child*, but also about 3 times faster. Our BT-CST variants using a differential encoding of A present significant time improvements, yet at the cost of considerably more space. Other indexes are never Pareto-optimal.

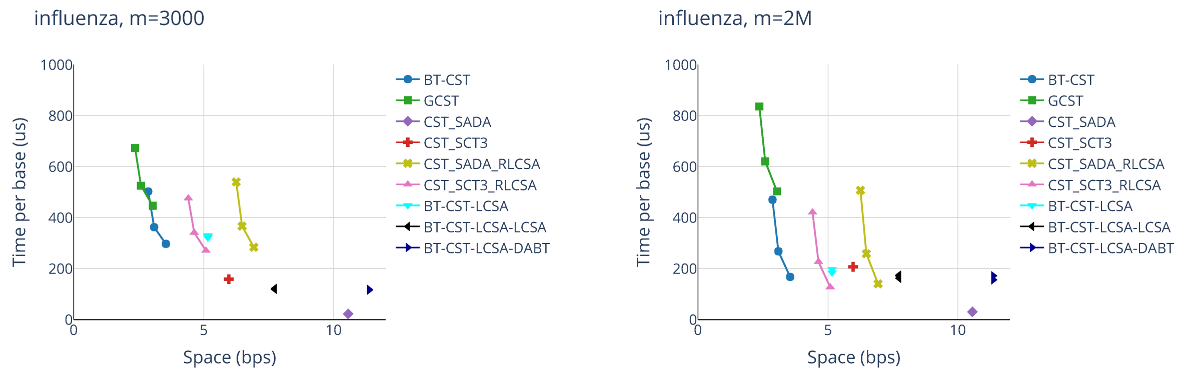


Figure 5: Performance of CSTs when solving the MEMs problem. The y-axis is time in microseconds per base in the smaller sequence (of length m).

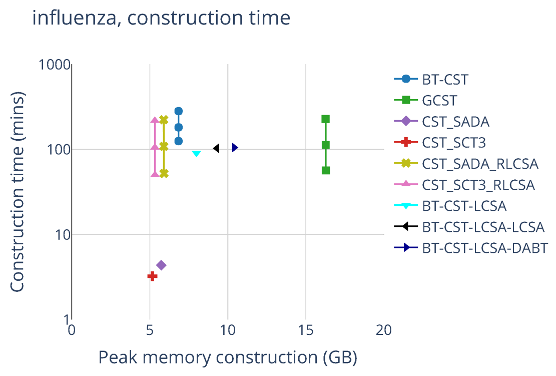


Figure 6: Time and peak memory of CST constructions. The y-axis is time in minutes in log-scale. The x-axis is the peak memory consumption in gigabytes.

Construction Figure 6 shows the behavior of the construction algorithms both in time and peak memory consumption. The CSTs blind to repetitiveness (CST_SADA and CST_SCT3) are the fastest (less than 10 minutes) and most space efficient (~ 5 GBs) to build. Their variants using the RLCSA (CST_SADA_RLCSA and CST_SCT3_RLCSA) have the same peak memory, indicating that the space usage of the construction of the RLCSA is less than the other components. However, the construction time of the variants using the RLCSA increases by one order of magnitude (around 100 minutes depending on the *sa-sampling*). In our BT-CSTs, a significant part of the time is used by the construction of the BT-CT (around 60 minutes). The peak memory usage of our BT-CSTs is 1–5 GBs larger than CST_SADA_RLCSA and CST_SCT3_RLCSA because of the construction of the BT-CT, and the space used by DABT and Re-Pair (LCSA) on the differential representations of A and A^{-1}. The peak memory usage of GCST is significantly larger than the other indexes (nearly 16 GBs) because of the space used by Re-Pair to compress the BP topology.

We recommend to use the low-space version of our index, BT-CST, unless we require operations that access the suffix array. In this case, BT-CST-LCSA uses more space but it achieving orders of magnitude improvements on operations involving A. In the case of highly repetitive inputs on large alphabets, like *einstein*, we recommend BT-CST-LCSA-LCSA, which is the fastest compared to its relatives and less sensitive to repetitiveness than BT-CST-LCSA-DABT.

7 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 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 [28] are a novel technique to represent a sequence in space close to its Lempel-Ziv complexity (with a logarithmic-factor penalty), but in a way that direct (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). Our structure dominated a significant part of the space/time tradeoff map when finding Maximal Exact Matches on a DNA repository, a typical problem in Bioinformatics.

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 [30]. This structure may be interesting for other repetitive trees beyond compressed suffix trees, such as XML datasets.

Although we have shown that in practice they perform as well as their classical counterpart [30], 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 [30] 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. [28] claiming logarithmic time for *min-excess* does not really solve the operation.[5]

We also use grammar and Block Tree-based representations of the suffix array and its inverse to enhance the RLCSA and improve the time performance of BT-CST in the operations using its CSA. We obtain improvements of one order of magnitude on these operations. Grammars compress considerably better than Block Trees in this case, unless repetitiveness is very high; we do not have a clear explanation to this observation. In any case, we note that this enhancement of the RLCSA could be applied to any CST using RLCSAs, for example, GCST, CST_SADA_RLCSA or CST_SCT3_RLCSA, which gives a new range of possibilities.

The most relevant challenge ahead for making these repetition-aware CSTs of wide use is to build them within low space and time on huge collections (which take relatively small space once compressed). The CSTs based on prefix-free parsing [46], though not yet achieving compression ratios (and usually, query times) comparable to our BT-CSTs, are an important step in this direction, being the only CSTs for repetitive text collections that can be built on gigabytes of data using reasonable time and main memory. Can we build a significantly smaller CST, for example the BT-CST, on huge collections, possibly making use of the prefix-free-parsing concept?

References

[1] Peter Weiner. Linear pattern matching algorithms. In *Proc. 14th Annual Symposium on Switching and Automata Theory (FOCS)*, pages 1–11, 1973.

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

[5] As checked with coauthor T. Gagie.

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

[4] Alberto Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, pages 85–96. Springer, 1985.

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

[6] 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.

[7] 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.

[8] 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.

[9] Stefan Kurtz. Reducing the space requirement of suffix trees. *Software Practice and Experience*, 29(13):1149–1171, 1999.

[10] David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage. In *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 383–391, 1996.

[11] 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.

[12] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[13] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[14] Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

[15] Enno Ohlebusch, Johannes Fischer, and Simon Gog. CST++. In *Proc. 17th International Conference on String Processing and Information Retrieval (SPIRE)*, pages 322–333, 2010.

[16] Johannes Fischer, Veli Mäkinen, and Gonzalo Navarro. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science*, 410(51):5354–5364, 2009.

[17] Andrés Abeliuk, Rodrigo Cánovas, and Gonzalo Navarro. Practical compressed suffix trees. *Algorithms*, 6(2):319–351, 2013.

[18] 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.

[19] Sarah A. Tishkoff and Kenneth K. Kidd. Implications of biogeography of human populations for 'race' and medicine. *Nature Genetics*, 36:S21–S27, 2004.

[20] Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.

[21] Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.

[22] 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.

[23] Gonzalo Navarro. Indexing highly repetitive collections. In *Proc. 23rd International Workshop on Combinatorial Algorithms (IWOCA)*, pages 274–279, 2012.

[24] T. Gagie, G. Navarro, and N. Prezza. Fully-functional suffix trees and optimal text searching in BWT-runs bounded space. *Journal of the ACM*, 67(1):article 2, 2020.

[25] 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.

[26] Gonzalo Navarro and Alberto Ordóñez. Faster compressed suffix trees for repetitive collections. *ACM Journal of Experimental Algorithmics*, 21(1):1–8, 2016.

[27] Rajeev Raman and S. Srinivasa Rao. Succinct representations of ordinal trees. In *Space-Efficient Data Structures, Streams, and Algorithms*, pages 319–332. Springer, 2013.

[28] Djamal Belazzougui, Travis Gagie, Pawel Gawrychowski, Juha Kärkkäinen, Alberto Ordónez, Simon J Puglisi, and Yasuo Tabei. Queries on LZ-bounded encodings. In *Data Compression Conference (DCC), 2015*, pages 83–92, 2015.

[29] Alberto Ordóñez. *Statistical and repetition-based compressed data structures.* PhD thesis, Universidade da Coruña, 2016.

[30] Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3):16, 2014.

[31] 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.

[32] 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.

[33] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), 2007.

[34] 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.

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

[36] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM (JACM)*, 52(4):552–581, 2005.

[37] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):20, 2007.

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

[39] Rodrigo González, Gonzalo Navarro, and Héctor Ferrada. Locally compressed suffix arrays. *ACM Journal of Experimental Algorithmic*, 19:1.1:1.1–1.1:1.30, 2015.

[40] N. Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.

[41] Simon J. Puglisi and Bella Zhukova. Relative Lempel-Ziv compression of suffix arrays. In *Proc. 27th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 89–96, 2020.

[42] Shanika Kuruppu, Simon J. Puglisi, and Justin Zobel. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In *Proc. 17th International Conference on String Processing and Information Retrieval (SPIRE)*, pages 201–206, 2010.

[43] Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Fully Dynamic Data Structure for LCE Queries in Compressed Space. In *Proc. 41st International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 72:1–72:15, 2016.

[44] Andrea Farruggia, Travis Gagie, Gonzalo Navarro, Simon J. Puglisi, and Jouni Sirén. Relative suffix trees. *The Computer Journal*, 61(5):773–788, 2018.

[45] Djamal Belazzougui, Fabio Cunial, Travis Gagie, Nicola Prezza, and Mathieu Raffinot. Composite repetition-aware data structures. In *Proc. 26th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 26–39, 2015.

[46] Christina Boucher, Ondrej Cvacho, Travis Gagie, Jan Holub Giovanni Manzini, Gonzalo Navarro, and Massimiliano Rossi. PFP compressed suffix trees. In *Proc. 23rd Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2021. To appear.

[47] Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Towards a definitive measure of repetitiveness. In *Proc. 14th Latin American Symposium on Theoretical Informatics (LATIN)*, 2020. To appear.

[48] Manuel Cáceres. *Compressed Suffix Trees for Repetitive Collections Based on Block Trees.* MSc. Thesis, University of Chile, 2019.

[49] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *Proc 13th Symposium on Experimental Algorithms (SEA)*, pages 326–337. Springer, 2014.

[50] 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.

A Time/Space Plots

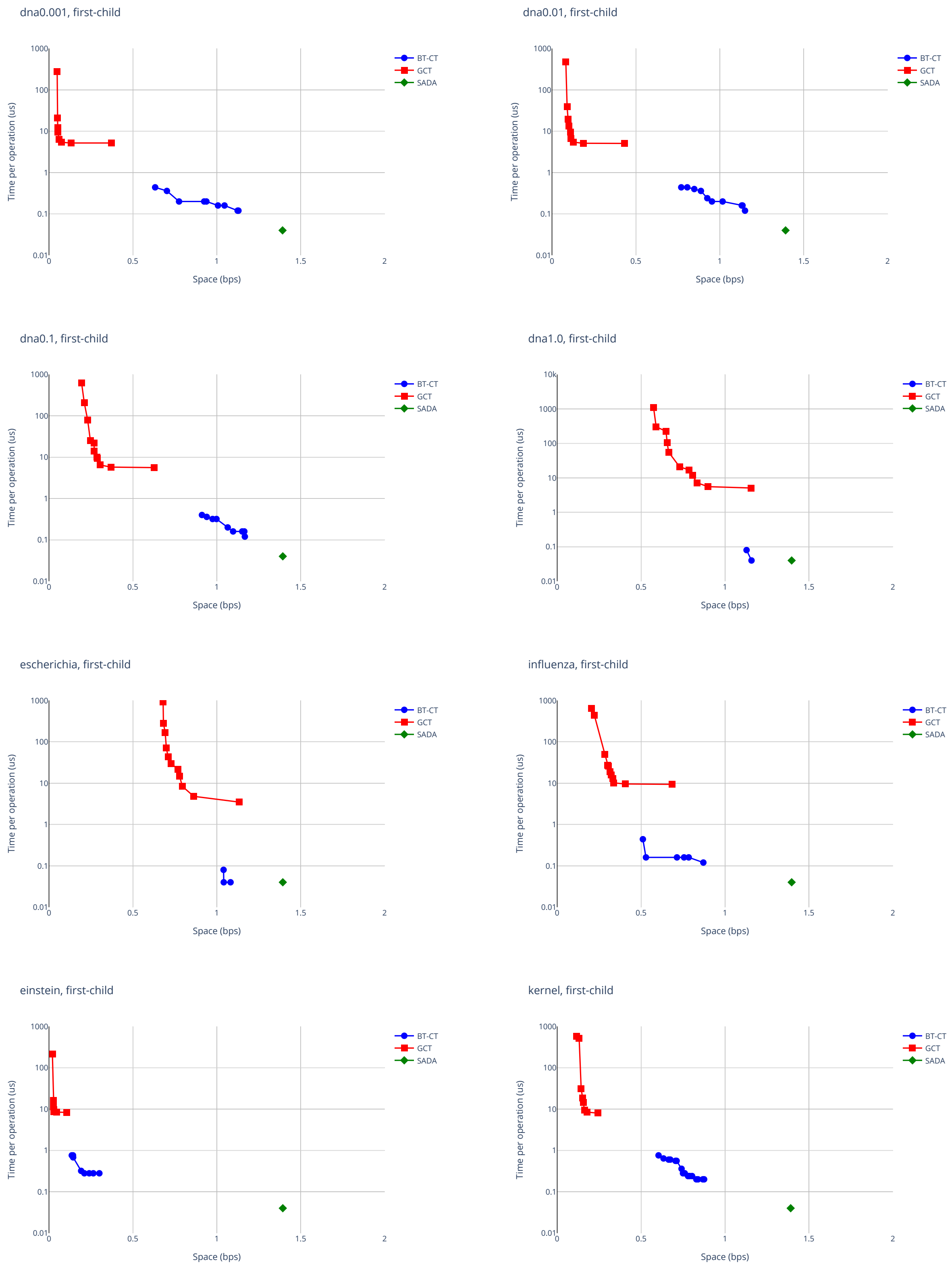


Figure 7: Performance of *first-child* in different BP representations. The y-axis is time in microseconds in log-scale.

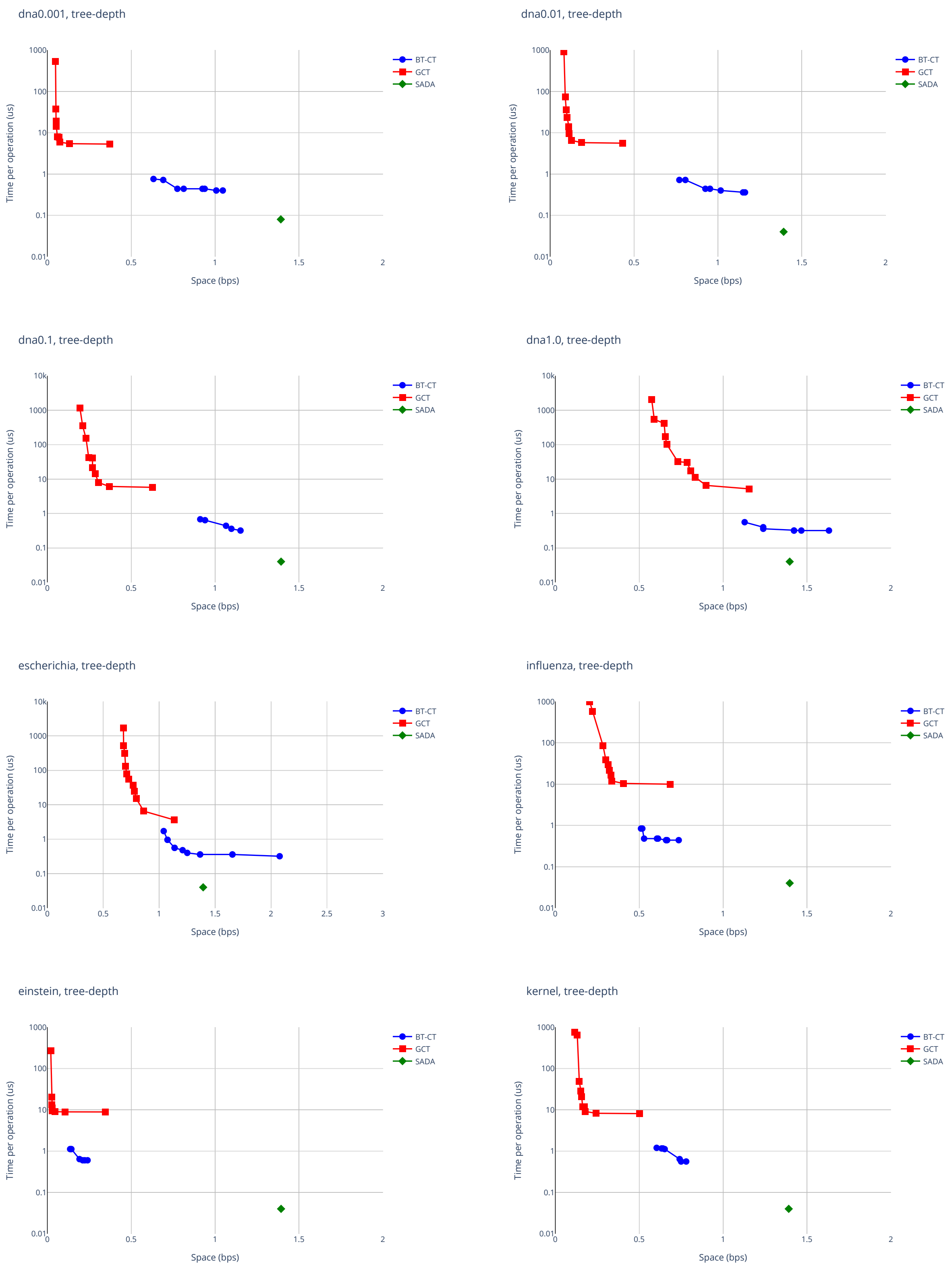


Figure 8: Performance of *tree-depth* in different BP representations. The y-axis is time in microseconds in log-scale.

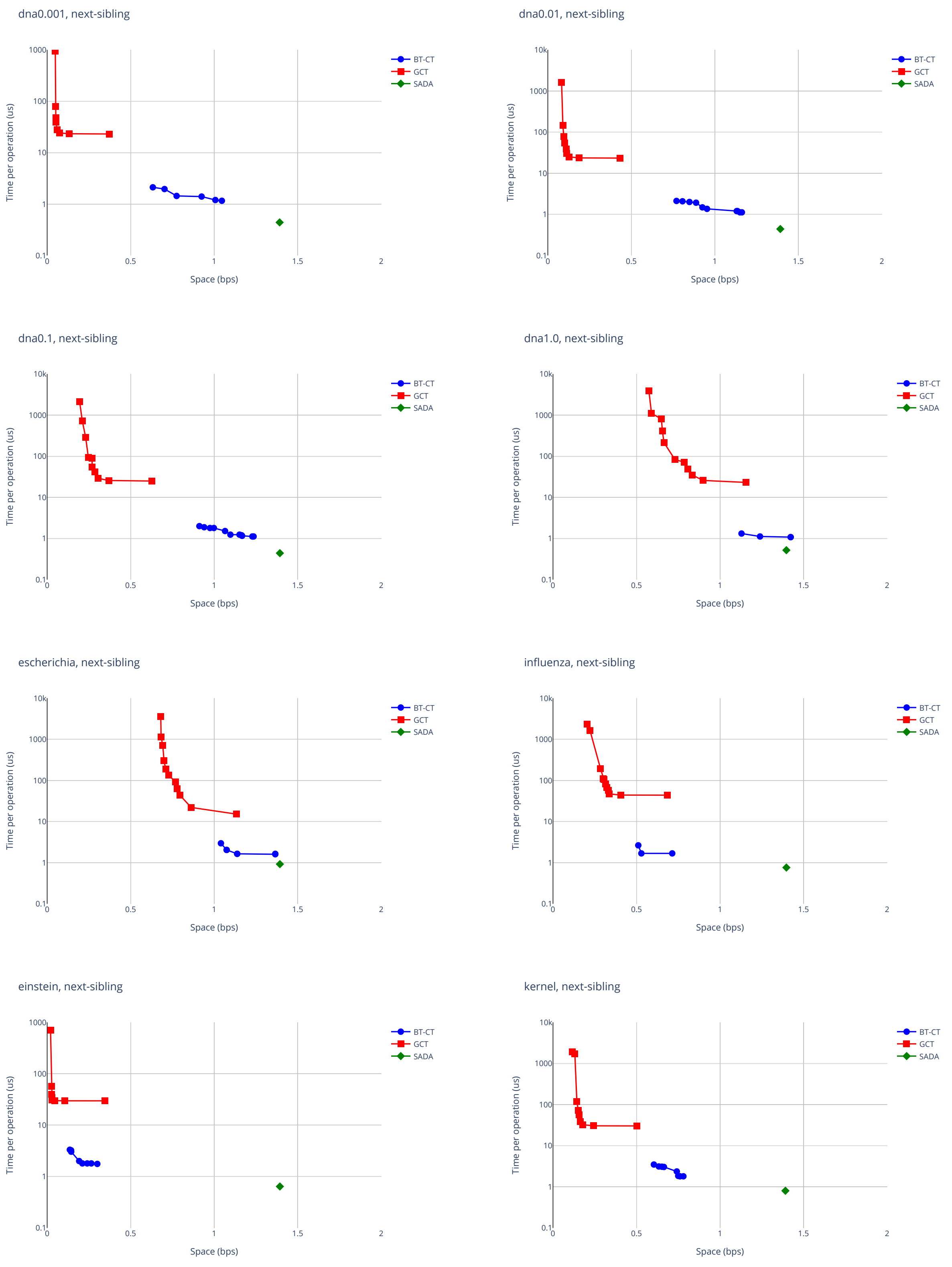


Figure 9: Performance of *next-sibling* in different BP representations. The y-axis is time in microseconds in log-scale.

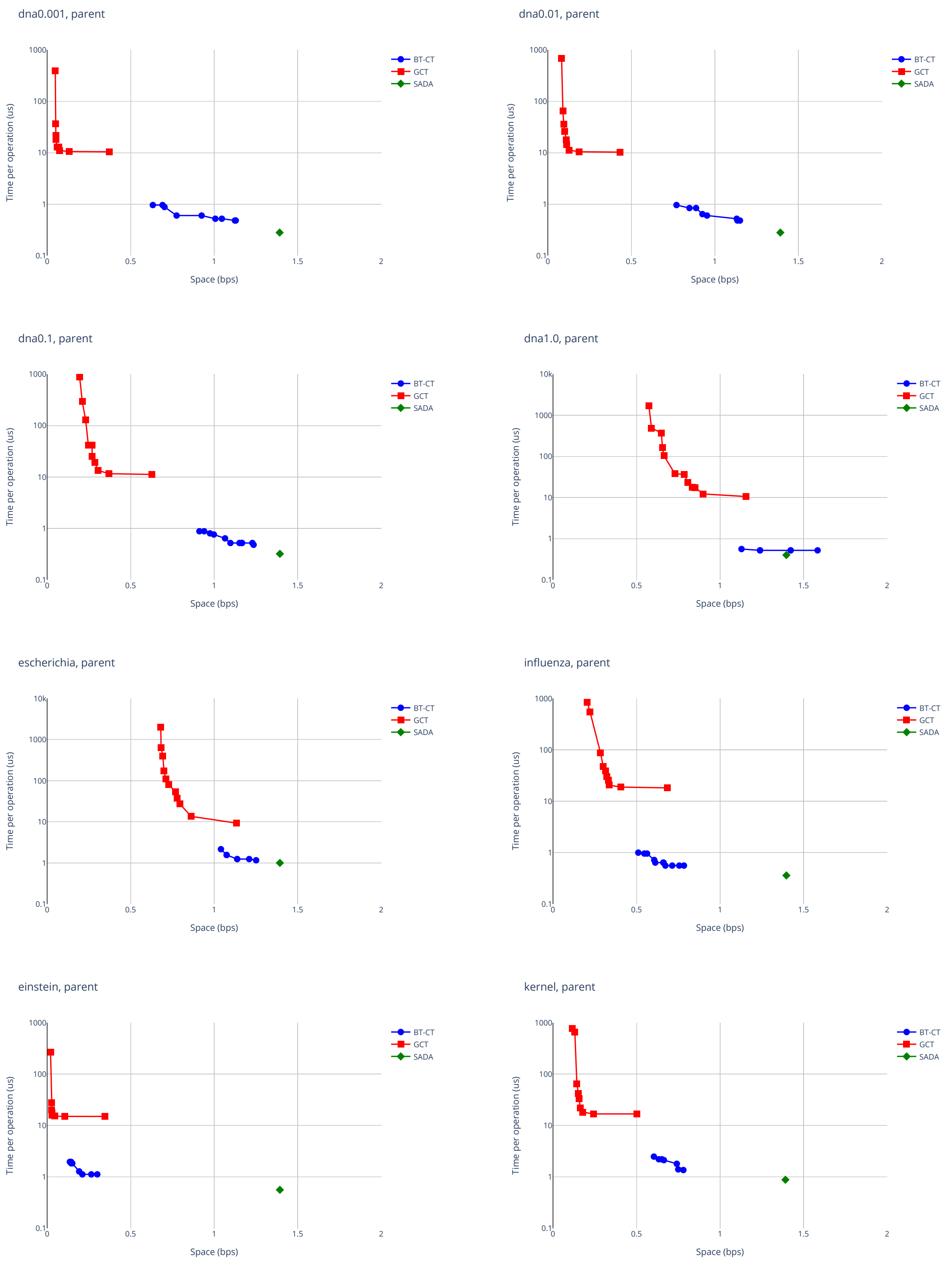


Figure 10: Performance of *parent* in different BP representations. The y-axis is time in microseconds in log-scale.

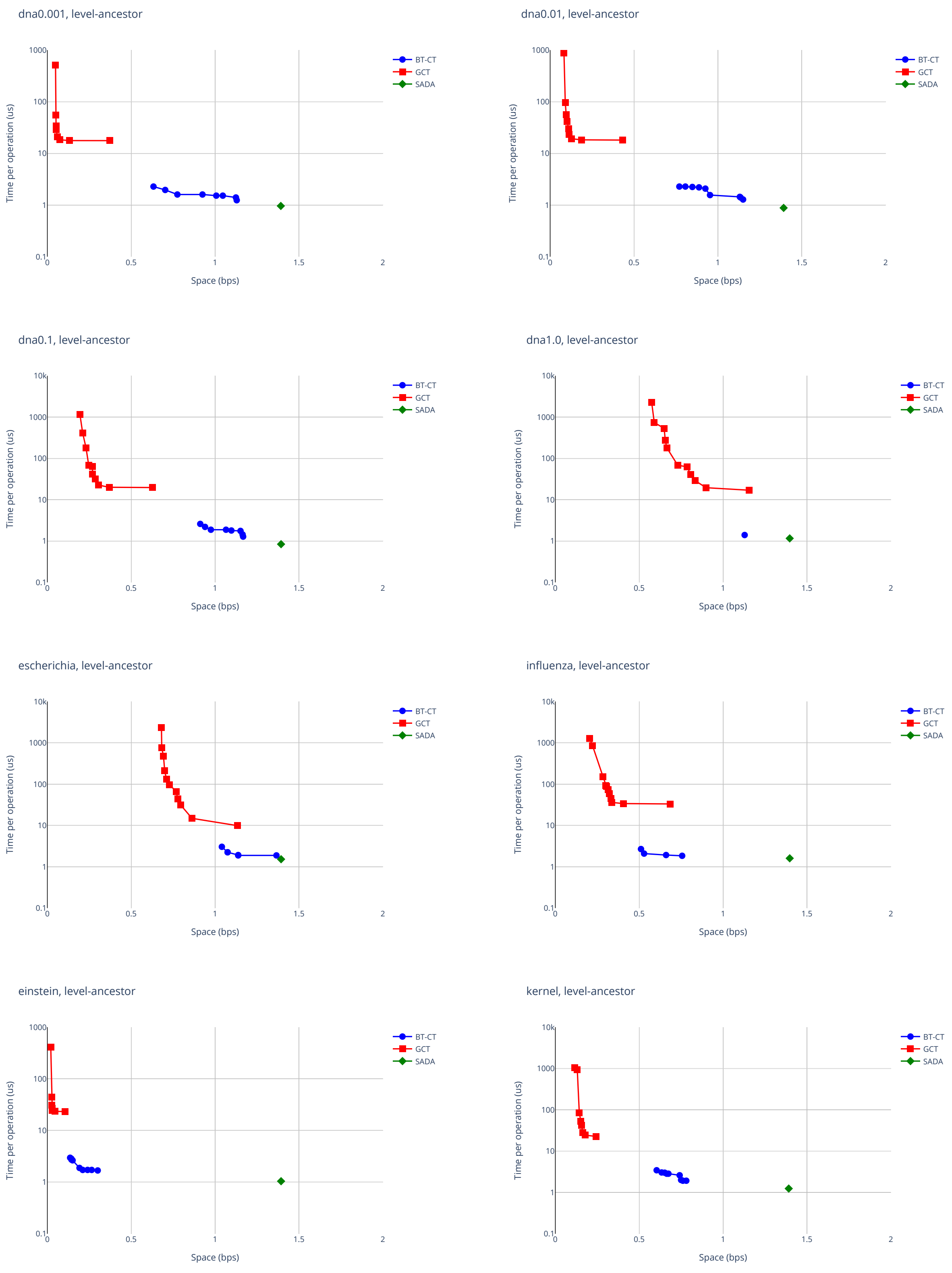


Figure 11: Performance of *level-ancestor* in different BP representations. The y-axis is time in microseconds in log-scale.

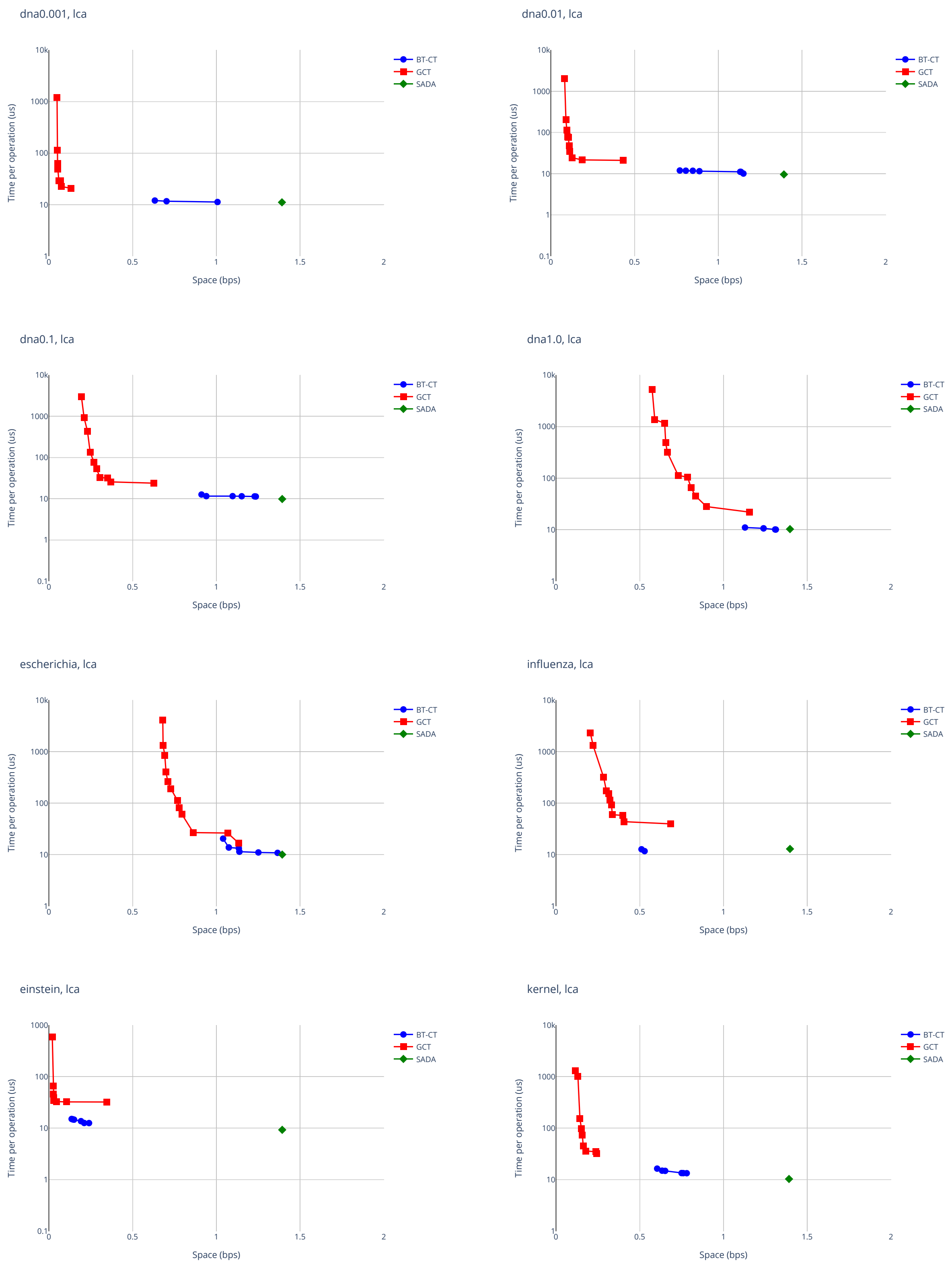


Figure 12: Performance of *lca* in different BP representations. The y-axis is time in microseconds in log-scale.

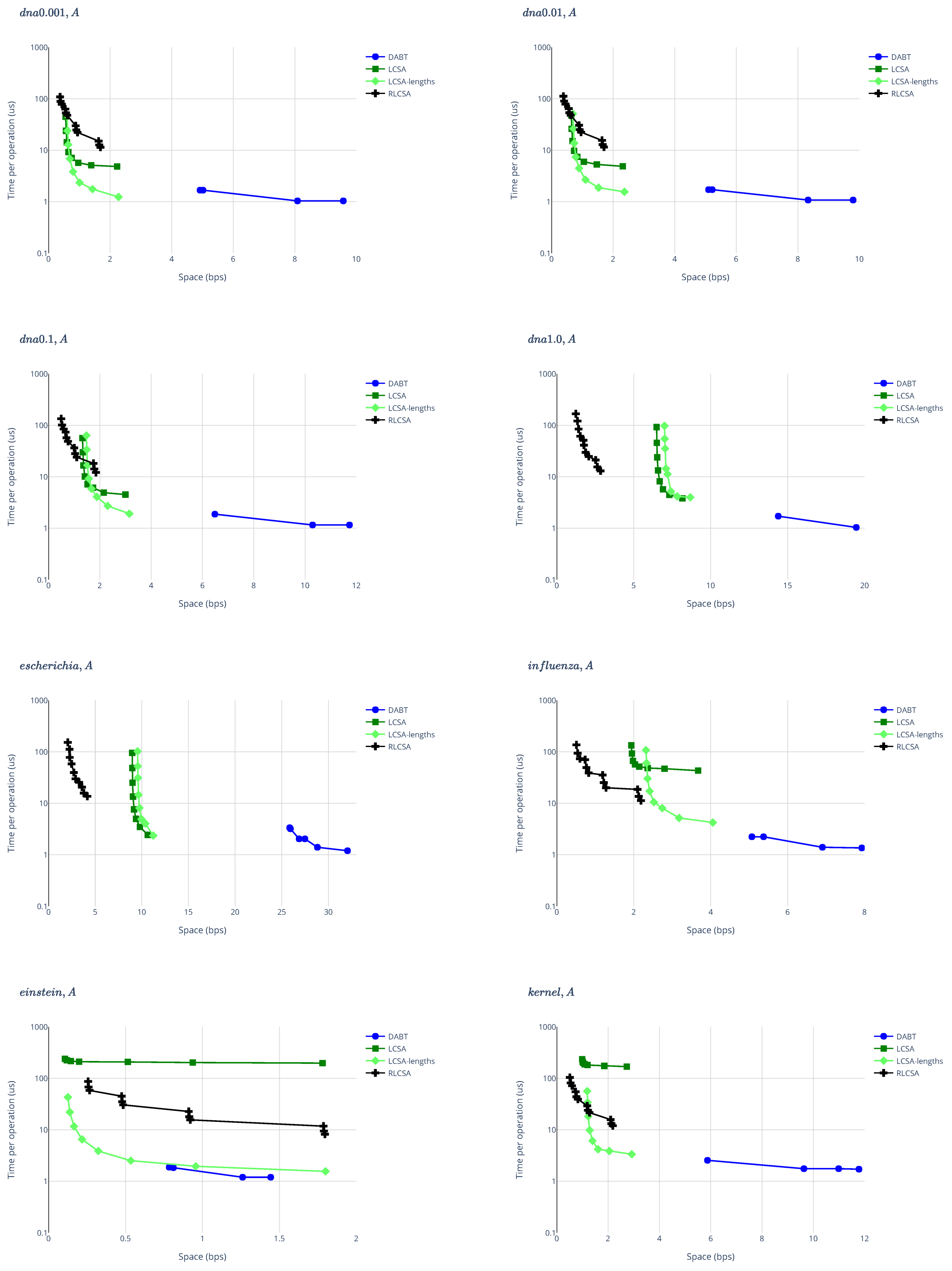


Figure 13: Performance of *access* in different representations of the suffix array. The y-axis is time in microseconds in log-scale.

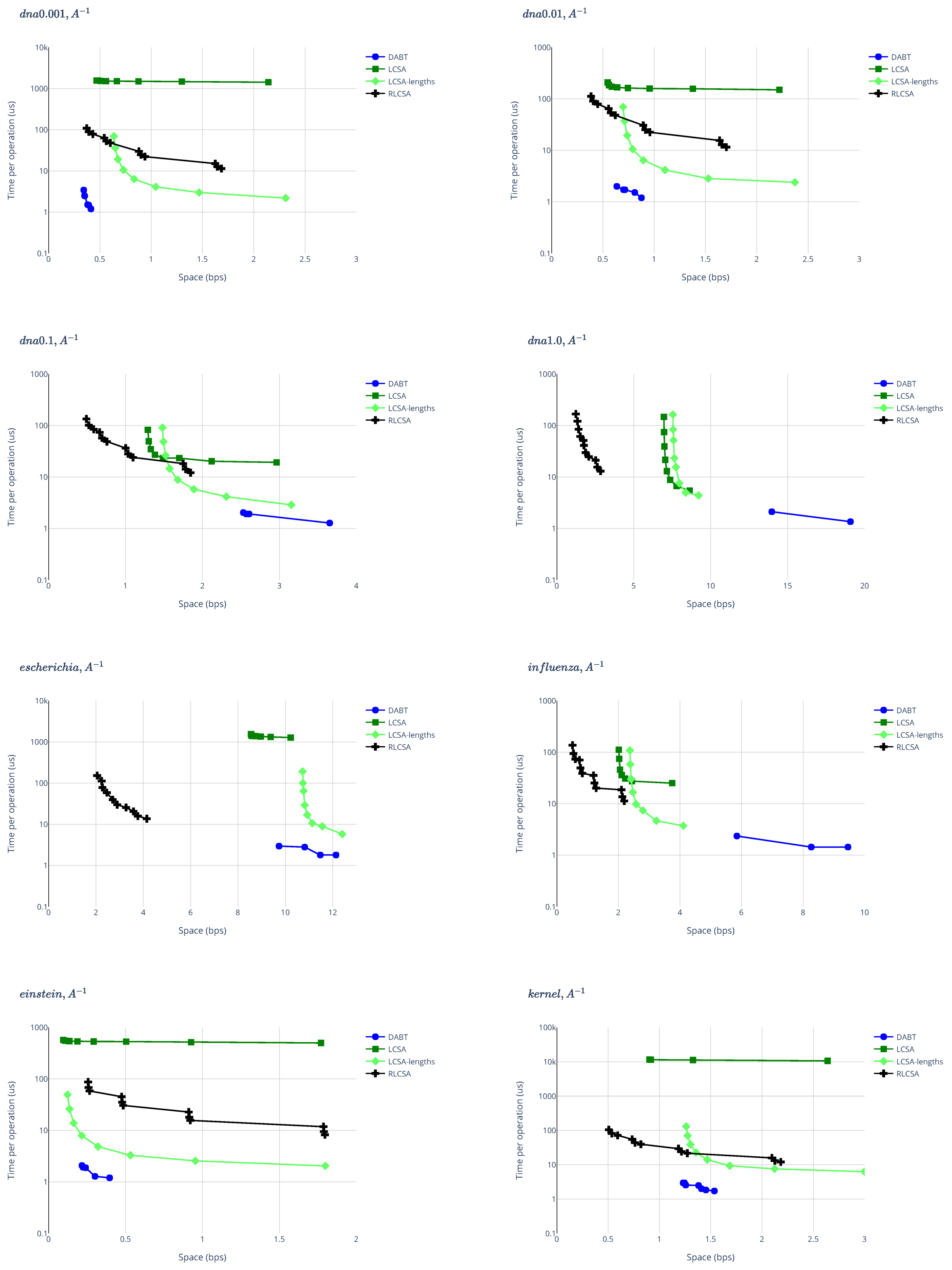


Figure 14: Performance of *access* in different representations of the inverse suffix array. The y-axis is time in microseconds in log-scale.

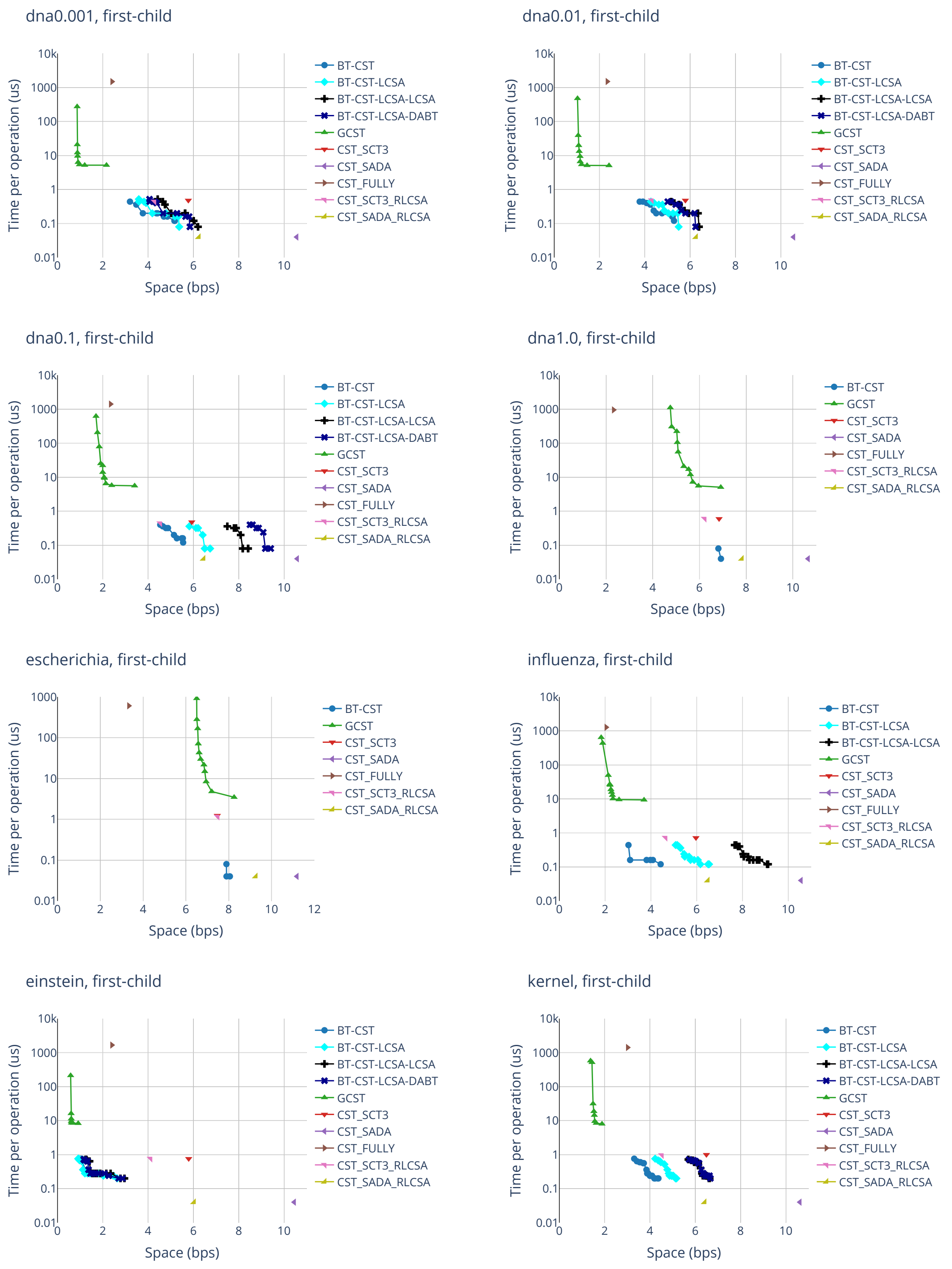


Figure 15: Performance of *first-child* in different CSTs. The y-axis is time in microseconds in log-scale.

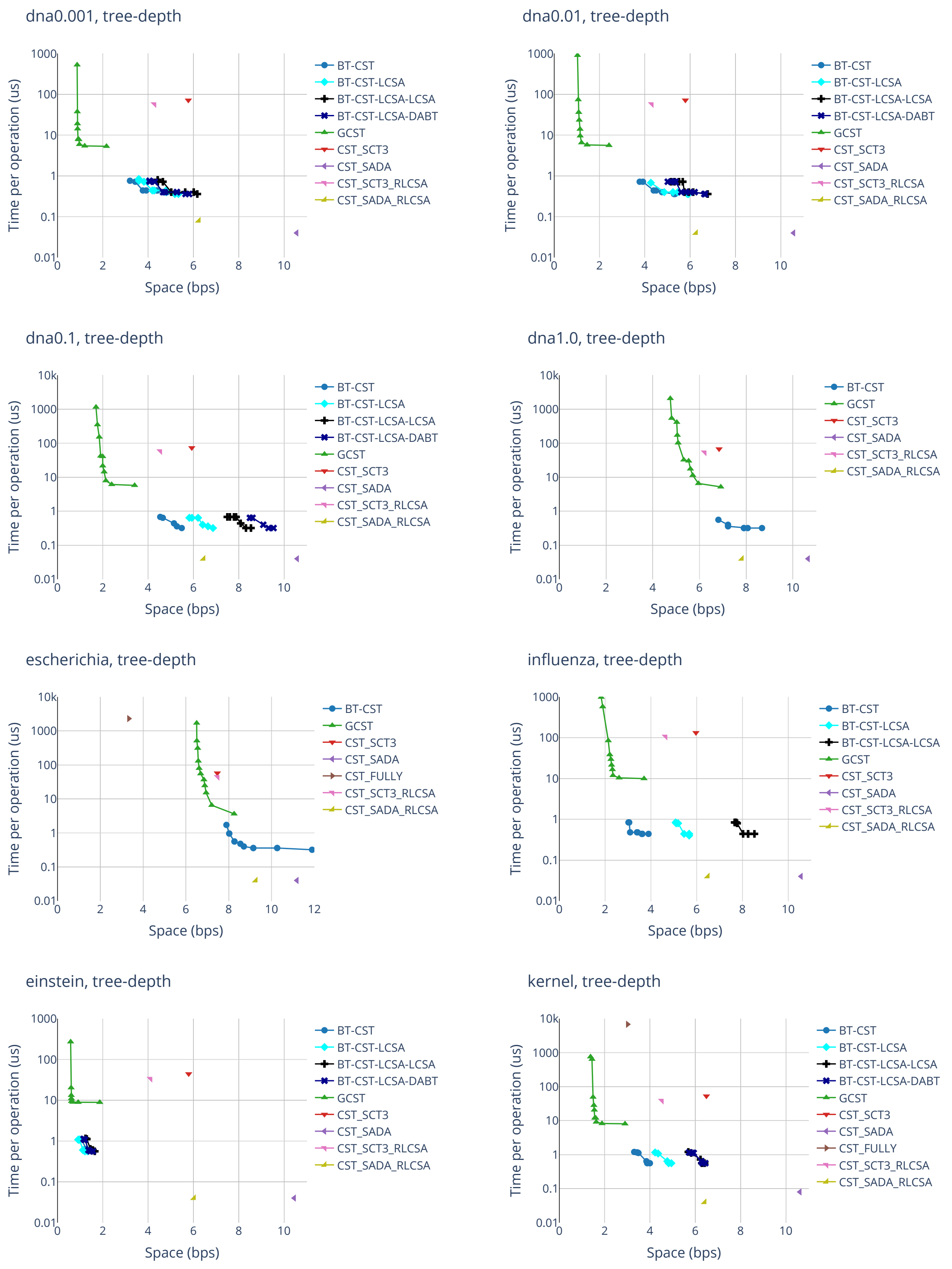


Figure 16: Performance of *tree-depth* in different CSTs. The y-axis is time in microseconds in log-scale.

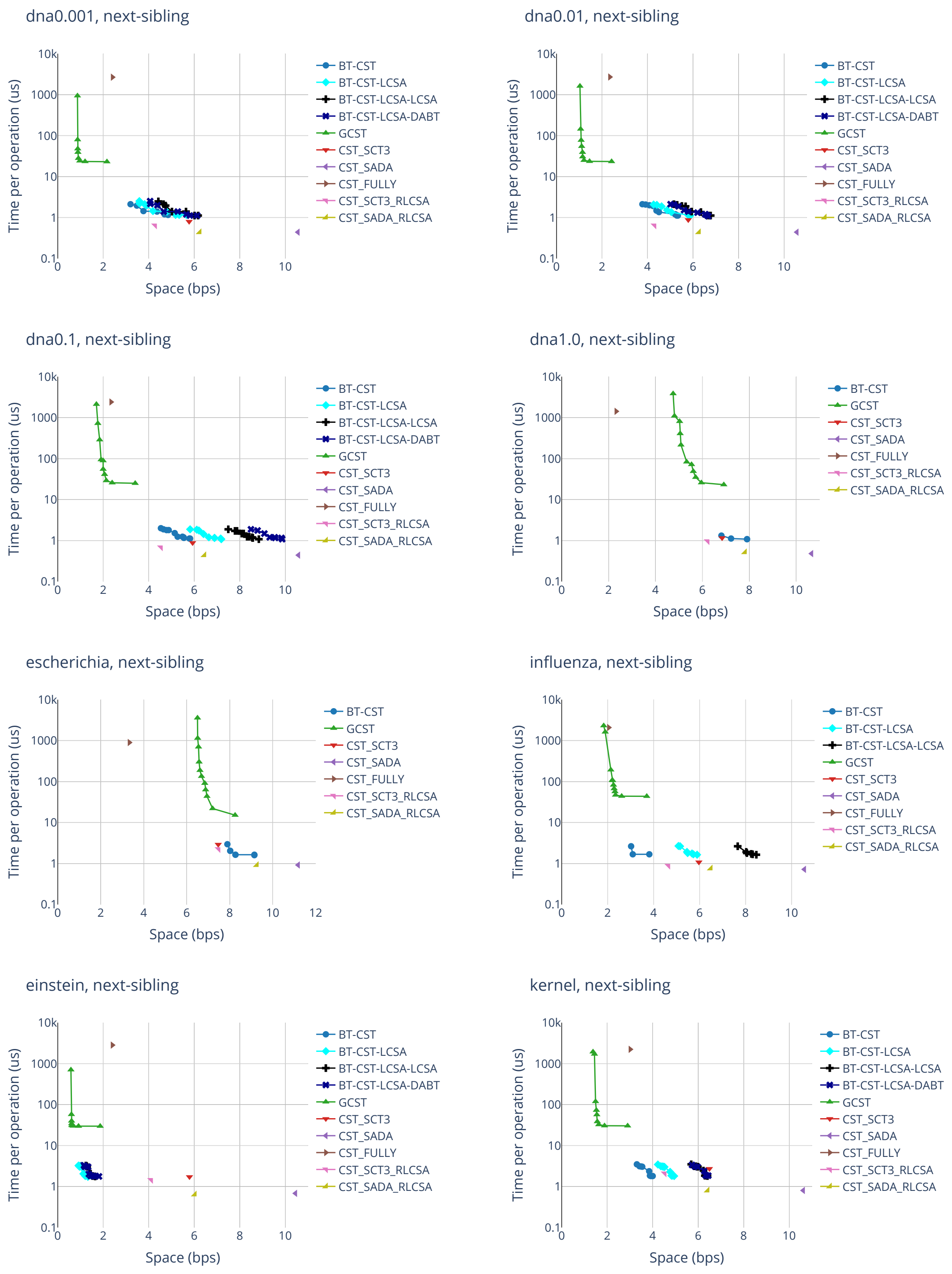


Figure 17: Performance of *next-sibling* in different CSTs. The y-axis is time in microseconds in log-scale.

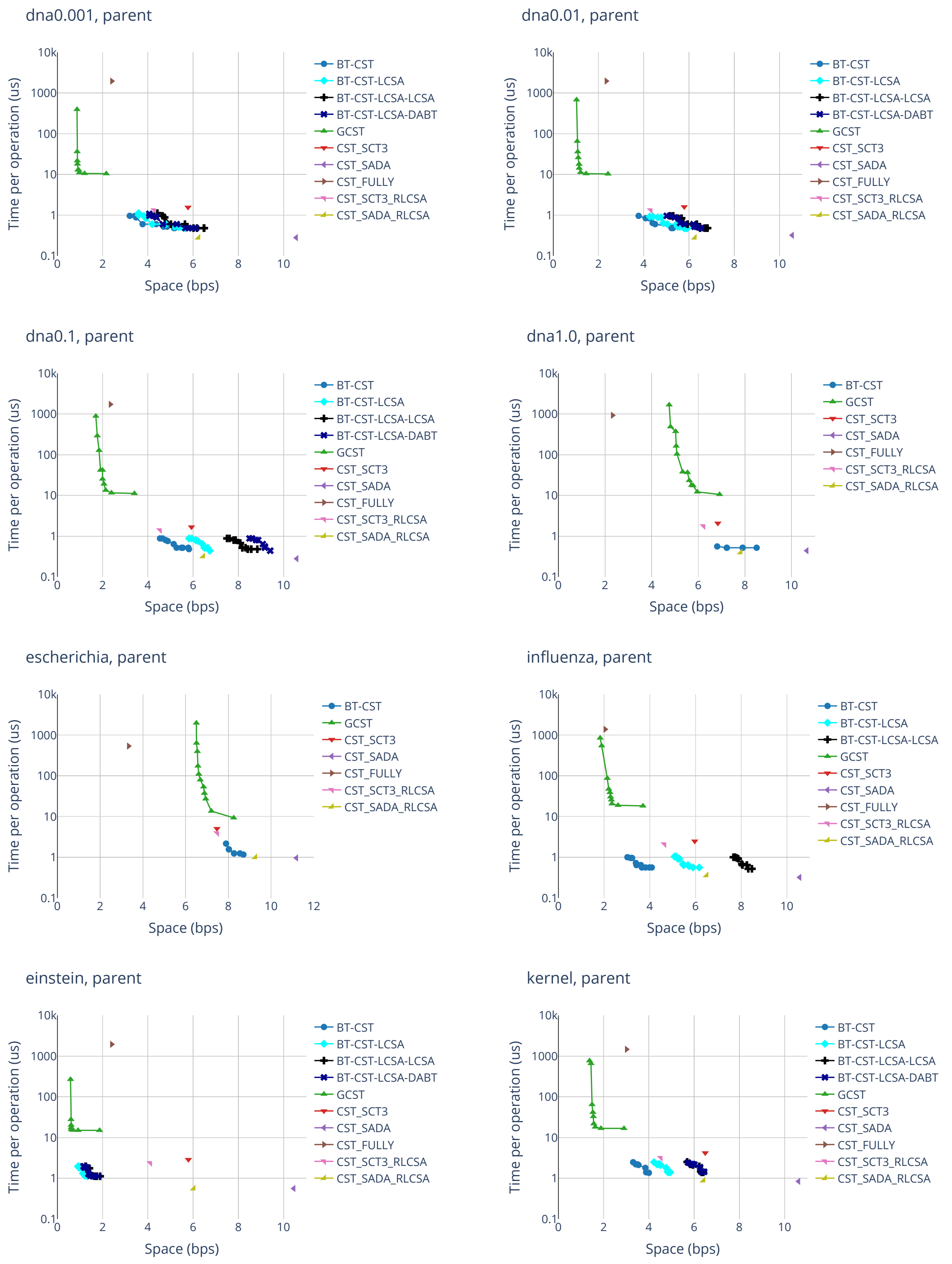


Figure 18: Performance of *parent* in different CSTs. The y-axis is time in microseconds in log-scale.

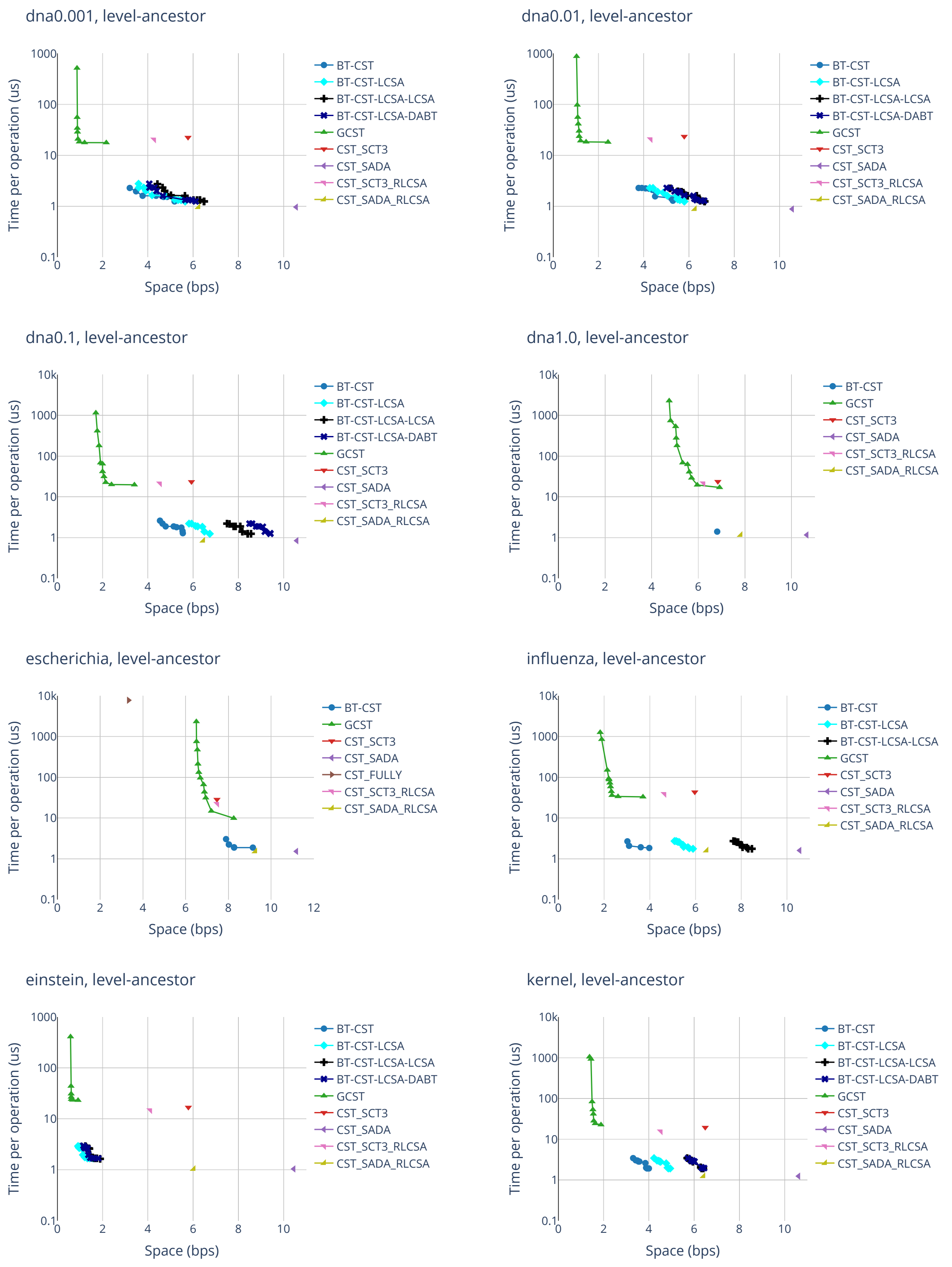


Figure 19: Performance of *level-ancestor* in different CSTs. The y-axis is time in microseconds in log-scale.

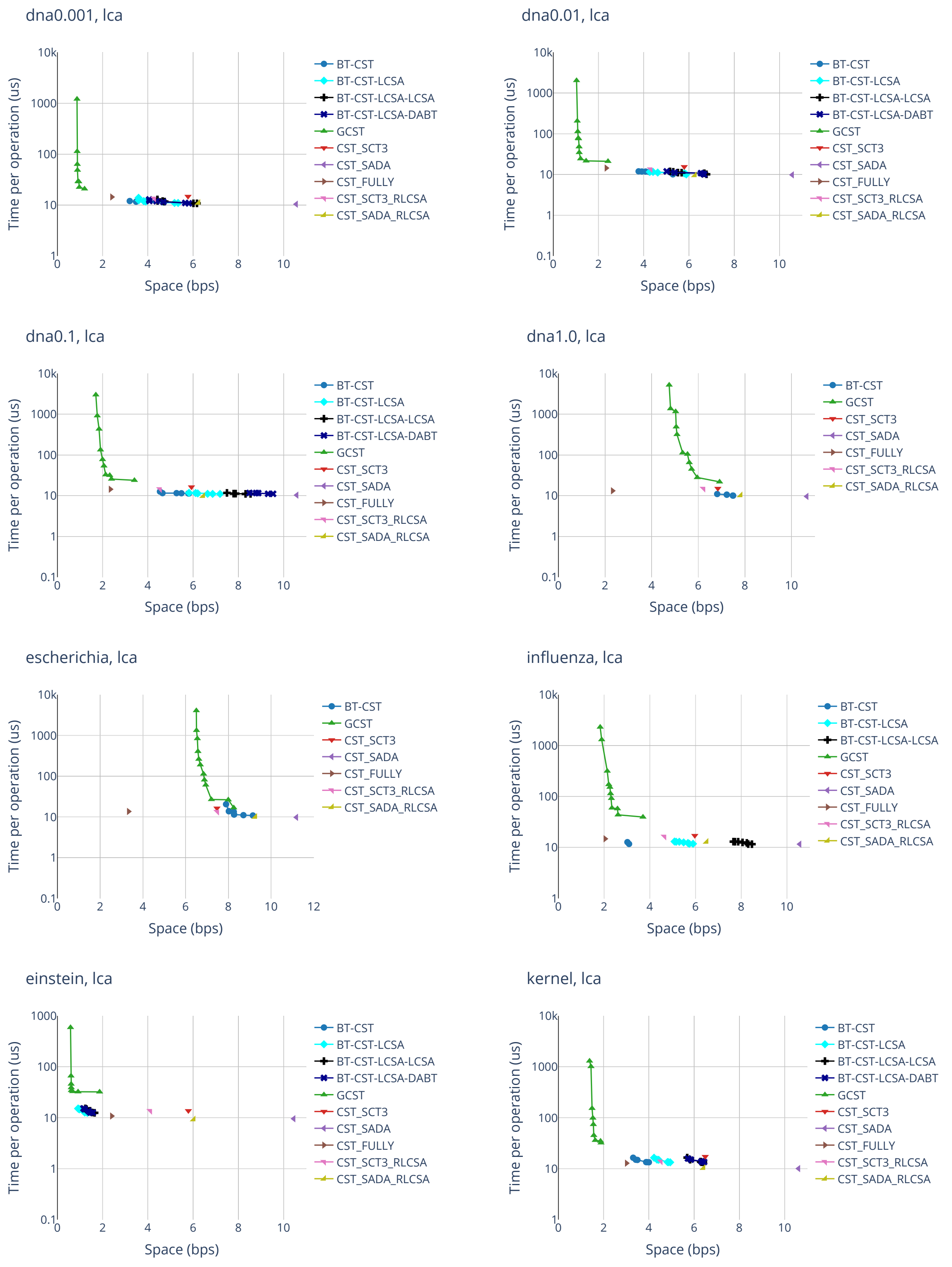


Figure 20: Performance of *lca* in different CSTs. The y-axis is time in microseconds in log-scale.

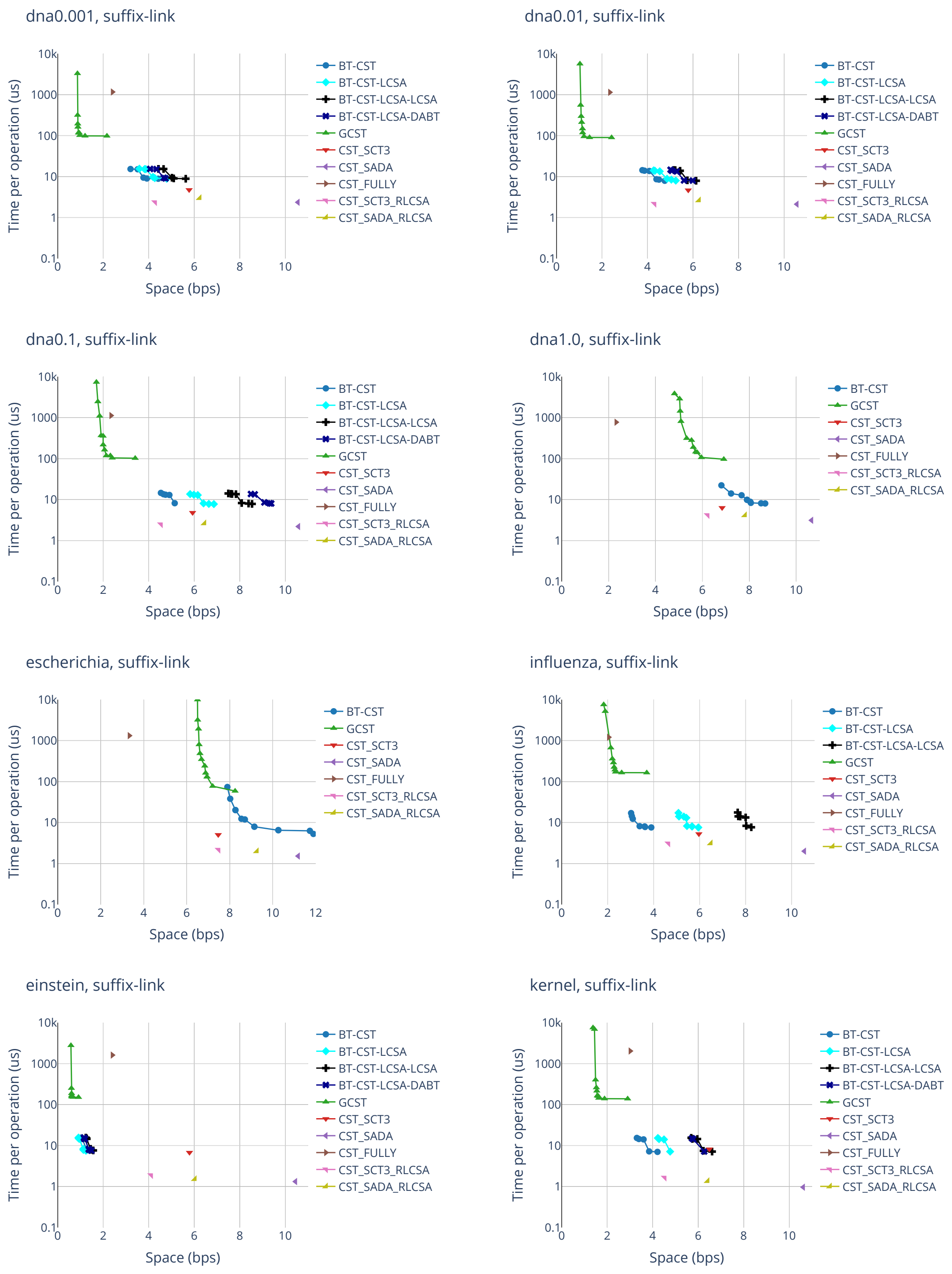


Figure 21: Performance of *suffix-link* in different CSTs. The y-axis is time in microseconds in log-scale.

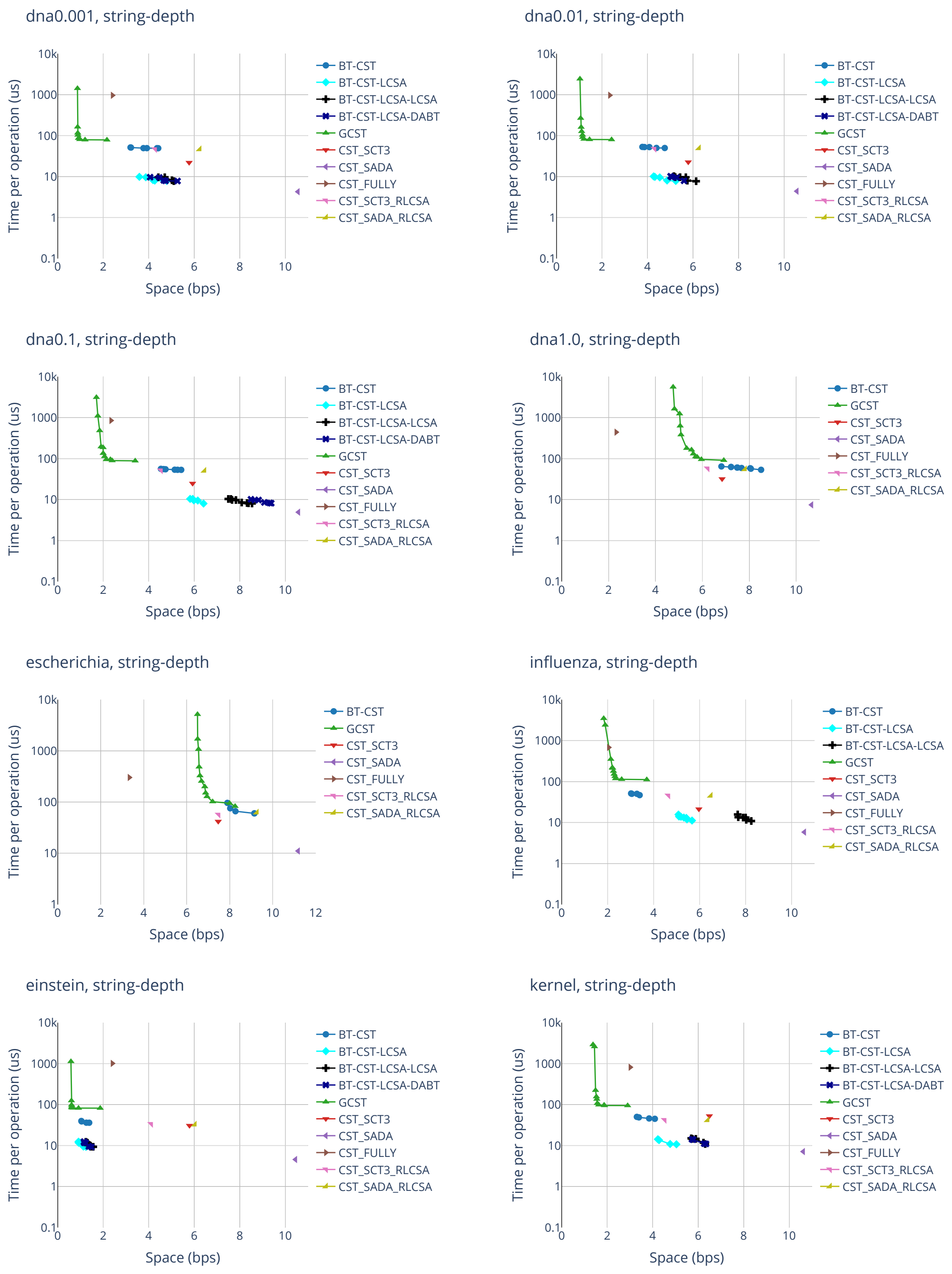


Figure 22: Performance of *string-depth* in different CSTs. The y-axis is time in microseconds in log-scale.

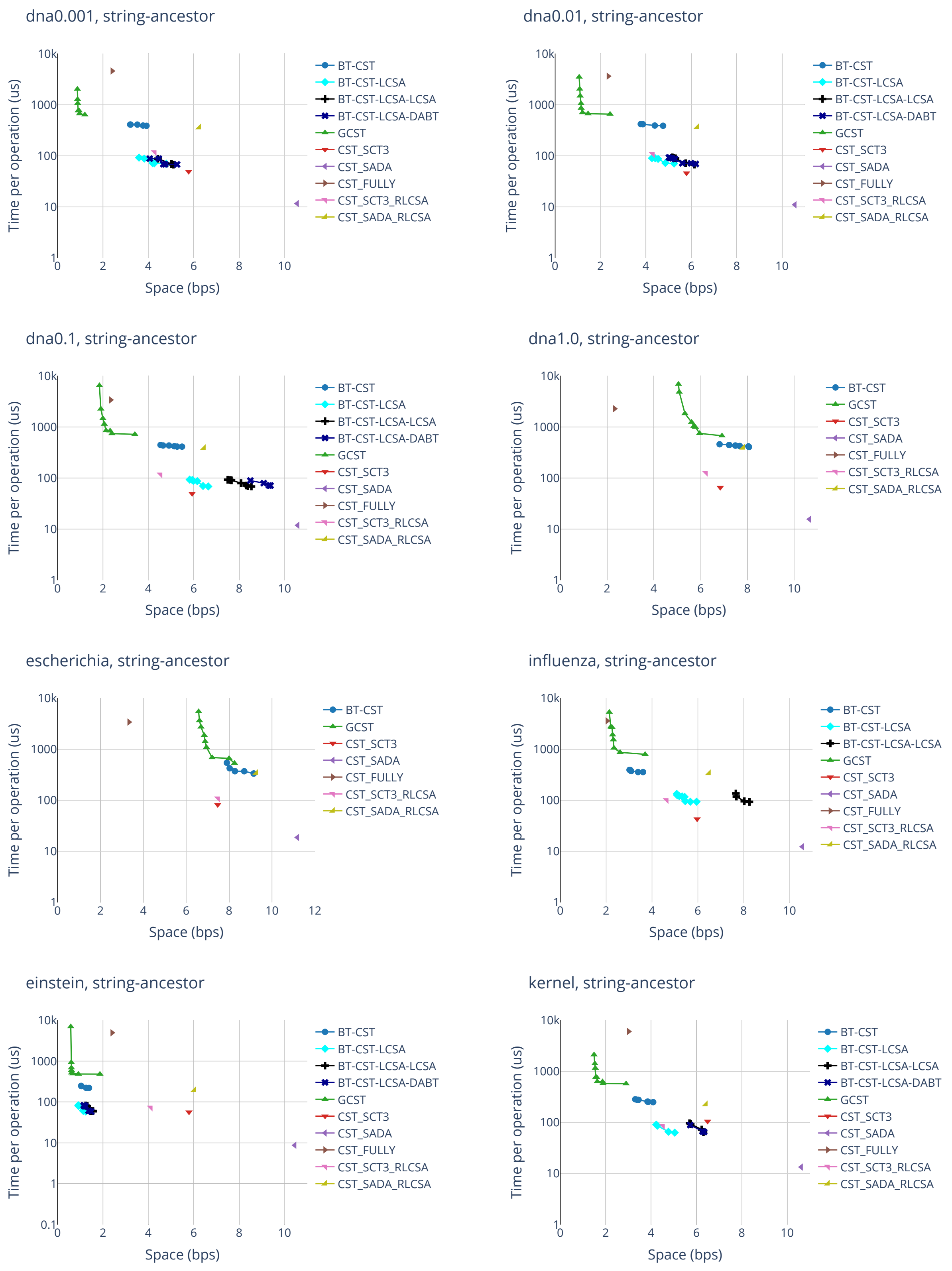


Figure 23: Performance of *string-ancestor* in different CSTs. The y-axis is time in microseconds in log-scale.

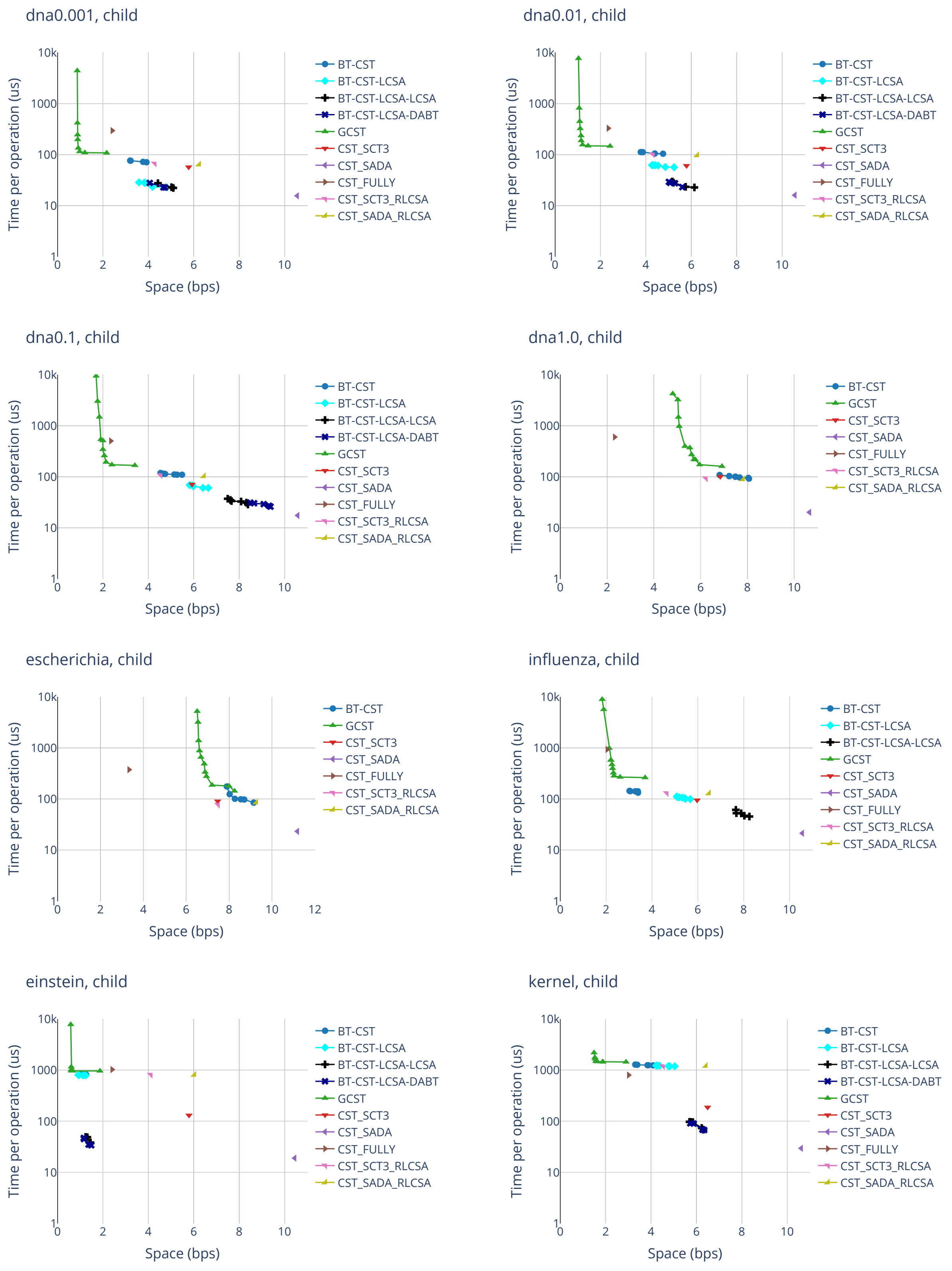


Figure 24: Performance of *child* in different CSTs. The y-axis is time in microseconds in log-scale.