UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

COMPRESSED SUFFIX TREES FOR REPETITIVE COLLECTIONS BASED ON BLOCK TREES

TESIS PARA OPTAR AL GRADO DE MAGÍSTER EN
CIENCIAS MENCIÓN COMPUTACIÓN
MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

MANUEL ARIEL CÁCERES REYES

PROFESOR GUÍA:
GONZALO NAVARRO BADINO

MIEMBROS DE LA COMISIÓN:
PABLO BARCELÓ BAEZA
BENJAMÍN BUSTOS CÁRDENAS
DIEGO ARROYUELO BILLIARDI

Este trabajo ha sido parcialmente financiado por Proyecto Fondecyt 1-170048, Proyecto Basal FB00001 (CeBiB) y Proyecto BIRDS.

SANTIAGO DE CHILE
2019

Resumen

Block Tree es una estructura de datos propuesta recientemente para representar una secuencia T de largo n en espacio acotado por el número de frases z del parseo de Lempel-Ziv de T. Usa espacio $O(z \log(n/z))$ y toma tiempo logarítmico para responder acceso y otras operaciones.

Nuestras contribuciones en el desarrollo de Block Trees son teóricas y prácticas. Demostramos nuevas propiedades, presentamos la primera implementación fiel a la descripción teórica, y ofrecemos mejoras y estudiamos un amplio conjunto de variantes de la estructura.

Los Árboles de Sufijos son una estructura de datos fundamental en procesamiento de texto, con muchas aplicaciones en bioinfomática, permitiendo soluciones eficientes a problemas complejos como búsqueda de patrones (aproximada), encontrar subcadenas repetidas, calcular estadísticas de búsqueda, calcular ocurrencias maximales, entre otros. Sin embargo, su uso de espacio, aunque lineal, es un problema importante en las aplicaciones.

Una línea de investigación que ataca este problema consiste en construir representaciones compactas de árboles de sufijos, llamadas *Árboles de Sufijos Comprimidos (CSTs)*, las cuales simulan la funcionalidad de un árbol de sufijos usando espacio acotado por la información contenida en la secuencia. Implementaciones actuales usan 10 bits por símbolo tardando unos pocos microsegundos en las operaciones; o usan 5 bits por símbolo pero las operaciones tardan milisegundos. Una rama reciente de esta área son los *Árboles de Sufijos Comprimidos conscientes de la repetitividad*, los cuales se centran en la construcción de árboles de sufijos para inputs repetitivos, así como un documento versionado o una colección de cadenas de ADN de un grupo de humanos. CSTs de esta área adaptan índices basados en compresión *run-length*, Lempel-Ziv y de Gramáticas. El CST más exitoso en la práctica es el *Árbol de Sufijos Comprimido con Gramáticas* (GCST), el cual usa cerca de 2 bits por símbolo y tarda hasta cientos de microsegundos.

En este trabajo usamos nuestros Block Trees para diseñar un nuevo CST consciente de la repetitividad. Aunque más espacioso que el GCST (usando 3–4 bits por símbolo), es más rápido por órdenes de magnitud. Para lograrlo usamos Block Trees en varios componentes del CST. Primero, para comprimir la representación de paréntesis de la topología del árbol; estos Block Trees deben ser extendidos para implementar la navegación en el árbol sobre los paréntesis. Segundo, usamos Block Trees en varios arreglos codificados diferencialmente que componen el CST: el arreglo de sufijos, su inverso y el arreglo *longest-common-prefix*. Terminamos ofreciendo un conjunto de combinaciones recomendadas para futuros usuarios.

Todo nuestro código se encuentra público en `https://github.com/elarielcl/BT-CST`.

Abstract

The Block Tree is a recently proposed data structure representing a sequence T of length n in space bounded by the number of phrases z of the Lempel-Ziv parsing of T. It uses $O(z \log(n/z))$ space and supports access to symbols and other operations in logarithmic time.

We contribute both to the theoretical and practical development of Block Trees, proving new properties, presenting the first implementation faithful to its theoretical description, making further improvements to the structure, and studying a wide set of variants.

Suffix trees are a fundamental data structure in stringology with a myriad of applications in areas like bioinformatics, enabling efficient solutions to complex problems such as (approximate) pattern matching, finding repeated substrings, computing matching statistics, computing maximal matches, and many others. However, their space usage, though linear, is an important problem in applications.

A line of research addressing this problem consists on building compact representations of suffix trees, named *Compressed Suffix Trees (CSTs)*, which simulate all the suffix tree functionality within space bounded by the information content of the sequence. Current implementations use 10 bits per symbol and support the operations in a few microseconds; or use 5 bits per symbol but their operation times raise to milliseconds. A recent branch of this area is repetition-aware CSTs, which focuses on building suffix trees for repetitive inputs, such as a versioned document or a collection of DNA sequences of a group of humans. CSTs in this area adapt Lempel-Ziv, Grammar and run-length based indexes. The most successful CST in practice is the grammar-compressed suffix tree (GCST), which uses about 2 bits per symbol and supports the operations in tens to hundreds of microseconds.

We apply our Block Trees to design a new repetition-aware CST that, though larger than the GCST (i.e., using 3–4 bits per symbol), is faster by orders of magnitude. We obtain this result by using Block Trees on various components of the CST. First, we build them on the parentheses representation of the tree topology; these Block Trees must be augmented to support tree navigation operations on the parentheses. Second, we use Block Trees on various differentially-encoded arrays that compose a CST: the suffix array, its inverse, and the longest-common-prefix array. We end up with a set of recommended combinations for practitioners.

All our code is publicly available at `https://github.com/elarielcl/BT-CST`.

A quienes construyen nuevo conocimiento.

A quienes transmiten conocimiento.

A mi familia.

Y al amor de mi vida Constanza.

Agradecimientos

Este trabajo no hubiese sido posible sin la ayuda de diversas personas que me apoyaron y motivaron en distintas etapas.

A quienes crearon el conocimiento sobre el cual he sido formado. Y a quienes me transmitieron estos conocimientos. Gracias especiales a quienes más me motivaron e inspiraron, mi padre Luis, y mis profesores Teresa Miranda, Jonathan Gárate, Juan Álvarez, Pablo Dartnell, Patricio Poblete, Jorge Pérez, Jérémy Barbay, Gonzalo Navarro. Y a quienes comparten su conocimiento a través de cursos online gratuitos, especialmente a Robert Sedgewick, Erik Demaine y sus equipos. También a quienes respetuosamente me escucharon cuando fui profesor auxiliar y profesor de cátedras en la Universidad. Cada una de esas clases las preparé pensando en ustedes. Espero haber transmitido eficazmente los conocimientos e inspirado a gente, así como otros me han inspirado a mí.

A mis amigos, quienes me acompañaron en este proceso universitario, escuchando mis conversaciones filosóficas, motivándome cuando lo necesitaba o simplemente compartiendo un rato de relajo. Gracias especiales a Nicolás Gajardo y mi primo Cristian, con quienes compartí domicilio, también a mis primeros amigos universitarios: Susana, Lerko y Daniel, al grupo de amigos "Los Tía Sonia" quienes siempre se preocuparon por mantener la amistad, y a los innumerables amigos que gané en el DCC, con quienes además comparto mi pasión por las ciencias de la computación.

A quienes me ayudaron en el desarrollo de esta tesis, ya sea explicándome conceptos clave, escuchando presentaciones y dando consejos y correcciones, o facilitándome y explicándome software. Gracias especiales a Travis Gagie, Simon Puglisi, Luís Russo, Alberto Ordóñez, a los miembros de la comisión y a Simon Gog por la librería sdsl que fue de gran ayuda en el desarrollo del trabajo. A Adrián Gómez y Daniel Valenzuela por apadrinarme y guiarme en los países que visité, ustedes se han convertido en ejemplos para mi pensando en el doctorado.

Gracias a mi profesor guía Gonzalo Navarro, quien confió en mi desde el comienzo. Cuando comencé Gonzalo me sugirió leer un sinnúmero de publicaciones (incluyendo su libro Compact Data Structures) sin muchas de las cuales creo que podría haber realizado este trabajo de todas maneras, pero que me ayudaron a darme cuenta de lo mucho que me gusta la investigación y decidirme por realizar un doctorado. A lo largo de estos años él se a convertido en un referente en la academia para mí; siempre trabajando en su investigación con muchos colaboradores, pero también reservando tiempo para preocuparse por sus cursos, sus alumnos y su familia. No fuimos muy regulares en nuestras reuniones, pero siempre que necesitaba de su ayuda se hizo el tiempo para atenderme tanto académica como humanamente. Quiero

agradecer en particular todas las oportunidades que me brindó y el apoyo en este trabajo, especialmente en la parte final donde nos reunimos más frecuentemente.

A mi familia, que formó gran parte de lo que soy como persona y siempre me apoyó en todos los desafíos que he enfrentado, en especial con quienes he vivido la mayor parte de mi vida, mis padres Erika y Luis, por ser tan diferentes :) y mostrarme que el mundo es mucho más de lo que pueden contarme o incluso más de lo que yo soy capaz de percibir, a mi hermana Erika por quererme y defenderme incondicionalmente, y a mi tía Paty y mi abuelita Fresia por criarme y amarme como si fuese un hijo más. Les agradezco todo lo que hicieron por mi y los amo con todo mi corazón. También quiero agradecer a la familia de mi polola Constanza; Pablo, Jeannette y Andrea por ayudarme en mis últimos años de universidad, tratándome como un miembro más de su familia, dándome acogida en su hogar e incluso invitándome de vacaciones.

Finalmente, quiero agradecer a Constanza Escobar, el amor de mi vida, por amarme tanto, por apoyarme incondicionalmente, por cuidarme, por escucharme divagar acerca de este trabajo, por motivarme y por muchas cosas que solo tu podrías generar en mí. Nuestra vida juntos ya ha comenzado y estoy ansioso por ver como la construiremos.

Contents

Introduction **1**
Outline and Main Contributions . . . 3
Publications . . . 3

1 Basic Concepts **4**
1.1 Empirical Entropy . . . 4
1.2 Arrays . . . 5
1.3 Rank/Select/Access (RSA) . . . 6
1.3.1 Bitvectors . . . 6
1.3.2 Wavelet Trees . . . 7
1.4 Succinct Tree Representations . . . 7
1.5 Range min-Max Tree . . . 9
1.6 Suffix Array and Suffix Tree . . . 11
1.7 Dictionary Compressors . . . 11
1.8 Karp-Rabin Algorithm . . . 13

2 Related Work **14**
2.1 RSA-Structures using Grammar Compression . . . 14
2.2 Compressed Suffix Arrays . . . 15
2.3 Compressed Suffix Trees . . . 16
2.3.1 The First Compressed Suffix Tree . . . 16
2.3.2 Fully-Compressed Suffix Tree (FCST) . . . 17
2.3.3 Faster Entropy-Bounded CST . . . 17
2.4 Repetition-Aware Compressed Suffix Trees . . . 18
2.4.1 The First Repetition-Aware CST . . . 18
2.4.2 Grammar-Compressed Suffix Tree (GCST) . . . 19
2.4.3 Other Approaches . . . 19
2.5 Repetitive Datasets . . . 19
2.6 Succinct Data Structure Library . . . 20

3 Block Tree Improvements **22**
3.1 Block Trees . . . 22
3.1.1 Rank and Select . . . 24
3.2 Previous Implementation of Block Trees . . . 25
3.2.1 Definition . . . 26
3.2.2 Construction . . . 26

3.2.3 Compact Representation . . . 27
3.3 Improvements . . . 28
3.3.1 No Padding Block Tree . . . 28
3.3.2 Compressed Compact Representation . . . 29
3.3.3 Construction Algorithm . . . 29
3.3.4 Elimination of First Levels . . . 30
3.3.5 Pruning . . . 30
3.3.6 New Sets of Rank/Select Fields . . . 32
3.3.7 New Heuristics . . . 32
3.4 Experimental Results . . . 34
3.4.1 Setup and Datasets . . . 34
3.4.2 Experiments and Structures . . . 34
3.4.3 Comparisons and Analysis . . . 35

4 Compression of Repetitive Differential Arrays 51
4.1 LCSA Adaptation . . . 51
4.2 GCC Adaptation . . . 52
4.3 Block Tree Adaptation . . . 54
4.4 Experimental Results . . . 54
4.4.1 Setup and Datasets . . . 54
4.4.2 Experiments and Structures . . . 55
4.4.3 Results and Analysis . . . 55

5 Block Tree Compressed Topology 60
5.1 Block Tree Augmentation . . . 60
5.1.1 Stored Fields . . . 61
5.1.2 Fields Computed on the Fly . . . 61
5.2 Description of Primitives . . . 62
5.2.1 *leaf-rank*(i) and *leaf-select*(j) . . . 63
5.2.2 *fwd-search*(i, d) and *bwd-search*(i, d) . . . 63
5.2.3 *min-excess*(i, j), *min-count*(i, j) and *min-select*(i, j, t) . . . 66
5.3 Further Optimizations on Primitives . . . 69
5.3.1 r-ary rmM-tree on First Level . . . 69
5.3.2 BT-CT Versions . . . 70
5.4 Experimental Results . . . 70
5.4.1 Setup and Datasets . . . 70
5.4.2 Experiments and Structures . . . 71
5.4.3 Results and Analysis . . . 71

6 Our Repetition-Aware Compressed Suffix Trees 79
6.1 BT-CT Adaptation . . . 79
6.1.1 Operation child . . . 80
6.2 RLCSA Speed Up . . . 80
6.3 Experimental Results . . . 80
6.3.1 Setup and Datasets . . . 80
6.3.2 Experiments and Structures . . . 81
6.3.3 Results and Analysis . . . 82

Conclusion and Future Work **96**

Bibliography **103**

Introduction

Suffix trees [70, 47, 69] are one of the most appreciated data structures in Stringology [3] and in application areas like Bioinformatics [30], 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 [48], authorship attribution [71], document retrieval [31], 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 $\Theta(n \log n)$ bits, whereas the information contained in the text is, in the worst case, just $n \log \sigma$ bits. From a practical point of view, even carefully engineered implementations [40] require at least 10 bytes per symbol, which may force applications to run the suffix tree on the (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 nucleobases represented with $\log 4 = 2$ 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 [15, 20]. 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 [46], 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 $O(n \log \sigma)$ 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 [67]. A recent implementation [58] 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 [64], 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% [68]. This repetitiveness is not well captured by statistical compression methods [38], on which most of the CSTs are based. Lempel-Ziv [43] and grammar [37] based compression techniques, among others, do better in this scenario [49], but only recently we have seen CSTs building on them, both in theory [25, 8] and in practice [1, 53]. 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 [63], 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 thesis we explore an alternative to grammar compression called Block Trees [10, 61], 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-node representations that are blind to repetitiveness [55]. We use the BT-CT to represent suffix tree topologies, but it might also be useful in other scenarios, such as representing the topology of repetitive XML collections [4].

Besides tree topologies, we contribute to the development of Block Trees for general sequences with contributions ranging from theoretical (new properties and versions of the structure) to practical (presenting the first Block Tree implementation faithful to its theoretical description and running an exhaustive set of experiments on each of the proposed features).

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.

Finally, we use Grammar and Block Tree-based representations to improve the performance and compressibility of the RLCSA [45], a key component used on repetition-aware Compressed Suffix Trees. We replace the sampling of the RLCSA, which has been difficult to compress [45], for each of our representations. We obtain up to 2 orders of magnitude improvement on operations related to the RLCSA, and better compressibility on our most repetitive tested inputs.

Outline and Main Contributions

Chapter 1: We explain the basic concepts and structures needed to understand the problem we are going to deal with.

Chapter 2: We describe Compressed Data Structures relevant to the work done in this thesis. We start by describing entropy-based structures and then the state-of-the-art solutions aware of the repetitiveness of its input. We also describe the datasets we will use to tests our implementations.

Chapter 3: We present our contributions to improve the current development of Block Trees. Our contributions range from theoretical to practical. We also made an exhaustive set of experiments to test each of these proposed improvements. This chapter presents the first Block Tree implementation faithful to its theoretical description.

Chapter 4: We adapt Block Trees and Grammar-based indexes to create array representations that capture the repetitiveness of their differential encodings. We test these new array representations in the case of suffix array, its inverse, and the LCP array.

Chapter 5: We present our new BT-CT, a Block-Tree-based representation of tree topologies. We compare its performance against other repetition-aware topology representations, and obtain that for a relatively small space penalty the BT-CT is able to reduce the time performance by an order of magnitude.

Chapter 6: We merge the work done in Chapters 5 and 6 to create the BT-CST, a new repetition-aware CST whose times are closer to those reached by general-purpose CSTs.

Conclusion and Future Work: We draw our conclusions and further lines of research that could be explored based on the results obtained in this work.

Publications

Part of this work (preliminary versions of Chapters 6 and 7) was submitted to the *String Processing and Information Retrieval* (SPIRE) 2019 conference under the title "Faster Repetition-Aware Compressed Suffix Trees based on Block Trees", Manuel Cáceres & Gonzalo Navarro. A preliminary version can be found in arXiv at `https://arxiv.org/abs/1902.03274`.

Chapter 1

Basic Concepts

In this chapter we introduce the fundamental tools to understand the previous work and contributions in the context of this thesis. We will be working with a text T of size n over an alphabet Σ of size σ, i.e., $T \in \Sigma^n$, where $|\Sigma| = \sigma$. T will be 1-based indexed, $T = T[1,n] = T[1]...T[n]$; then the i-th character of T is $T[i]$ and the i,j-substring (with $1 \leq i \leq j \leq n$) of T is denoted $T[i,j] = T[i]...T[j]$. A substring is called a suffix if $j = n$ and a prefix when $i = 1$. For the relevant case when $\sigma = 2$, we will change the notation to $B[1,n]$ and call it a bitmap, moreover, in this case we will assume $\Sigma = \{0,1\}$. Our logarithms will be in base 2, $[1,n]$ is the set $\{1,..,n\}$ and our computations are described under the word RAM model with word length $w = \Theta(\log n)$, that is, *access*, mathematical, and logical operations on $\Theta(\log n)$-bit integers are carried out in constant time.

1.1 Empirical Entropy

In Coding and Information Theory [17] codes are assigned to characters to represent them and Statistical Compressors use these codes to represent their original inputs. In this context, the empirical entropy $\mathcal{H}$ of T is defined, in general, as the minimum number of bits needed to unambiguously identify a random character from T.

There are different kinds of entropies; depending on the context we will use:

Worst-case entropy, $\mathcal{H}_{wc}(T) = \log \sigma$. We obtain it if we restrict the codes to be of the same length.

Zero-order entropy,

$$\mathcal{H}_0(T) = \sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n}{n_c}$$

which is the minimum average number of bits needed to represent the symbols of T (n_c is the number of cs in T) if we use the same code to represent the same symbol.

kth-order entropy,

$$\mathcal{H}_k(T) = \sum_{C \in \Sigma^k} \frac{|T_C|}{n} \cdot \mathcal{H}_0(T)$$

which is the minimum average number of bits needed to represent the symbols of T, but letting each code depend on the k-length preceding contexts to assign the codes (T_C is the string formed by collecting the symbol that follows each occurrence of the context C in T). It can be shown that $0 \leq \mathcal{H}_k(T) \leq \mathcal{H}_{k-1}(T) \leq ... \leq \mathcal{H}_0(T) \leq \mathcal{H}_{wc}(T)$.

A famous result [17] states that any Statistical Compressor using k-length contexts must use at least $n\mathcal{H}_k(T)$ bits to represent T. A well-known compressor of this type is the *Huffman compression algorithm* [32], whose encoding can represent the text optimally in less than $n(\mathcal{H}_0(T) + 1)$ bits. To get this space Huffman's algorithm computes the frequencies of each symbol and uses them to build the corresponding *Huffman tree*, a *trie* representing the codes for the symbols on its *root-to-leaf* paths.

The research area of *Compact Data Structures* is born from the intersection of Information Theory and Data Structures. Its objective consists in building compressed representations of data structures using space near their theoretic information content (i.e., entropy), yet answering queries without a major time penalty.

1.2 Arrays

One of the simplest data structures are *Arrays*, which allow *access* to a sequence A of n integer numbers using at most w bits. If l is the maximum number of bits used by an integer in the sequence, it is easy [51] to achieve constant-time *access* using $n \cdot l$ bits (instead of the classical $n \cdot w$ bits), that is, if we see A as a sequence it will require $n\mathcal{H}_{wc}(A)$ bits. We call this an *uncompressed array*.

There exist various encoders [51] for *Compressed Arrays*; we mention two that are important for the development of this work:

Differential encoding, corresponds to the general technique of storing the differences $A[i] - A[i-1]$ and a sampling of the original array to efficiently recover the absolute values. This option is advisable when working on increasing sequences or on specific problems that take advantage of the differences.

Direct access codes (DACs) [11], corresponds to a regrouping of a chunk division of the elements. Let $b_1, .., b_r$ the chunk sizes, such that $\sum_{i=1}^{r-1} b_i < l_{max} \leq \sum_{i=1}^{r} b_i$, where l_{max} is the maximum bit-length of an element in A. The element $A[j]$ of bit-length $l_j \leq l_{max}$ is divided into chunks of size $b_1, .., b_c$, where $\sum_{i=1}^{c-1} b_i < l_j \leq \sum_{i=1}^{c} b_i$, in the following way: the first chunk corresponds to the $l_j - \sum_{i=2}^{c} b_i$ most significant bits prepended with $l_j - \sum_{i=1}^{c} b_i$ zeros; the second chunk is the next b_2 most significant bits, the third chunk is the next b_3 most significant bits, until the c-th chunk, which contains the b_c least significant bits. These

chunks are reorganized so that the most significant chunks are grouped together, then second most significant chunks, and so on. To decode a number, the k-th group uses a bitvector (see Section 1.3.1) marking the member elements of that group (i.e., with bit-length $> \sum_{i=1}^{k-1} b_i$) that do not belong to the next group (i.e., with bit-length $\leq \sum_{i=1}^{k} b_i$). Brisaboa et al. [11] showed that optimal chunk sizes can be computed efficiently by dynamic programming. They also experimentally show to be superior over other encoders. When we refer to *DACs*, we mean this space-optimized version.

1.3 Rank/Select/Access (RSA)

A little more complex set of queries are *rank*, *select*, and *access*, where:

- $access(i) = T[i]$, the symbol at position i.
- $rank_c(i) = |\{k \mid k \leq i \wedge T[k] = c\}|$, the number of cs up to position i in T.
- $select_c(j) = \min(\{k \mid rank_c(k) = j\} \cup \{\infty\})$, the index of the j-th c in T.

We call an RSA-structure a structure answering these three queries. Though simple, these queries suffice to implement efficient algorithms for a variety of problems such as *inverted indices* [5, 6, 7], *self-indices* [21, 22], and *document listing* [50], among others.

1.3.1 Bitvectors

A *bitvector* is an RSA-structure over a bitmap. We call a *plain bitvector* an *uncompressed array* of B using $o(n)$ extra bits to support *rank* and *select*. Jacobson [33] showed how to implement *rank* in constant time and Clark [14] did the same for *select*. Both can be achieved by *blocking* techniques, that is, divide the input into convenient sized superblocks, blocks and mini-blocks, and storing summary information for them. Besides the *plain bitvector*, it is important to mention three *compressed bitvectors* we will use in this thesis:

RRR bitvector. An $n\mathcal{H}_0(B) + o(n)$-bit representation is achieved by Raman et al. [62], where the $n\mathcal{H}_0(B)$ term comes from dividing B into blocks and representing them with a class and an offset, and the $o(n)$ extra bits are added to answer *rank* and *select* in constant time.

Sparse bitvector. Okanohara and Sadakane [59] proposed a bitvector that is space-efficient when the number m of 1s is very small ($m \ll n$). It uses $n\mathcal{H}_0(B) + O(m)$ bits, solves *rank* in $O\left(\log \frac{n}{m}\right)$ time and $select_1$ in constant time. In broad terms, it consists in storing the $select_1$ values in a particular way, separating the highest and lowest bits and representing them in different $n\mathcal{H}_0(B)$-bit and $O(m)$-bit bitvectors, respectively.

OZ bitvector. Delpratt et al. [18] showed how to transform a bitvector with long runs of 0s and 1s into two sparse bitvectors, O and Z, representing the runs of ones and zeros, respectively. *select* queries on B are easily translated into *select* queries on O or Z and *rank* and *access* are solved by binary searching the precomputed *select* answers.

1.3.2 Wavelet Trees

In the case of general sequences, one of the most practical RSA-structures is the *wavelet tree* [28]. It uses $n \log \sigma + o(n \log \sigma) + O(\sigma \log n)$ bits and solves *rank*, *select* and *access* in $O(\log \sigma)$ time. The *wavelet tree* is a binary tree where each node represents a subsequence of T over a subalphabet $\mathcal{A}$ of the original Σ. The represented subsequence consists of the characters of T belonging to $\mathcal{A}$. At the root the alphabet is partitioned into two roughly equal subalphabets, the first containing the $\sim \sigma/2$ lowest symbols in the alphabet and the second containing the rest. This process is repeated on every node until the alphabet reaches size 1 at leaves[1]. To represent the partition at each node (except at leaves), a constant-time *bitvector* is used to mark the characters that belong to each subalphabet.

Access and *rank* are solved by a *root-to-leaf* traversal of the tree, using *rank* queries on the bitvectors to relocate the query to the subalphabets. On the other hand, *select* is solved with a *leaf-to-root* traversal, using *select* queries on the bitvectors to re-accommodate the index to the current subalphabet.

Mäkinen and Navarro [44] showed how to give a Huffman shape to the *wavelet tree*, i.e., the shape of the Huffman tree of T. This wavelet tree is called *Huffman-shaped wavelet tree* and achieves $n(H_0(T) + 1)(1 + o(1)) + O(\sigma \log n)$ bits of space. Claude et al. [16] presented the *wavelet matrix* and the *Huffman-shaped wavelet matrix*, which are adaptations of the previous trees where the bitvectors in a level are permuted and concatenated. This gets rid of the *wavelet tree* pointers, thus the *wavelet matrix* is suitable when the alphabet is big because it removes the $O(\sigma \log n)$ bits of *wavelet trees.*

1.4 Succinct Tree Representations

A balanced parentheses (BP) representation (there are others such as LOUDS or DFUDS, see [63] for a comprehensive survey) of the topology of an ordinal tree $\mathcal{T}$ of t nodes is a bitvector $P[1, 2t]$ built as follows: we traverse $\mathcal{T}$ in preorder, writing an open 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". Interpreting nodes as the position of their opening parenthesis in P, the following primitives can be defined on P:

- $excess(i) = rank_1(i) - rank_0(i)$, the number of open parentheses minus close parentheses up to position i.

[1]Note that this partition strategy yields a height of $\sim \log \sigma$.

Table 1.1: List of typical tree operations; v and u are the identifiers (positions of open parenthesis) of their corresponding nodes.

Operation	**Description**	**Reduction to primitives**
root()	The root of the tree	1
is-leaf(v)	True if v is a leaf node	True iff $access(v+1)=0$
first-child(v)	The first child of v	$v+1$
last-child(v)	The last child of v	$bwd\text{-}search(fwd\text{-}search(v,-1)-1,0)$
tree-depth(v)	The number of edges from root() to v	$excess(v)$
next-sibling(v)	The next sibling of v	$fwd\text{-}search(v,-1)+1$
previous-sibling(v)	The previous sibling of v	$bwd\text{-}search(v-1,0)$
parent(v)	The parent of v	$bwd\text{-}search(v,-2)+1$
subtree(v)	The size of the subtree rooted at v	(next-sibling(v)$-v$)/2
is-ancestor(v,u)	True if v is an ancestor of u	True iff $v \leq u \leq fwd\text{-}search(v,-1)$
level-ancestor(v,d)	The ancestor of v at tree-depth tree-depth(v)-d	$bwd\text{-}search(v,-d-1)+1$
lca(v,$u>v$)	The lowest common ancestor between v and u	parent($rmq(v,u)+1$), where $rmq(v,u)$ is $fwd\text{-}search(v-1, min\text{-}excess(v,u))$
leaves-num(v)	The number of leaves in the subtree rooted at v	$leaf\text{-}rank(fwd\text{-}search(v,-1)) - leaf\text{-}rank(v)$
children-num(v)	The number of children of v	$min\text{-}count(v, fwd\text{-}search(v,-1)-2)$
child(v, l)	The l-th child of v	$min\text{-}select(v, fwd\text{-}search(v,-1)-2, l)+1$
child-rank($v \neq root()$)	The position of v among its siblings	$min\text{-}count$(parent(v), v)

- $leaf\text{-}rank(i) = rank_{10}(i) = |\{1 \leq j \leq i-1 \mid P[j]=1 \wedge P[j+1]=0\}|$, the number of leaves up to position i.
- $leaf\text{-}select(j) = select_{10}(j) = \min(\{i \mid leaf\text{-}rank(i+1)=j\} \cup \{\infty\})$, the position of the open parenthesis of the j-th leaf.
- $fwd\text{-}search(i,d) = \min(\{j>i \mid excess(j) = excess(i)+d\} \cup \{\infty\})$, the least we have to move forward from i for the excess to grow by d units.
- $bwd\text{-}search(i,d) = \max(\{j<i \mid excess(j) = excess(i)+d\} \cup \{-\infty\})$, the least we have to move backward from i for the excess to grow by d units.
- $min\text{-}excess(i,j) = \min(\{excess(k) - excess(i-1) \mid i \leq k \leq j\} \cup \{\infty\})$, the minimum relative (as if P started at i) excess between i and j.
- $min\text{-}count(i,j) = |\{i \leq k \leq j \mid excess(k) - excess(i-1) = min\text{-}excess(i,j)\}|$, the number of times *min-excess* is reached between i and j.
- $min\text{-}select(i,j,l)$: Position of the l-th leftmost minimum excess between i and j, and ∞ if $l \notin [1, min\text{-}count(i,j)]$.

These primitives suffice to implement a large number of tree navigation operations (see Table 1.1), and can all be supported in constant time using $o(t)$ bits on top of P [55]. The key structure to achieve this efficiently is the *range-min-Max(rmM)*-tree, which helps to speed up the primitive operations on P.

1.5 Range min-Max Tree

The *range-min-Max(rmM)*-tree [55] is a complete binary tree whose nodes are augmented with excess information. Every leaf of the rmM-tree represents a consecutive block of P of size b, $P[i, i+b-1]$, and stores three fields: $e = excess(i+b-1) - excess(i-1)$, the excess of the block, $m = min\text{-}excess(i, i+b-1)$, the minimum excess reached by a prefix of the block, and $c = min\text{-}count(i, i+b-1)$, the number of prefixes reaching this minimum. Internal nodes represent blocks of P that are the concatenation of the representation of its two children and store the same three fields. The idea of these fields is that they summarize information on blocks of the input, which will be used to speed up the operations.

fwd-search(i, d) is solved for $d \leq 0$ ($d > 0$ requires an additional field in the nodes). The procedure initializes a counter of excess e to 0; if e reaches d the algorithm ends and returns the current position. The search is divided into 4 steps. In the first step, the leaf of the rmM-tree where $i+1$ falls is identified; this leaf is then scanned from $i+1$ onwards, increasing e by 1 every time a 1 is seen and decreasing e by 1 every time a 0 is seen. If e reaches d then the corresponding position is returned, otherwise we proceed to the next step. In the second step, we move through the leaf-to-root path, querying the m fields of the neighbor nodes from this path; more precisely, we identify the right neighbor u of the current node and check whether $e + u.m \leq d$, in which case we move to u and follow to the next step, otherwise we move to *parent*(v) and increase e by $u.e$ if v is a left child. If the traversal reaches *root*$()$ we return ∞. In the third step, we start with a node u such that the block represented by u covers the answer to the search; this step consists on a u-to-leaf path where we move down to the leaf containing the answer to the search. More precisely, in a node u whose left child is v and whose right child is w, we check whether $e + v.m \leq d$, in which case we move to v, otherwise we move to w and increase e by $v.e$. Finally, the fourth step consists on scanning the block represented by the leaf, until e reaches d, and returning the corresponding answer.

bwd-search(i, d) is also solved for $d \leq 0$. The procedure follows the same structure as the one described for *fwd-search*. The fundamental difference is that the scan is done right-to-left, so the description of the procedure flips, that is, in the first and fourth steps the scans are done right-to-left, in the second step we look for the field of the left neighbor, and in the third step we look for the field of the right child. Moreover, what matters for *bwd-search* is the maximum excess reached in the suffixes of blocks, which is exactly $e - m$, because the maximum suffix excess is reached exactly to the right of where the minimum prefix excess is reached.

min-excess(i, j) is similar to the searches, however, instead of reaching a particular value d with e, we store the minimum value reached by e in a variable m. In the first step we identify the leaf of the rmM-tree where i falls, then we scan the corresponding block from i to the minimum between the end of the block and j, and update e as before. If at some position e reaches a value less than m, we store in m that value. If j falls in a different leaf, we continue to the next step, otherwise we return m. In the second step, similarly to *fwd-search*, we follow the leaf-to-root path, checking whether the right neighbors u of nodes in the path contain j in the block they represent, if that is the case, we finish this step, otherwise we move to the parent of the current node and if the current node is a left child, we update m by $\min(m, e + u.m)$ and increase e by $u.e$. In the third step we start from a node u containing

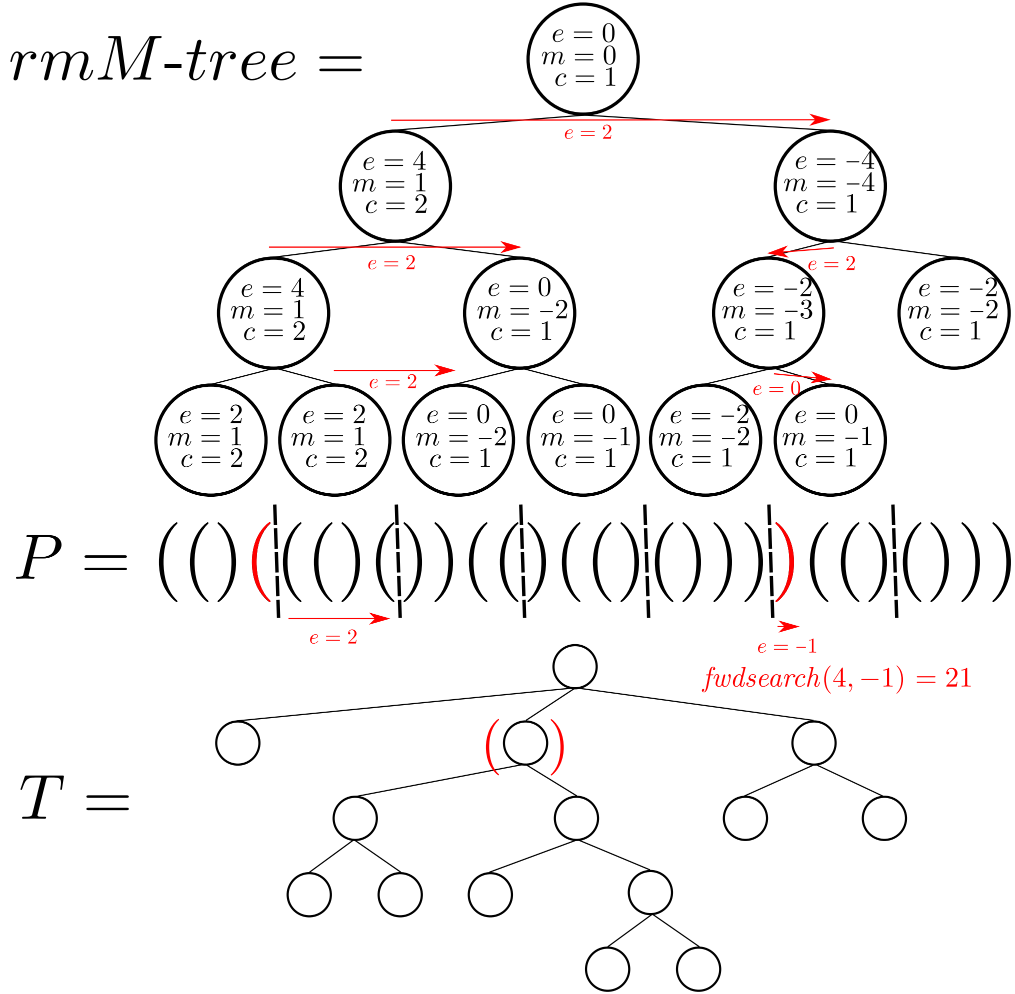


Figure 1.1: T is a tree, P the BP representation of its topology, and on top is built a simplistic version of the corresponding rmM-tree. The BP is divided into blocks of length 4. In red it is represented the *fwd-search* operation from the opening parenthesis of the second child of the root to its closing parenthesis. The four steps are represented, from left to top to down to right.

j in the block it represents, and descend to the corresponding leaf containing j. In a node u, whose left child is v and whose right child is w, if the block represented by v contains j we move to v, otherwise we move to w, update m by $\min(m, e + v.m)$, and increase e by $u.e$. Finally, in the fourth step we scan the leaf from the start until j, updating e as before, and if at some position e reaches a value less than m, we store that value in m, returning it at the end.

min-count(i, j) is similar to *min-excess* but we also query the c field of nodes, and carry an extra variable *count* containing the number of times m is reached, which will be the answer at the end. In the first and fourth steps, if the scans reach some e equal to m we increase *count* by one, and if we update m to a new value, we set *count* to 1. In the second and third steps, when we are updating m, if $m = e + v.m$ we increase *count* by $u.c$, otherwise if $m > e + v.m$, we set *count* to $v.c$.

min-select(i, j, t) first calls *min-excess*(i, j) and stores the answer in a variable m, then a process similar to *fwd-search* is done looking for the position $k \geq i$ of the t-th place where *excess*(k) $-$ *excess*$(i-1) = m$ (*fwd-search*$(i-1, m)$ would be the special case of *min-select*$(i, j, 1)$). In this case the steps stop when a variable *count* initialized at 0 reaches t. In particular, in the first and fourth steps, if at some point e reaches m, *count* increases by 1, and if *count* reaches t the corresponding position is returned. In the second step, we only stop the leaf-to-root traversal if $e + v.m = m$ and $count + v.c \geq t$, otherwise we also increase *count* by $v.c$ if the current node is a left child and $e + v.m = m$. In the third step, we only go to the left child v if $e + v.m = m$ and $count + v.c \geq t$, otherwise we also increase *count* by $v.c$ if $e + v.m = m$.

1.6 Suffix Array and Suffix Tree

The *suffix array* [46] $A[1, n]$ of T is a permutation of $[1, n]$ such that $A[i]$ is the starting position of the i-th suffix in increasing lexicographical order. One well-known functionality of this index is to *locate* the occurrences of a pattern $S[1, m]$ in T in $O(m \log n)$ character comparisons, which is possible 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 *longest-common-prefix (LCP) array* [46], $LCP[1, n]$, 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 longest common prefixes between lexicographically consecutive suffixes of $T[1, n]$. If we enhance the suffix array with the LCP array, *locate* is possible in time $O(m + \log n)$ [46] and if we add even more information, we can achieve $O(m)$ time [2].

The *suffix tree* [70, 47, 69] of 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, thus a classical/pointer-based implementation utilizes $\Theta(n \log n)$ bits. The suffix tree supports a set of operations (see Table 1.2) that suffices to solve a large number of problems in Stringology [3] and Bioinformatics [30]; for example, (approximate) pattern matching, pattern discovery, finding repeated substrings, computing matching statistics, computing maximal matches, and many others. Note that leaves descending from a suffix tree node span a range of suffixes in A and the string-depth of the *lowest common ancestor* between two consecutive suffix tree leaves is exactly its corresponding LCP entry.

1.7 Dictionary Compressors

There exists a well-studied [39, 24] weakness of the entropy model to capture repetitions (substrings of T that appears various times). In fact, if for example we concatenate T with itself it holds that $\mathcal{H}_k(TT) \geq \mathcal{H}_k(T)$, i.e., entropy is not sensitive to very long repetitions. In face of this weakness to capture the compressibility of repetitive sequences, there has been much interest in compression algorithms with the ability to capture this repetitiveness. The general idea of these algorithms consists in replacing substrings of T with references to a

Table 1.2: 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.

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 an 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

dictionary of strings; this is why they are called *dictionary compressors*. In this thesis we focus in two of them:

Straight Line Programs (SLPs). An SLP [37] is a context-free grammar, in Chomsky normal form [13] (only production rules of the form $A \rightarrow BC$ and $A \rightarrow a$ are allowed, where A, B, C represent non-terminals and a represents a terminal) that derives only the string T. Even though finding the size g^* of the smallest SLP deriving T is NP-complete [12], there exist efficient heuristics [56, 42] that produce small grammars. One of those that has gained popularity in the field of Compressed Data Structures is Re-Pair [42]. This is a linear time and space algorithm that builds an SLP by repeating the following: "find the most frequent pair ab of characters in T, create the rule $X \rightarrow ab$ and replace all ab occurrences in T by X". This procedure is repeated until no pair appears more than once, and finally the initial rule $S \rightarrow C$ is added, where $C = C[1, c]$ is a string representing the final result of T after replacing the most frequent pairs.

Lempel-Ziv compression (LZ). The family of Lempel-Ziv algorithms [43] divides T into repeated substrings called factors, and builds short factorizations of the text with them. LZ77 [72] and LZ78 [73] were the first two algorithms in the family, being the former a stronger version of the latter. In LZ77 the text is processed *left-to-right* and pointers of maximal substrings to previous occurrences are output. These maximal substrings are the factors of LZ77 and the number of them is known as z.

Both g^* and z are alternative measures of compressibility that better handle repetitive

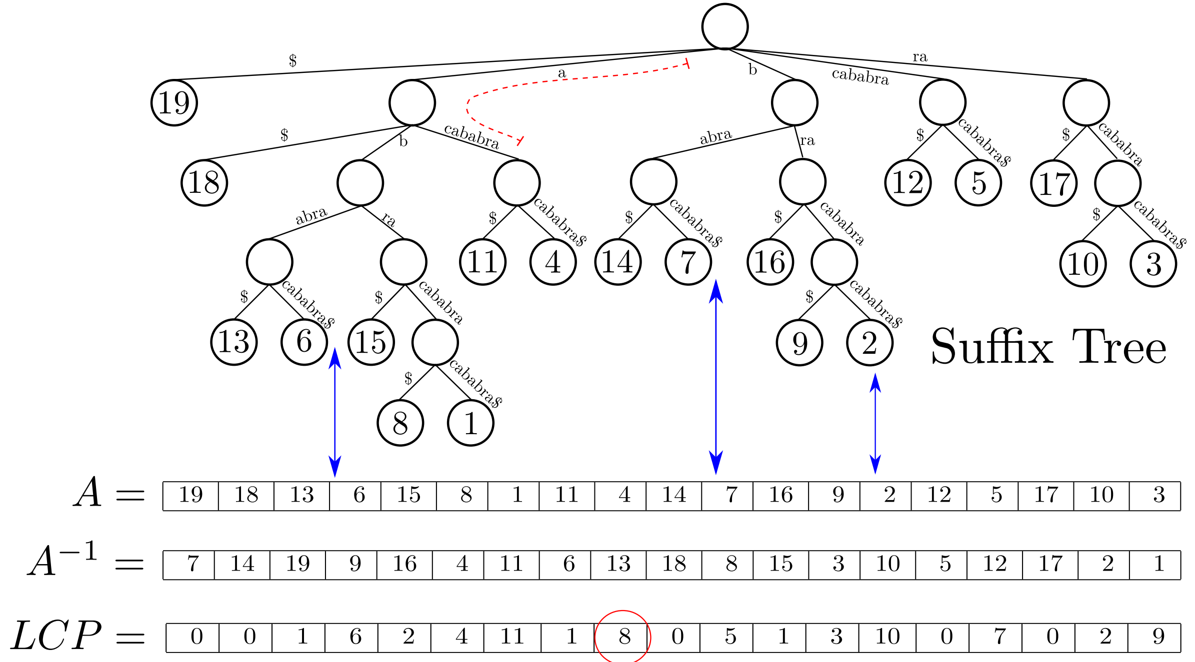


Figure 1.2: The suffix tree for the input text $T = abracababracabarbra\$$. The leaves of the tree are marked with the suffix they represent. We also show the corresponding suffix array A, its inverse A^{-1} and the LCP array. In blue it is represented the relation between the leaves of the suffix tree and the corresponding suffix array. In red it is represented the relation between the string-depth of the lowest common ancestor of two consecutive leaves, and its corresponding LCP entry.

sequences than entropy-based models. It is known [12, 34, 65] that $z \leq g^* = O(z \log n/z)$. Finally, it is worth mentioning that all the *dictionary compressors* aim to solve the same combinatorial problem, which is to find a small set of positions capturing all the substrings of T; such a set is called a *string attractor* [36].

1.8 Karp-Rabin Algorithm

Rabin and Karp proposed an algorithm [35] for the string matching problem, finding the occurrences of $S[1, m]$ in T. To do so they first compute the *fingerprint* of S, f_S, then run an *sliding window* of size m over T computing the *fingerprints* of each substring of size m of T, and wherever this *fingerprint* is equals to f_S they compare the substring with S and report the occurrence if this is the case. For the algorithm to work efficiently, a *rolling hashing* function is used to compute the *fingerprints*, that is, a hash function that allows constant time computation on the substrings of a *sliding window*.

Chapter 2

Related Work

In this chapter we present various results and structures in the context of the work done in this thesis. In general, they represent scientific contributions that aim to solve the same kind of problems we do or contributions we use to build on top of them.

2.1 RSA-Structures using Grammar Compression

The first attempt to use grammar compression to build up an RSA-structure was due to Navarro et al. [54]. They build the first RSA-structure for bitmaps, i.e., the first Grammar-based bitvector. For this, they grammar-compress B with Re-Pair and store fields attached to the non-terminals of the resulting grammar. For every non-terminal variable X they store $l(X)$ and $z(X)$, the length and the number of 0s in the string derived by X, respectively, and a uniform sampling on B storing p: the index in the initial rule C containing the sampled bit, o: the offset from the beginning of $C[p]$ to the sampled bit, and rnk: the number of 0s in the expansion of the initial rule before $C[p]$, that is, the $rank_0$ of the expansion of $C[1]...C[p-1]$. Using this augmented structure, they answer RSA queries by traversing the rules of the grammar in *S-to-terminal* paths[1]. Further, they use this representation to compress the bitvectors of the nodes in the wavelet tree and get the first Grammar compressed RSA-structure. However, they noticed that this approach is better suitable for tiny values of alphabet size σ because the repetitiveness of the bitmaps is quickly lost with the depth of the wavelet tree, so they finally proposed a version where the bitvector representation (grammar or statistically compressed) is chosen dynamically at construction time.

Inspired in the Grammar-based bitvector Ordóñez et al. [60] created GCC, a Grammar compressed RSA-structure for general sequences. They generalize the structure of Navarro et al. [54] by adding rank information for each of the σ symbols, so instead of $z(X)$ they store for every non-terminal X and every symbol a, $r_a(X)$, the number of as in the string derived by X, and instead of rnk in the sampling of B they store rnk_a for every symbol a. They tested two versions of GCC, the first doing a uniform sampling on B as described ($GCCN$)

[1] Which are *root-to-leaf* paths in the derivation tree.

and the second storing the sampling in the symbols of the initial rule ($GCCC$), resulting the latter more compact and slower than the former. As the space grows proportionally to σ, they recommend GCC for small alphabets only.

In broad terms, the operations are implemented as follows. For $rank_a(i)$ they first identify the rule $C[p] = X$ where the query falls and add the corresponding rnk_a to the result. Then they expand the rules $X \to YZ$ until reaching the terminal $T[i]$; when expanding they check $l(B)$ to decide whether the expansion continues to B or C, and in the case the expansion continues through C they add $r_a(B)$ to the result. Answering $select_a(i)$ is similar, but in the initial rule a binary search over the rnk_a values is run to find out the rule containing the answer. The expansion path is decided by looking at $r_a(B)$ as well.

Also, they generalize the use of GCC not only to wavelet trees but to wavelet matrices and *multi-way wavelet trees*, a variant of the *wavelet tree* where the alphabet is partitioned into $r > 2$ parts in each node. In this case they represent the partitions with a sequence over an alphabet of size r with RSA support (where GCC fits).

2.2 Compressed Suffix Arrays

A milestone in the area of Compact Data Structures was the emergence of Compressed Suffix Arrays (CSAs) [52], 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, $A^{-1}[j]$. Moreover, CSAs have been adapted to efficiently answer the queries *count(P)* (returning the number of occurrences of P as a substring in T), and *locate(P)* (returning the positions of these occurrences).

The most effective of these indexes can be classified into two groups. The first group [29, 66] takes advantage of properties of the Ψ function to compress it and samples A and A^{-1}, recovering the original values using its interplay with Ψ; for A a text-position regular sampling is stored, so if we want to recover a particular value from A, we apply Ψ iteratively until we get to a sampled position, and finally subtract the steps we advanced using Ψ. For A^{-1} a regular sampling is stored, and when we want to recover a particular value, we move to the nearest sample to the left, and apply Ψ iteratively as much as the distance to the sampled position. The indexes of the second group [21] are called FM-indexes. One of the most successful implementations [22] of these indexes base their functionality on answering *access* and *rank* queries on the *Burrows Wheeler Transform* (BWT) of the text T_{BWT}, a permutation of the characters of T such that $T_{BWT}[i] = T[A[i]-1]$ (and \$ if $A[i] = 1$), that is, the previous character of the suffixes in lexicographical order. The index is built using T_{BWT}, samplings in A and A^{-1}, and the function $LF(i) = \Psi^{-1}[i]$ (which is not stored but computed using *rank* and *access* on T_{BWT}). Both class of indexes achieve space bounded by its k-th order entropy, for $k < \log_\sigma n$, and take $O(\text{polylog } n)$ time to access A and A^{-1}.

In the scenario of repetitive sequences, 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*: *RLFMI* [44], which run-length compresses (keeping the run-heads only) T_{BWT} into T'_{BWT} and stores additional structures to translate the queries on T_{BWT}. It achieves $O(nH_k(T)\log\sigma)$ bits of space and retains the search complexities of the FM-index. The *RLCSA* [45] run-length compresses the runs of 1s in the differential encoding (Section 1.2 on page 5) of Ψ. To retrieve the original values, it stores a sampling on Ψ, using space proportional to the number of runs: r, and it also stores a sampling on A and A^{-1} using its interplay with Ψ to recover a particular value as explained before. More precisely, the index uses $O(r(2\log(n/r)+\log\sigma)(1+o(1)))$ bits and can access A in $O(s\log n)$ time using extra $O((n/s)\log n)$ bits for the sampling of A.

Finally, a relevant repetition-aware suffix array representation is the *Locally Compressed Suffix Array* (LCSA) [27]. They use Re-Pair to grammar-compress the differential encoding (recall Section 1.2 on page 5) of the suffix array (along with a sampling) and add a bitvector $L[1,n]$ indicating the positions in T where each symbol of the initial rule $S \rightarrow C[1]...C[c]$ starts. They justify this encoding on the fact that runs in Ψ show up as repetitions in the differential suffix array and a grammar compressor will capture this repetitiveness. The process to recover a suffix array value is as follows: first identify the nearest sample to its left, then map the query and the nearest sample to the rules containing them, say they are $C[j]$ and $C[i]$ ($i \leq j$) respectively, and then expand $C[i]...C[j]$ adding the differentials from the sampling to the query. They also note that it is not necessary to fully expand $C[j]$ to complete this task.

2.3 Compressed Suffix Trees

2.3.1 The First Compressed Suffix Tree

Sadakane [67] designed the first Compressed Suffix Tree (CST), on top of a CSA, using $|\mathrm{CSA}|+O(n)$ bits and solving all the suffix tree operations in time $O(\mathrm{polylog}\ n)$. He makes up a CST from three components: a CSA, for which he uses his own proposal [66] (a Ψ-based CSA); a BP representation of the suffix tree topology, using at most $4n+o(n)$ bits; and a compressed representation of LCP, $PLCP[i]=LCP[A^{-1}[i]]$ (i.e., the LCP array in text order) encoded in a bitvector $H[1,2n]$, which marks the positions $PLCP[i]+2i$, thus $PLCP[i]=\mathit{select}_1(i)-2i$. A recent implementation [58] of this index requires about 10 bits per character and takes a few microseconds per operation.

The topology operations are solved by the BP representation on the topology as explained in Section 1.4 on page 7. The rest of the operations is solved with the help of the CSA and the compressed LCP representation, as follows:

- letter(v,i) is solved by computing $lr \leftarrow \mathit{leaf\text{-}rank}(v)$, then $s \leftarrow \Psi^{i-1}[lr+1]$, and returning the first symbol of the s-th lexicographical suffix. This method is preferred to directly accessing $T[A[lr+1]+i-1]$ because in the CSA it is faster to use Ψ and query the first letter of the s-th suffix than using access in both A and T.

– To solve string-depth(v), we first ask whether v is a leaf or not (with is-leaf(v)). In the case v is a leaf we compute $lr \leftarrow$ *leaf-rank*(v) and return $n - A[lr+1]+1$. Otherwise we return $LCP[$*leaf-rank*(next-sibling($v+1$))$+1]$. To understand the last statement recall that suffix trees do not have nodes with exactly one child, so next-sibling($v+1$) takes us to the second child of v; accessing LCP of its *leaf-rank*$+1$ is asking for the length of the *longest common prefix* between the rightmost leaf of the first child and the leftmost leaf of the second child, which is exactly the length of the string represented by v.

– The suffix-link(v) operation is solved by first computing $l \leftarrow$ *leaf-rank*(v) $+ 1$, $r \leftarrow$ *leaf-rank*(*fwd-search*($v, -1$)) (the indexes of the leftmost and rightmost leaves of v), and then returning $lca(leaf\text{-}select(\Psi[l]), leaf\text{-}select(\Psi[r]))$. Note that l and r are the lexicographical position of the suffixes represented by the leaves, so $\Psi[l]$ and $\Psi[r]$ are the suffixes next to them, and its lca corresponds to the node we are looking for [67].

– string-ancestor(v,d) is done by a binary search on the ancestors of v, using string-depth to check the current node and level-ancestor to move through the ancestors.

– To solve child(v,c) we first compute $s \leftarrow$string-depth(v), then we linearly (using next-sibling) scan the children of v looking for the child ch with letter(ch,$s+1$) $= c$.

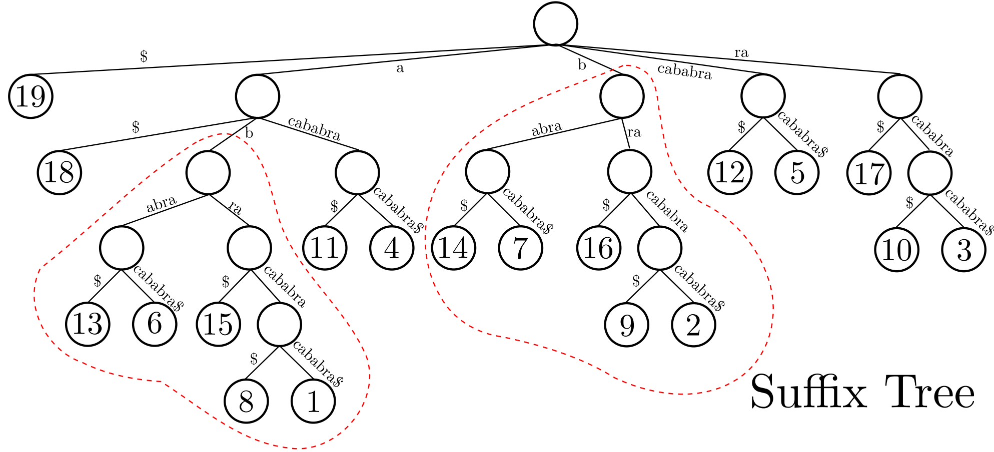
2.3.2 Fully-Compressed Suffix Tree (FCST)

Russo et al. [64] 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 [64] uses as little as 5 bits per character, but the operations take milliseconds, as slow as running in secondary storage. The sampling of the nodes is based on traversals of the suffix tree by following suffix links, this sampling combined with the operation lca implemented as a primitive operation manages to transform operation on real nodes into operation on sampled ones. The FM-index is used as the CSA. The LCP array is not stored (instead, a sampling of the string-depths replaces its functionality).

2.3.3 Faster Entropy-Bounded CST

Another approach [23] 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). Topology operations are solved by a combination of the interval representation of the nodes plus the use of the LCP primitives, for example, lca($v = [v_l, v_r], u = [u_l, u_r]$) = (if none of them is an ancestor of the other and $v_r < u_l$), first compute $k \leftarrow rmq(v_r + 1, u_l)$ and return $[psv(k), nsv(k) - 1]$. They also noted that bitvector H contains $2r$ runs, 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 CSTs.

$$T = abracababracababra\$$$



$$P = (()(()((()())(()(()())))(()()))((()())(()(()())))(()())(()(()())))$$

$$\Delta A = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 19 & -1 & -5 & -7 & 9 & -7 & -7 & 10 & -7 & 10 & -7 & 9 & -7 & -7 & 10 & -7 & 12 & -7 & -7 \\ \hline \end{array}$$

$$\Delta A^{-1} = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 7 & 7 & 5 & -10 & 7 & -12 & 7 & -5 & 7 & 5 & -10 & 7 & -12 & 7 & -5 & 7 & 5 & -15 & -1 \\ \hline \end{array}$$

$$\Delta LCP = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 1 & 5 & -4 & 2 & 7 & -10 & 7 & -8 & 5 & -4 & 2 & 7 & -10 & 7 & -7 & 2 & 7 \\ \hline \end{array}$$

$$H = 0000000000001111111111100110001110101$$

$$\Psi = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 7 & 1 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 5 & 6 & 17 & 18 & 19 & 3 & 4 & 2 & 8 & 9 \\ \hline \end{array}$$

$$BWT = arccbb\$rraaaaaaabbb$$

Figure 2.1: On top, the suffix tree for the input text $T = abracababracababra\$$. P is the BP representations of its topology. Prefix Δ indicates a differential encoding. We show this encoding for the suffix array A, its inverse A^{-1}, and the LCP array. We also show the bitvector H, the Ψ function and the BWT. We show repetitions in red and runs in blue for the structures.

2.4 Repetition-Aware Compressed Suffix Trees

2.4.1 The First Repetition-Aware CST

Abeliuk et. al [1] also presented the first CST for repetitive collections. They built on the approach of Section 2.3.3 [23], so they do not represent the tree topology. They use the RLCSA [45] 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 [42]) 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.

2.4.2 Grammar-Compressed Suffix Tree (GCST)

Navarro and Ordóñez [53] 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 1.4 on page 7. In general terms this augmentation of the non-terminal variables is an additional augmentation of the GCC RSA-structure, including fields similar to the rmM-tree, that is, fields indicating the minimum excess reached and information to solve *leaf-rank* and *leaf-select*. They use the RLCSA and, 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 three 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.

2.4.3 Other Approaches

It is worth mentioning the recent work by Farruggia et al. [19], who builds on Relative Lempel-Ziv [41] 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). Some recent theoretical work includes Gagie et al.'s [25] $O(r \log(n/r) \log n)$-bits CST using Run-Length Context-Free Grammars [57] on the LCP, and the Suffix Array, supporting most operations in $O(\log n)$ time. Although these grammars have theoretical guarantees of space, heuristics such as RePair work better in practice. Belazzougui and Cunial's [8] CST based on the CDAWG [9] (a minimized automaton that recognizes all the substrings of T). It also supports most operations in time $O(\log n)$. However, experiments [25] using CDAWG show that it uses significantly more space than other repetition-aware techniques.

2.5 Repetitive Datasets

In our experiments we use repetitive datasets obtained from the Repetitive Corpus of the *Pizza&Chili* platform[2]. We artificially create repetitive dna sequences: *dna0.001*, *dna0.01*, *dna0.1*, *dna1.0*, where *dnap* is built taking a 1MB prefix of the dna sequence in the corpus and coping it 100 times, where each copied base is mutated with probability $p/100$. The 1MB prefix of dna used as base for the construction was obtained from the Gutenberg Project[3] and contains 5 different elements as its alphabet. These sequences are compressible by p7zip[4], a

[2] `http://pizzachili.dcc.uchile.cl/repcorpus`
[3] http://www.gutenberg.org/
[4] `http://p7zip.sourceforge.net/`

Table 2.1: Summary table of the repetitive datasets. Length n is measured in millions (and rounded). Compression by `p7zip` is shown in bits per symbol (bps).

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	Human readable (documents)	0.009
`kernel`	259	160	Human readable (code)	0.205

practical Lempel-Ziv compressor, to 0.52%, 0.54%, 0.76% and 2.35%, respectively.

We also use two real dna sequences, `influenza`, a collection composed of 78,041 sequences of *Haemophilus Influenzae* (15 different symbols, 148MB, compressible by p7zip to 1.69%); and `escherichia`, a collection of dna sequences of different *Escherichia Coli* individuals (15 different symbols, 107MB, compressible by p7zip to 7.76%). Both sequences come from the NCBI[5].

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* (117 different symbols, 89MB, compressible by p7zip to 0.11%); and `kernel`, a set of 36 versions of the Linux Kernel (160 different symbols, 247MB, compressible by p7zip to 2.56%).

Finally, we use the BP representation (see Section 1.4 on page 7) of the topology of the underlying suffix trees (Section 1.6 on page 11) for the sequences presented before. We name such BPs by appending the suffix *.par* to the corresponding input sequence name.

2.6 Succinct Data Structure Library

To build our implementations we use intensively the *succinct data structure library* (sdsl) [26], which contains practical implementations of a myriad of *Compact Data Structures*, from arrays to suffix trees. Specifically, we use the following implementations:

- `int_vector`, Section 1.2 on page 5, an array using the logarithm of the maximum represented value of bits per element.
- `dac_vector_dp`, the DAC compressed array from Section 1.2 on page 5.
- `bit_vector`, Section 1.3.1 on page 6, an uncompressed bitvector with $O(1)$ rank/select support.

[5] `https://www.ncbi.nlm.nih.gov/`

– `rrr_vector`, the RRR bitvector from Section 1.3.1 on page 6.
– `sd_vector`, the Sparse bitvector from Section 1.3.1 on page 6.
– `cst_sada`, Section 2.3.1, the CST of Sadakane [67].
– `cst_sct3`, Section 2.3.3, an improvement of the ideas of Fischer et al. [23].
– `cst_fully`, Section 2.3.2, the CST of Russo et al. [64].

Chapter 3

Block Tree Improvements

In this chapter we present the improvements made to the main data structure of this thesis. Our contributions are both theoretical and experimental. They fix errors present in the literature and further enrich the structure itself. We divide the chapter into three main sections. In the first section we give a general description of the structure, in the second section we revisit the previous implementation of the structure, and in the third section we present our contributions.

3.1 Block Trees

The Block Tree [10] is a full r-ary tree (all nodes have r children except leaves) representing a sequence T. Every node v represents a substring $v.blk$ of T: the root represents the entire sequence and its children represent r consecutive blocks of its parent[1], where the first ($n \mod r$) blocks are of size $\lceil \frac{n}{r} \rceil$ and the rest of size $\lfloor \frac{n}{r} \rfloor$. A node v, representing $v.blk = T[i, i+b-1]$ 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 $T[i-b, i+b-1]$ and $T[i, i+2b-1]$ are not their leftmost occurrences in T, then the block is replaced by its leftmost occurrence in T: node v stores a pointer $v.ptr = u$ to the node u such that the first occurrence of $v.blk$ starts inside $u.blk = T[j, j+b-1]$, more precisely it occurs in $T[j+o, j+o+b-1]$. This offset inside $u.blk$ is stored at $v.\mathit{off} = o$. Node v is not considered at deeper levels. BackBlocks are also considered as leaves.

InternalBlock: Otherwise, the block is split into r blocks of size $\lceil \frac{b}{r} \rceil$ and $\lfloor \frac{b}{r} \rfloor$ with the same division rule applied to the root, handled in the next level by the children of v. The node v then stores a pointer to its children.

[1] Sometimes we will refer indistinctly to the node and the substring represented by that node.

Proposition 1 At each level l (where the root is at level 0) the blocks are of lengths either $\left\lceil \frac{n}{r^l} \right\rceil$ or $\left\lfloor \frac{n}{r^l} \right\rfloor$.

Proof. By induction in l; the base case corresponds to the division rule in the root node. Suppose the result holds until level l. Analyze the smallest block length generated by the division rule at level $l+1$, which is $s = \left\lfloor \frac{\left\lfloor \frac{n}{r^l} \right\rfloor}{r} \right\rfloor$. If we write $n = cr^l + k$, where $k \in [0, r^l - 1]$, then $s = \left\lfloor \frac{c}{r} \right\rfloor = \left\lfloor \frac{n}{r^{l+1}} \right\rfloor$, which can be obtained writing c as its decomposition modulo r. Analogously we can show that the largest block length is equal to $l = \left\lceil \frac{n}{r^{l+1}} \right\rceil$. Thus, all the blocks lengths at level $l+1$ are of size either $\left\lceil \frac{n}{r^{l+1}} \right\rceil$ or $\left\lfloor \frac{n}{r^{l+1}} \right\rfloor$. □

Choosing $mll = O(\log_\sigma n)$ and getting rid of the first $O(\log_r z)$ levels of the Block Tree we obtain the following result.

Proposition 2 [10] The Block Tree can reach height $h_{bt} = O\left(\log_r \frac{n \log \sigma}{z \log n}\right)$, where z is the number of phrases of the LZ77 parsing of T.

Proposition 3 Block Trees are well defined, that is, BackBlocks point to a well-defined block or pair of blocks in the same level, containing its leftmost occurrence. Even more, this block or pair of blocks are InternalBlocks.

Proof. Suppose that the leftmost occurrence of a BackBlock is $T[j, j+b-1]$. Then there must exist one or two blocks whose concatenation contains $T[j, j+b-1]$: a block intersecting a prefix of $T[j, j+b-1]$ and a block intersecting a suffix of $T[j, j+b-1]$ (these could be the same). Since they contain the leftmost occurrence of $T[j, j+b-1]$, both blocks must be InternalBlocks, because they do not satisfy the condition to be BackBlocks. □

Proposition 4 The Block Tree can be implemented using $O\left(zrh_{bt} \log n\right)$ bits of space.

Proof. As each node stores a constant number of pointers and numbers, it uses $O(\log n)$ bits. It suffices to show that each level contains $O(zr)$ nodes. For this, consider the blocks in a level and note that at most $3z$ blocks intersect or have a neighbor intersecting an LZ77 phrase. The rest are BackBlocks by definition, because they and their neighbors are completely contained in an LZ77 phrase. Then at most $3z$ block are InternalBlocks, thus we have at most $3zr = O(zr)$ blocks in the next level. □

To speed up the construction of the Block Tree, a top-down levelwise approach is applied. We first compute the Karp-Rabin signature of the blocks in the level and use them as keys in a *Hash-Table*. After that, we pass a rolling-hash (Karp-Rabin) on the input sequence, querying the *Hash-Table* and identifying in this way the first occurrences for every block in the level. Note that they do not specify how to enforce the conditions to create BackBlocks.

The Block Tree can answer $access(i)$ in $O(h_{bt})$ time recursively in a node v of size b as follows:

- If v is a LeafBlock, the block is explicitly stored in $v.blk$ and the i-th character is returned.
- If v is a BackBlock, the query is translated to block $v.ptr$ or its next block, according to i and $v.off$. If $i \leq b - v.off$ then the query is transformed into $access(i + v.off)$ in $v.ptr$, otherwise the query is transformed into $access(i - b + v.off)$ in the block next to $v.ptr$.
- If v is an InternalBlock, the query is translated to some of its children, according to i. The query is transformed into $access((i-1) \mod r + 1)$ in the $\left(\left\lfloor \frac{i-1}{r} \right\rfloor + 1\right)$-th child.

Since we do a constant amount of work per visited node, and considering Proposition 3, *access* takes time $O(h_{bt})$. Note that we assume that all blocks in a level are of equal size b; this was assumed on its original description [10], and will be assumed for the next description of *rank* and *select*. We will remove this assumption and give a complete description of the operations in Section 3.3.1.

3.1.1 Rank and Select

We can also give the Block Tree support for *rank* and *select* by adding the following fields:

- For every node v representing the block $v.blk = T[i, i+b-1]$ except the root:
  - $v.p\text{-}rank_c$ (prefix rank), which is the number of cs in the prefix of $parent(v).blk = T[j,k]$ preceding $v.blk$, that is $rank_c(i-1) - rank_c(j-1)$ in T.
- 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 $v.off = o$:
  - $fb\text{-}rank_c$ (first block rank), the number of cs in the prefix of O contained in $u.blk$ ($O \cap u.blk$, the first block spanned by O), that is, $rank_c(j+b-1) - rank_c(j+o-1)$ in T.
  - $pfb\text{-}rank_c$ (prefix first block rank), the number of cs in the prefix of $u.blk$ that precedes O ($u.blk - O$), that is, $rank_c(j+o-1) - rank_c(j-1)$ in T.

As we add at most three fields per $c \in \Sigma$, the space complexity of the Block Tree increases by a σ factor. Using them we can answer $rank_c(i)$ recursively in $O(h_{bt} + mll)$ time in a node v as follows:

- If v is a LeafBlock, we count the number of cs in $v.blk[1, i]$ and return it in time $O(mll)$.
- If v is a BackBlock the query is translated into $v.ptr$ or the next block according to i and $v.off$. If $i \leq b - v.off$ then the we answer $rank_c(i + v.off)$(in $v.ptr$) $-v.pfb\text{-}rank_c$, otherwise we return $rank_c(i - b + v.off)$ (in the block next to $v.ptr$) + $v.fb\text{-}rank_c$.
- If v is an InternalBlock the query is translated into some of their children according to i. Suppose w is the $\left(\left\lfloor \frac{i-1}{r} \right\rfloor + 1\right)$-th child of v, then we answer $rank_c((i-1) \mod r+1)$ (in w) + $w.p\text{-}rank_c$.

Select can be answered in $O(h_{bt})$ time by adding a constant-time predecessor data structure on top of the $p\text{-}rank_c$ fields, which increases the space [10]. Here we give an $O(rh_{bt} + mll)$

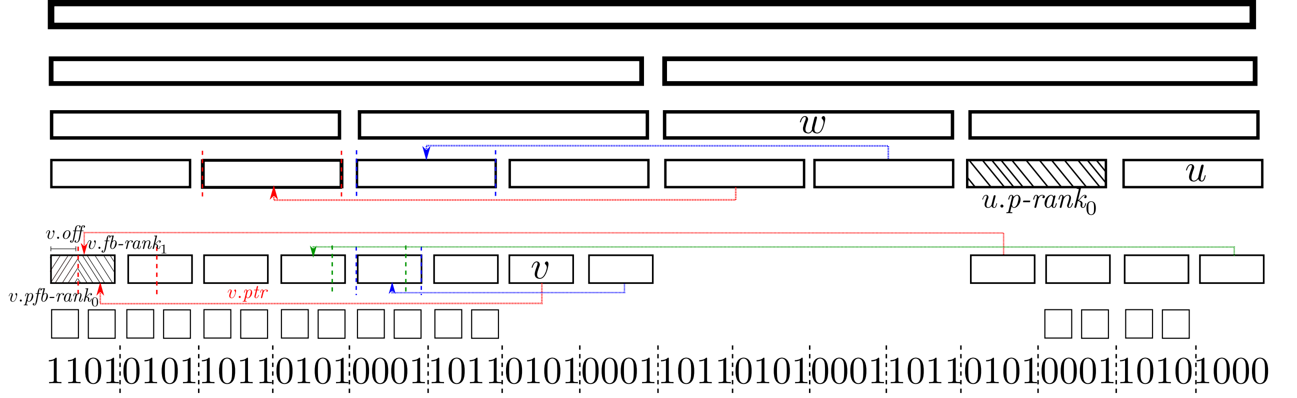


Figure 3.1: Block Tree for a bit-sequence representing a BP of a tree topology. Nodes have been put on top of the substrings they represent, and the child pointers were eliminated for clarity. For BackBlock v, some of its fields were represented, as for node u. Note that node w was not converted to a BackBlock although $w.blk$ is not a leftmost occurrence, which happens because its sibling contains a leftmost occurrence. On the other hand, the children of w are BackBlocks, since concatenated with their neighbors they do not contain leftmost occurrences.

time solution without including any further field. $select_c(j)$ is solved recursively in a node v as follows:

- If v is a LeafBlock, we look for the j-th symbol c in $v.blk$ and return its index or ∞ if no index is found in $O(mll)$ time.

- If v is a BackBlock, the query is translated into $v.ptr$ or its next block according to j and $v.fb\text{-}rank_c$. If $j \leq v.fb\text{-}rank_c$ then the we answer $select_c(j + v.pfb\text{-}rank_c)$ (in $v.ptr$) $-$ $v.off$, otherwise we return $select_c(j - v.fb\text{-}rank_c)$ (in the block next to $v.ptr$) $+b - v.off$.

- If v is an InternalBlock, the query is translated into some of its children according to j and the $p\text{-}rank_c$ fields of the children. We look for k-th child w such that $w.p\text{-}rank_c <$ $j \leq next\text{-}sibling(w).p\text{-}rank_c$ or *last-child(v)* is none is found. We then return $select_c(j - w.p\text{-}rank_c)$ (in w) $+(k-1)b/r$.

Note that in the previous description b/r is an integer number thanks to padding.

3.2 Previous Implementation of Block Trees

The first implementation of Block Trees appears in the thesis of Alberto Ordoñez [61]. This implementation does not use pointers, as naive tree implementations do, and instead it uses *Compressed Data Structures* to represent the tree. In practical terms, this reduces the space dramatically, even if the order is the same. Also, the implementation does not follow the theoretical proposal completely, but presents good results anyway.

3.2.1 Definition

There are several differences between the Block Tree described in the thesis of Ordóñez and the definition given before:

1. The condition for a node to be a BackBlock is weaker. In this case it suffices that a block has a previous occurrence to be a BackBlock.

2. At each level of the Block Tree, the representations of blocks are concatenated forming a new input sequence, thus making consecutive blocks that were not so in previous levels.

The second difference shows us that space could be reduced by considering not only text-consecutive neighbors in a level, but neighbors in the Block Tree. Also, although the first difference seems negligible, it has important consequences. Now BackBlocks can have pointers to their first occurrences, even though they could be a first occurrence when concatenated with one of their neighbors.

- Applying this condition is expected to yield more BackBlocks than applying the original condition, which could reduce space.
- The condition could eliminate first text-occurrences needed for blocks in lower levels (for example a first-text occurrence of a lower block overlaps the block and its neighbor, but the block is set as a BackBlock, because it is not its first occurrence in its level). Then our proof of Proposition 4 is no longer valid, since it requires that BackBlocks point to first-text occurrences[2].
- This weaker condition breaks the property from Proposition 3 that BackBlocks do not point back to BackBlocks and it does not ensure a cost proportional to h_{bt} for *access* nor the respective costs for *rank* and *select*. Moreover, if the first occurrence of a block overlaps the block itself, an operation on a block could cost as much as the length of that block.

3.2.2 Construction

The construction uses the same idea of top-down levelwise Karp-Rabin scan to find the first occurrences of blocks, however, a totally different criterion is used to form BackBlocks.

In a level, after arranging the blocks in the HashTable using its Karp-Rabin signature, a Karp-Rabin scan on the sequence is done. Suppose that in the scan we find an occurrence of a block $v.blk$ that overlaps the blocks $x.blk$ and $y.blk$ (or just $x.blk$).

- If $v = x$ or $v = y$, we set v as an InternalBlock and delete $v.blk$ from the HashTable.
- Otherwise, we set v as a BackBlock with $v.ptr = x$, set both x and y as InternalBlocks,

[2]We could bound the number of blocks by $3z_l r$, where z_l is the number of LZ77 phrases of the sequence considered in level l, but we do not know a relation between z and z_l.

and delete $v.blk$ from the HashTable.[3]

The first point avoids self-references and pointers to later blocks, because ultimately, $v.blk$ is removed from the table when we reach v in the scan. The second point ensures that BackBlocks point to InternalBlocks, because every time we form a link the pointed blocks are set to be InternalBlocks.

Note that when a block is set to be a BackBlock, x or y could have been BackBlocks, but they are turned into InternalBlocks. This design decision of which blocks are Internal-Blocks and which BackBlocks is different from what was described in the definitions given in Sections 3.1 and 3.2.1. Another difference between this implementation and the previous definitions is that it is assumed that all blocks in a level are of the same length. For this to work, they applied a padding technique which adds at most one extra block per level. The technique computes the number of leaves as $\lceil n/mll \rceil$, and the number of nodes at each upper level dividing by r and taking the ceil on the number of nodes in the previous level.

This construction builds a pointer-based structure that afterwards is transformed into a more compact representation explained below.

3.2.3 Compact Representation

To represent a Block Tree, for each level l is used:

- A bitvector bv_type_l, marking which nodes are InternalBlocks, and a constant-time *rank* implementation over it. This lets us index the nodes by their position in the level, which is possible because the number of children of nodes is always r. *Rank* is used to descend to the next level by a child. For example, if we are in the node whose level-index is p and we want to move to its i-th child, this is: $r \cdot rank_1(p-1)(\text{in } bv_type_l)+i$. *Select* would be required to go to the parent in the previous level, but the implemented operations only require going downwards by a child.
- To represent the offset and pointers of BackBlocks, a pair of arrays is used, off_l and ptr_l. While off_l stores $v.off$ for every BackBlock v in the level, ptr_l stores the index of $v.ptr$ among the BackBlocks of the current level.
- For the *rank/select* information, three arrays are used: $p\text{-}rank_l$, $pfb\text{-}rank_l$ and $fb\text{-}rank_l$, storing for every node v in the level $v.p\text{-}rank$, $v.pfb\text{-}rank$ and $v.fb\text{-}rank$, respectively. These arrays were stored using *DAC*s (Section 1.2 on page 5).
- The LeafBlocks are stored as the concatenation of their blocks in an array called *leaves-string*.

Note that BackBlocks own two indices: their position among blocks in the level and their position among BackBlocks in the level. The second one is obtained from the first with *rank* in bv_type_l, and is used to access arrays storing exclusive information of BackBlocks.

[3]There is a detail: it could be that two or more blocks have identical contents; in this case we delete $v.blk$ only after processing all these nodes.

3.3 Improvements

3.3.1 No Padding Block Tree

To build a Block Tree faithful to its original definition we do not use padding to ensure that all blocks in a level l are the same size, but instead work with blocks of two possible sizes, $\lceil \frac{n}{r^l} \rceil$ and $\lfloor \frac{n}{r^l} \rfloor$ (Proposition 1). These are called large and small blocks, respectively.

For this to work, we add the field $v.len$ to the nodes, which is their length. The compact representations of these fields are h_{bt} bitvectors. For each level l we store the bitvector bv_length_l, marking the large blocks in the level, and we give them constant-time rank support. Note that these bitvectors suffice to simulate the $v.len$ field because the length of the current block in an operation can be maintained as we move through the nodes.

We decided to keep all LeafBlocks in the same last level h_{bt}. For a level to be the last it must be true that the length of large blocks is $\leq mll$. As Ordoñez [61], we concatenate all the contents of LeafBlocks.

The RSA-operations are slightly modified; we take into account the $v.len$ field to find the next block in the recursive query and to relocate the query adequately. The recursive operations over a node v are now as follows:

access(i)

- If v is a LeafBlock, the block is explicitly stored in $v.blk$ and the i-th character is returned. In this case we use *rank* on $bv_length_{h_{bt}}$ to relocate the query on *leaves-string*.
- If v is a BackBlock the query is translated into $v.ptr$ or its next block according to i, $v.off$ and $v.ptr.len$. If $i \leq v.ptr.len - v.off$ then the query is transformed into $access(i + v.off)$ in $v.ptr$, otherwise the query is transformed into $access(i - v.ptr.len + v.off)$ in the block next to $v.ptr$.
- If v is an InternalBlock the query is translated into some of its children according to i and $v.len$, which can be computed in constant time.

rank$_c(i)$

- If v is a LeafBlock, we count the number of cs in $v.blk[1, i]$ and return it. In this case we use *rank* on the $bv_length_{h_{bt}}$ to relocate the query on *leaves-string*.
- If v is a BackBlock the query is translated into $v.ptr$ or its next block according to i, $v.off$ and $v.ptr.len$. If $i \leq v.ptr.len - v.off$ then the we answer $rank_c(i + v.off)$(in $v.ptr$) $-v.pfb\text{-}rank_c$, otherwise we return $rank_c(i - v.ptr.len + v.off)$ (in the block next to $v.ptr$) $+ v.fb\text{-}rank_c$.
- If v is an InternalBlock the query is translated into some of its children according to i and $v.len$, which can be computed in constant time, and the field $p\text{-}rank_c$ of this child is added to the result.

$\textbf{\textit{select}}_c(j)$

- If v is a LeafBlock, we look for the j-th symbol c in $v.blk$ and return its index or ∞ if no index is found. In this case we use *rank* on $bv_length_{h_{bt}}$ to relocate the result from *leaves-string*.
- If v is a BackBlock the query is translated into $v.ptr$ or its next block according to j and $v.\textit{fb-rank}_c$. If $j \leq v.\textit{fb-rank}_c$ then the we answer $\textit{select}_c(j + v.\textit{pfb-rank}_c)$(in $v.ptr$) $-v.\textit{off}$, otherwise we return $\textit{select}_c(j - v.\textit{fb-rank}_c)$ (in the block next to $v.ptr$) $+v.ptr.len - v.\textit{off}$.
- If v is an InternalBlock the query is translated into some of its children according to j and the $\textit{p-rank}_c$ of its children. We look for k-th child w such that $w.\textit{p-rank}_c < j \leq \textit{next-sibling}(w).\textit{p-rank}_c$ or *last-child(v)* is none is found. We then return $\textit{select}_c(j - w.\textit{p-rank}_c)$ (in w) $+(k-1)\left\lfloor \frac{v.len}{r} \right\rfloor + \min(k-1, v.len \mod r)$.

3.3.2 Compressed Compact Representation

We note that the compact representation proposed by Ordoñez [61] uses plain bitvectors and uncompressed arrays except for the *rank*/*select* fields, which are represented with DACs [11]. We will study the use of compressed representations of arrays and bitvectors. We call Block Tree the implementation using uncompressed arrays and plain bitvectors on all its components, and Compressed Block Tree to the implementation using DACs to represent the arrays and RRR-bitvectors [62].

3.3.3 Construction Algorithm

Another contribution consists on fixing the construction algorithm given in the Block Tree publication [10]. They claim that it suffices to run only one Karp-Rabin rolling hashing with window size equal to the block size, however they do not specify how the condition to be a BackBlock is checked. We propose a construction algorithm that runs two rolling hashes, one with window size equal to the block size and another with window size equal to twice the block size.

Our algorithm is also top-down levelwise. For each level, the following stages are carried out:

1. **Fill block HashTable:** Block-wise Karp-Rabin scan to put the nodes on a HashTable called *blocks*; the signature of the nodes is the Karp-Rabin signature of their blocks.

2. **Find first occurrences:** Sliding window Karp-Rabin scan, using window size equal to the block size. If we find a node v in *blocks* with block content equal to the current window, we set a pointer from v to the node or pair of nodes whose blocks overlap the occurrence, and remove v from the HashTable.

3. **Fill pair-block HashTable:** Pair-of-blocks-wise Karp-Rabin scan to put the consec-

utive pairs of nodes on a HashTable called *pairs*; the signature of the pair of blocks is the Karp-Rabin signature of their concatenation.

4. **Check BackBlock condition:** Sliding window Karp-Rabin scan, using window size equal to twice the block size. If we find a pair of consecutive nodes v_i, v_{i+1} in *pairs* with the same concatenation of blocks as the current window, we put a label *left* on v_i and a label *right* on v_{i+1}, as long as the window is different from $v_i v_{i+1}$. We then remove the pair of blocks from the HashTable.

5. **Final iteration:** We iterate over the blocks. If a node has both *left* and *right* labels then this block corresponds to a BackBlock and we have already a pointer to its leftmost occurrence, otherwise it corresponds to an InternalBlock and is expanded to the next level.

An important note about this construction algorithm is that when we say that window scans have size equal to the block size and twice the block size, it is a simplification, since blocks in a level can be one of two lengths: b or $b+1$ (as stated in Proposition 1). Thus, our window scans refer to two windows scans with window size b and $b+1$ and to three window scans with window size $2b, 2b+1$, or $2b+2$.

3.3.4 Elimination of First Levels

We decided not to (compute and) use z to remove the first $O(\log_r z)$ levels as mentioned in Proposition 2, but instead, after constructing the Block Tree, we remove all the top levels that do not have BackBlocks, being the first level of the tree the first having some BackBlock.

A problem that arises when removing first levels, not addressed in the Block Tree publication, is how to identify the first queried block in an operation. For this to work we store σ arrays $global_prefix_rank_c$, which contain for every node in the first level the number of cs appearing in the blocks preceding it in the level.

For $select_c$ we just do a binary search on $global_prefix_rank_c$ to identify the first queried block. $Access(i)$ and $rank_c(i)$ are easy in padding versions: just divide $i-1$ by the length of the blocks in the first level and add 1, and in the case of $rank_c$ incorporate the corresponding $global_prefix_rank_c$ to the result. However, when working without padding the previous approach could bring errors, because blocks are not all of the same length in a level. To solve the problem, we apply a *guess & verify* approach: if the lengths in a level are $b+1$ and b, first, we *guess* $\left\lfloor \frac{i-1}{b+1} \right\rfloor + 1$ (which is a lower bound for the entry block), then we use *rank* on bv_length_1 to verify this guess. If this is not the block, we identify the error (in text positions) and divide it by $(b+1)$, guessing the answer again.

3.3.5 Pruning

We note that in the definition of Block Tree the condition to be a BackBlock is conservative, that is, it forces InternalBlocks that could have been BackBlocks without any problem. For

example, we could have set a block as an InternalBlock, which is not its first occurrence in T but it contains a first occurrence overlapping some of its neighbors, yet there is no node pointing to some descendant of this node. Node w from Figure 3.1 shows a good example of this.

From this observation we define an *unnecessary expansion* as a block that is not its first occurrence in T, holding one of those:

- It is a LeafBlock or a BackBlock. We call these blocks leaves.
- It is an InternalBlock, no node points to it, and its children are unnecessary expansions.

We would like to eliminate such expansions, i.e., change these InternalBlocks to BackBlocks. For this, we note that, to avoid dependency issues we need to eliminate these expansions in a *postorder right-to-left* traversal of the Block Tree. Moreover, when analyzing a block, it is enough to check if its children are all leaves, because if they were unnecessary expansions they would have been already processed in the traversal and turned into BackBlocks.

This removal of unnecessary expansions is called *pruning*. To implement pruning, we add a counter of the number of blocks pointing to each node, which increases every time a link is created and decreases every time a link is removed. This counter indicates whether a node is pointed by some other node, what happens when its value is greater than 0.

An interesting property of pruning is that if we apply it to a completely expanded Block Tree, where all nodes are InternalBlocks and LeafBlocks, we get the same tree as if we had applied it to the original definition of Block Tree.

Proposition 5 Let T_{bt} be a Block Tree as in the definition given in Section 3.1, and T_{exp} a completely expanded Block Tree as explained before. Both trees with the same input sequence T, arity r, and max length of leaves mll. Then, applying pruning on T_{bt} gives the same result than applying pruning on T_{exp}.

Proof. Assume we run the algorithms in parallel, that is, one node per step (we consider a parallel step valid even if one of the nodes exists in T_{exp} but not in T_{bt}). We will show by induction that at any step the previous nodes left by pruning are the same in both partial results of the algorithms, which implies that the final results of both executions are the same.

In principle both algorithms have not visited any node, so the property holds by vacuity. Suppose that we are in a step processing a node v and all the previous nodes left are the same in both trees. If $v \notin T_{bt}$, it means that the concatenation of $v.blk$ with each of its neighbors is not a leftmost occurrence, nor is any substring of these concatenations. Then $v.blk$ is not a leftmost occurrence, and there are no blocks pointing to v. Therefore, v is removed by the pruning, preserving our inductive hypothesis. If $v \in T_{bt}$, the action that pruning will take depends only on whether $v.blk$ is a leftmost occurrence, which is equals in both trees, and on the children and nodes pointing to v, which are the same by inductive hypothesis, since they were processed before in the right-to-left postorder traversal of the pruning. Then our hypothesis is preserved. □

This property could save us construction time, since the construction reduces to finding first occurrences and then apply pruning, but it requires much more space.

3.3.6 New Sets of Rank/Select Fields

We note that the fields used to store the *rank/select* information are redundant, for example, if a $v.ptr$ node is not a last child we have that $v.\mathit{pfb\text{-}rank}_c = (\mathit{next\text{-}sibling}(v.ptr).\mathit{p\text{-}rank}_c - v.ptr.\mathit{p\text{-}rank}_c) - v.\mathit{fb\text{-}rank}_c$. From this observation we propose four sets of fields that are alternatives to the original proposal. Although they do not improve the space complexity, they could be important in practical implementations:

Rank set: $v.\mathit{fb\text{-}rank}_c$ and $v.\mathit{rank}_c$, which stores the numbers of cs in the representation of the block ($v.\mathit{pfb\text{-}rank}_c$ can be inferred from these). This set reduces space because we get rid of $v.\mathit{pfb\text{-}rank}_c$, and numbers in $v.\mathit{rank}_c$ are smaller (the maximum of them) than numbers in $v.\mathit{p\text{-}rank}_c$. However, *rank* requires $O(r)$ time to descend to a child, requiring $O(rh_{bt})$ time in total.

Cumulative set: $v.\mathit{fb\text{-}rank}_c$ and $v.\mathit{cum\text{-}rank}_c$, which is the number of cs from its first sibling to the block $v.blk$ itself (inclusive). Note that when $v.ptr$ is not a first child, $v.\mathit{pfb\text{-}rank}_c = (v.ptr.\mathit{cum\text{-}rank}_c - \mathit{previous\text{-}sibling}(v.ptr).\mathit{cum\text{-}rank}_c) - v.\mathit{fb\text{-}rank}_c$, and $v.\mathit{pfb\text{-}rank}_c = v.ptr.\mathit{cum\text{-}rank}_c - v.\mathit{fb\text{-}rank}_c$ when $v.ptr$ is a first child. This set of fields keeps constant time for descending to a child and gets rid of $v.\mathit{pfb\text{-}rank}_c$, but the numbers it stores are larger.

Dynamic set: $v.\mathit{fb\text{-}rank}_c$ and $v.\mathit{dyn\text{-}rank}_c$, which stores $v.\mathit{p\text{-}rank}_c$ if v is not a first child and $u.\mathit{rank}_c$, where u is the previous node in the level, when v is a first child. The first node in the level stores the rank_c of the last node. This redistribution of fields is possible because we are indexing the Block Tree nodes by its level index on its compact representation. This set of fields keeps constant time for descending to a child, gets rid of the $v.\mathit{pfb\text{-}rank}_c$ field, and has number similar to those in the original set of fields.

NoBack set: Only $v.\mathit{rank}_c$. $v.\mathit{fb\text{-}rank}_c$ is computed recursively when needed by an operation. Note that in this case we do not ensure time proportional to h_{bt}, but space is reduced.

We call Prefix set to the original set of fields.

3.3.7 New Heuristics

We develop new heuristics in the construction of Block Trees. Similar to the heuristics proposed by Ordoñez [61], our heuristics differ from the definition in the algorithms used to decide whether a node is a BackBlock or not.

Below we name and detail each of these heuristics, including the Block Tree definition and the heuristics presented by Ordoñez. All versions include the improvement of eliminating the

first levels explained in Section 3.3.4.

bt corresponds to the original definition given in Section 3.1, built with the construction algorithm from Section 3.3.3 and applying afterwards the pruning algorithm explained in Section 3.3.5.

no-clean-bt is the same as *bt*, but without applying the pruning algorithm, that is, the Block Tree described on its original publication [10].

pruning-c-bt is the same as *bt*, but we relax the definition of unnecessary expansion. We allow a pointed block to be a BackBlock but only if there is no chain of BackBlocks of length $\geq c$ pointing to the block, where c is a parameter. Note that this increases the complexity of the operations by a factor of c, because in the worst case we have to take a chain of c BackBlocks to descend to the next level.

heuristic-concatenate-bt corresponds to the Block Tree described in the thesis of Ordoñez but not implemented in that work; this was explained in Section 3.2.1. It finds leftmost occurrences in a level and points back to them without considering stronger conditions.

heuristic-bt is the same as *heuristic-concatenate-bt* but not concatenating the blocks in each level.

liberal-heuristic-bt is the same as *heuristic-bt* except that we expand the blocks that point to themselves. Note that in this heuristic BackBlocks can still point to other BackBlocks.

conservative-heuristic-bt is the Block Tree produced by the construction algorithm described in the thesis of Ordoñez and explained in Section 3.2.2.

other-conservative-heuristic-bt is like *conservative-heuristic-bt*, but when setting a block as a BackBlock we further check if any of the pointed blocks is a BackBlock, in which case we do not put the link. Note that in this case the heuristic decides not to set a possible BackBlock in exchange for not transforming pointed blocks into InternalBlocks.

greedy-conservative-heuristic-bt is like *conservative-heuristic-bt*, but when setting a block as a BackBlock we further check if it is "greedily" convenient to do it in terms of space (minimizing the number of blocks transformed into InternalBlocks).

back-front-bt and other-back-front-bt modify the scan for finding first occurrences. We now allow occurrences to appear to the right of the block itself. The rest is analogous to *conservative-heuristic-bt* and *other-conservative-heuristic-bt*, respectively, to avoid loops in a level.

Besides, we put a suffix *-no-pad* to the name when referring to some version combined with using the techniques explained in Section 3.3.1 to remove padding. Also, we put a *-compressed* suffix when referring to some version combined with using compressed components to build its compact representation as explained in Section 3.3.2.

3.4 Experimental Results

3.4.1 Setup and Datasets

Our experiments ran on an isolated Intel(R) Xeon(R) CPU E5620 @ 2.40GHz with 96GB of RAM and 10MB of L3 cache. The operating system is GNU/Linux, Debian 2, with kernel 4.9.0-6-amd64. The implementations use a single thread and all of them are coded in C++ and use the *sdsl* library (see Section 2.6 on page 20) for their internal components. The compiler is gcc version 6.3.0, with -O9 optimization flag set.

We use the datasets *dna*{*0.001, 1.0*}, *escherichia*, *influenza*, *einstein* and *kernel* described in Section 2.5 on page 19; we also use the suffix tree topologies *dna0.001.par* and *dna0.01.par* to test the behavior of Block Trees on repetitive topologies, which we will use in next chapters.

3.4.2 Experiments and Structures

We designed exhaustive experiments for each of the three operations, *access*, *rank* and *select*. In case of *access* we access every position in the input sequence (in a random order to avoid locality issues). For *rank*, we query on every position the letter located in that position, in a random order too. Finally, for *select* we select each of the characters in random order from 1 to the number of the characters in the string.

We ran *access* experiments for all the Block Tree versions of Section 3.3.7 on its padding versions, and on *bt-no-pad* and *bt-compressed* to check the effect of the corresponding applied techniques. We also ran *rank* and *select* using its different sets of fields on *bt*, and *rank* and *select* on *bt-no-pad* and *bt-compressed* but only using the *Rank set* of fields.

For every structure we vary the arity $r \in \{2, 4, 8\}$ and the max length of a string represented by a LeafBlock $mll \in \{4, 8, 16, 32, 64, 128\}$, and take the Pareto-optimal points from the space$\times$time map.

3.4.3 Comparisons and Analysis

We show graphs of Block Trees comparing the effects of different aspects. The graphs are in 2D, where the x-coordinate represents the space in bps (bits per symbol) and the y-coordinate is the time per operation in microseconds; the input and operation analyzed are indicated in the title of every graph.

Figures 3.2, 3.3 and 3.4 show the effect of implementing a Block Tree version faithful to the theoretical proposal (see Section 3.3.1), and another using the padding technique to keep all the blocks in a level of the same length. Padding is Pareto-optimal in *access* and *rank*, while in *select* they present a similar behavior. This difference is explained in time by the additional verifications and algorithms used to solve the operations, and in space by the additional bitvector *bv_length* used to mark the length of blocks in a level. This behavior appears in all the implemented versions, so from this point the versions we show implement the padding technique.

In Figures 3.5, 3.6 and 3.7, we compare the effect of using compressed structures for the internal components of Block Trees (see Section 3.3.2). The graphs show how the use of compressed components reduces the time performance typically by an order of magnitude, while offering a small gain in space (no more than 35%). We also observe that this gain in space is accentuated when the input sequence is not very repetitive and when compressing the rank/select fields. Given the small space improvement versus the big time penalty we decided to use plain components.

In Figure 3.8, we compare the effect of using the pruning improvement (see Section 3.3.5) to further reduce the space of Block Tree. The graphs show that pruning obtains a noticeable space improvement without a major time penalty. This improvement is bigger when the input is not very repetitive even more highlights when dealing with suffix tree topologies. Hence, we add the pruning technique to the canonical Block Tree.

Figure 3.9 shows the effect of allowing chains of BackBlocks of length up to c (see Section 3.3.7). In this case $c = 1$ corresponds to *bt*; we also consider $c \in \{2, 4, 8, 16\}$ for *pruning-c-bt*. We note that for all our inputs set $c = 16$ is equivalent to allow chains of arbitrary length, that is, *pruning-16-bt* = *pruning-*∞*-bt*. The evolution with c quickly converges to *pruning-*∞*-bt*, typically using $c = 2$ gets very close to its smallest version.

In Figure 3.10, we compare the performance of Block Trees proposed by Ordoñez [61], *heuristic-bt* and *conservative-heuristic-bt* (see Section 3.3.7) against our *bt*. Note that *bt* and *conservative-heuristic-bt* present a very similar behavior, while *heuristic-bt* uses less space at a small cost in time performance. This last point is relevant because it shows that the worst-case scenarios for *heuristic-bt* (BackBlocks pointing to BackBlock or even themselves) are very unlikely to occur.

Figure 3.11 shows the effect of concatenating the blocks in each level (*heuristic-concatenate-bt*) and of forbidding self-pointers in BackBlocks (*liberal-heuristic-bt*) on *heuristic-bt*. Note that the studied improvements are imperceptible, which means that there is no significant gain on concatenating blocks and confirms the unlikeliness of worst-case scenarios.

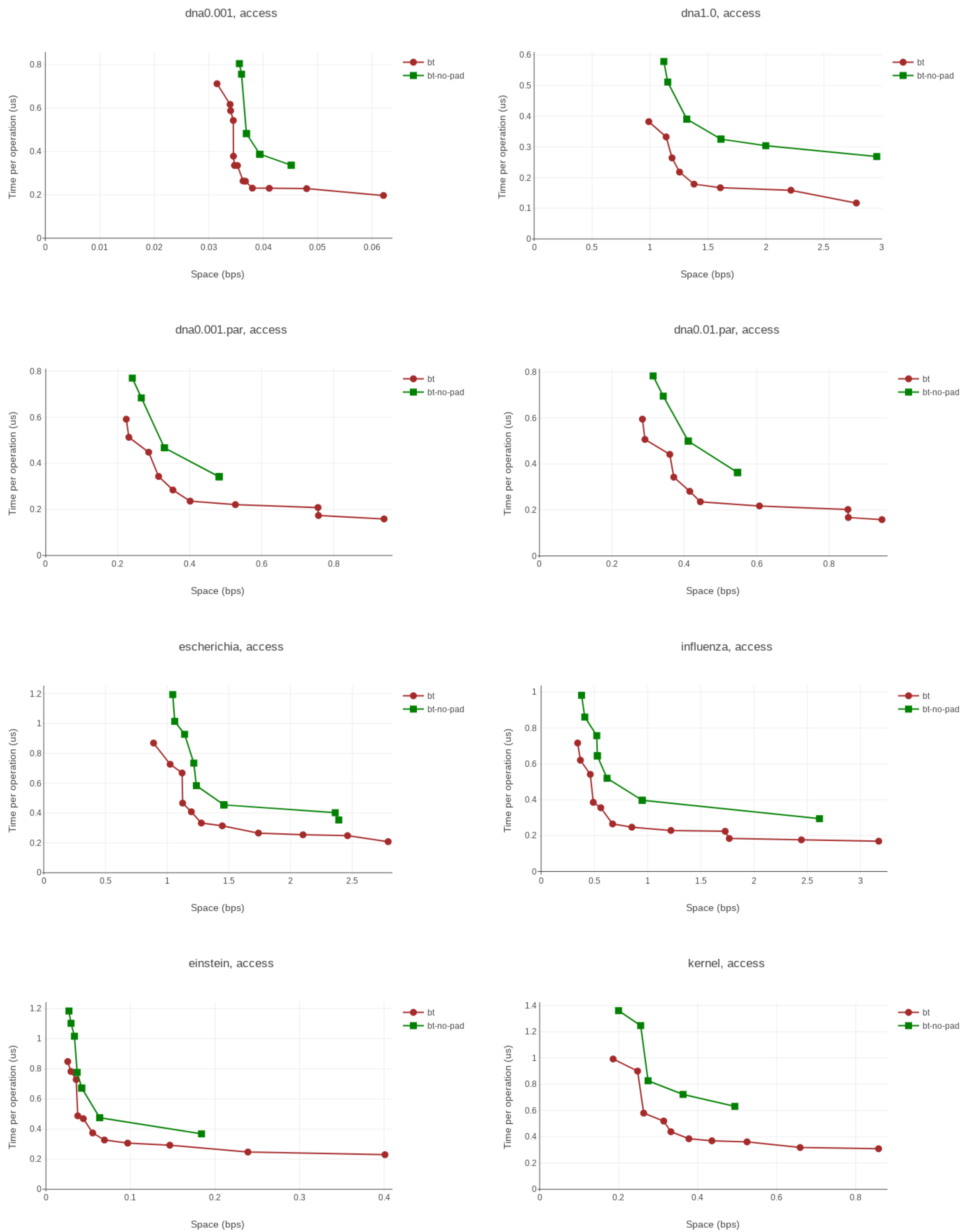


Figure 3.2: Performance of *access* in Block Trees with and without padding. The y-axis is time in microseconds in log-scale.

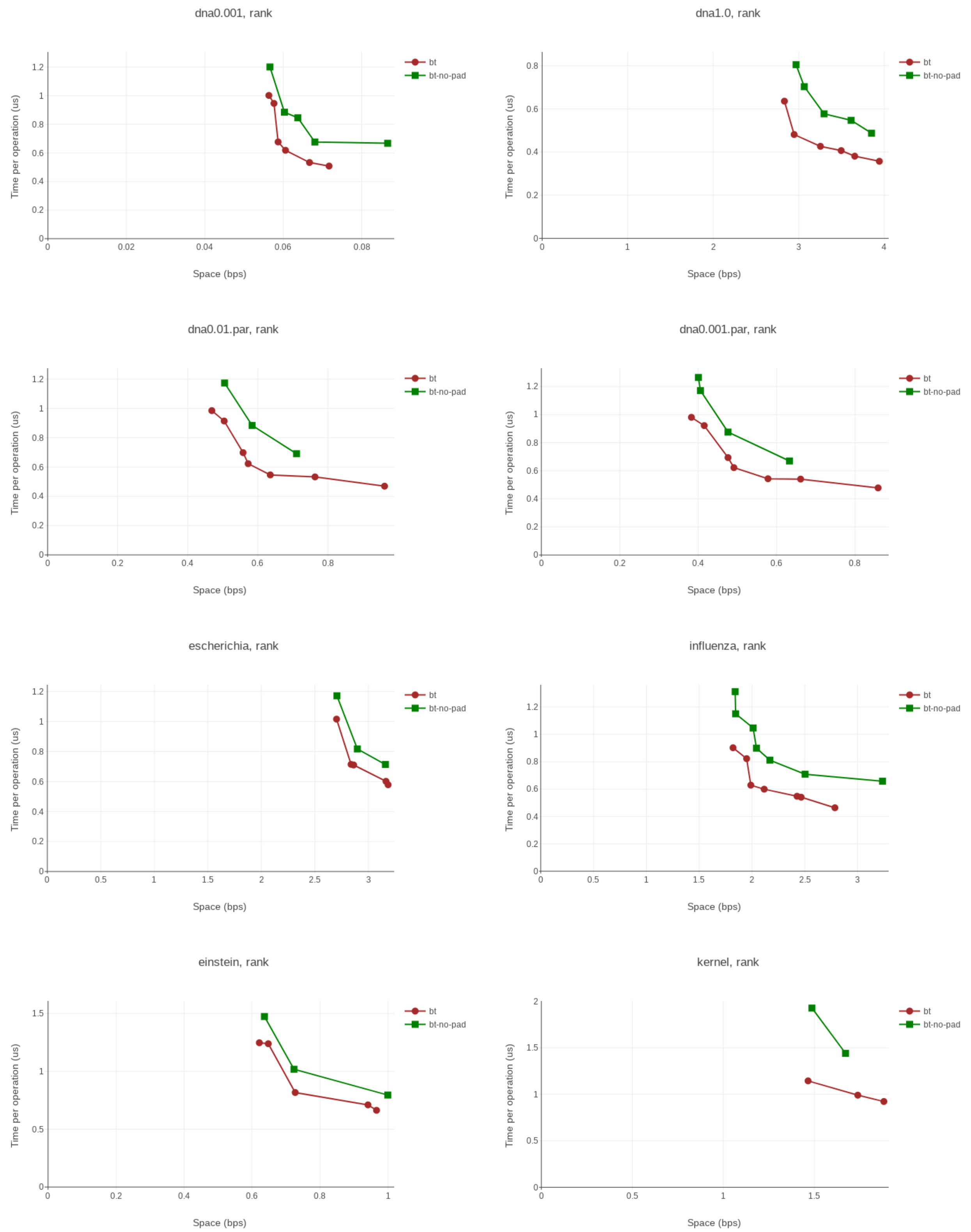


Figure 3.3: Performance of *rank* in Block Trees with and without padding. The y-axis is time in microseconds in log-scale.

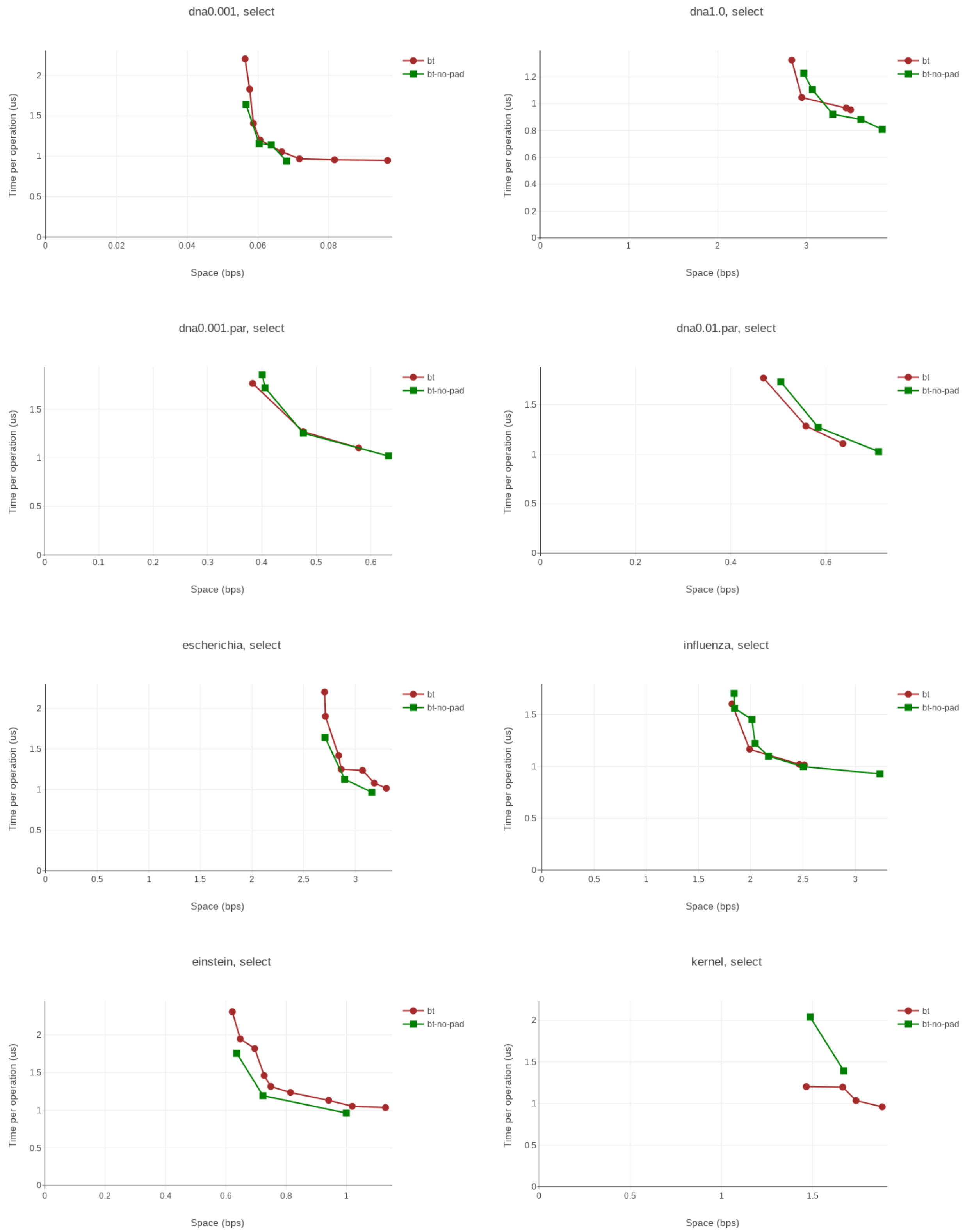


Figure 3.4: Performance of *select* in Block Trees with and without padding. The y-axis is time in microseconds in log-scale.

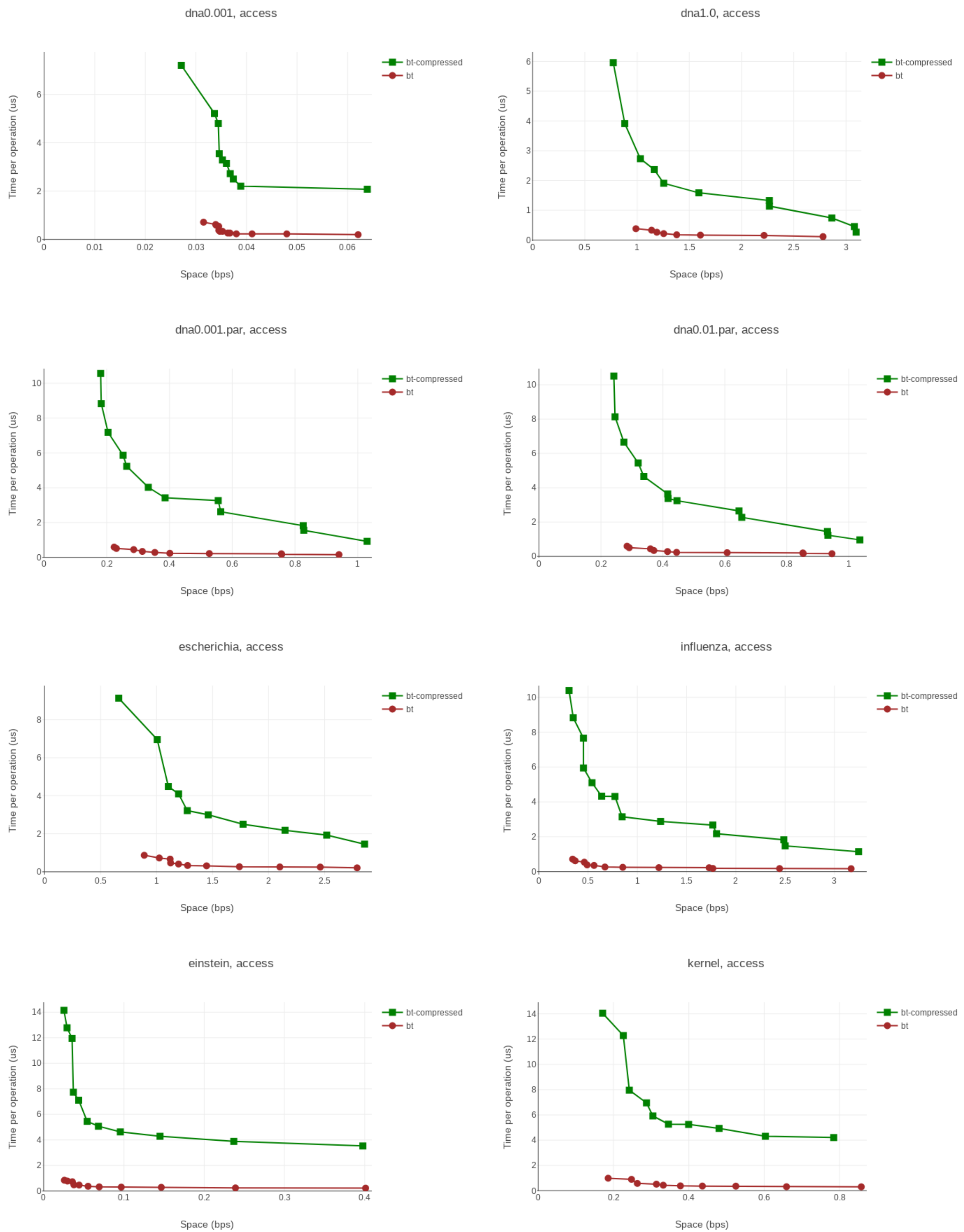


Figure 3.5: Performance of *access* in Block Trees using plain and compressed components. The y-axis is time in microseconds in log-scale.

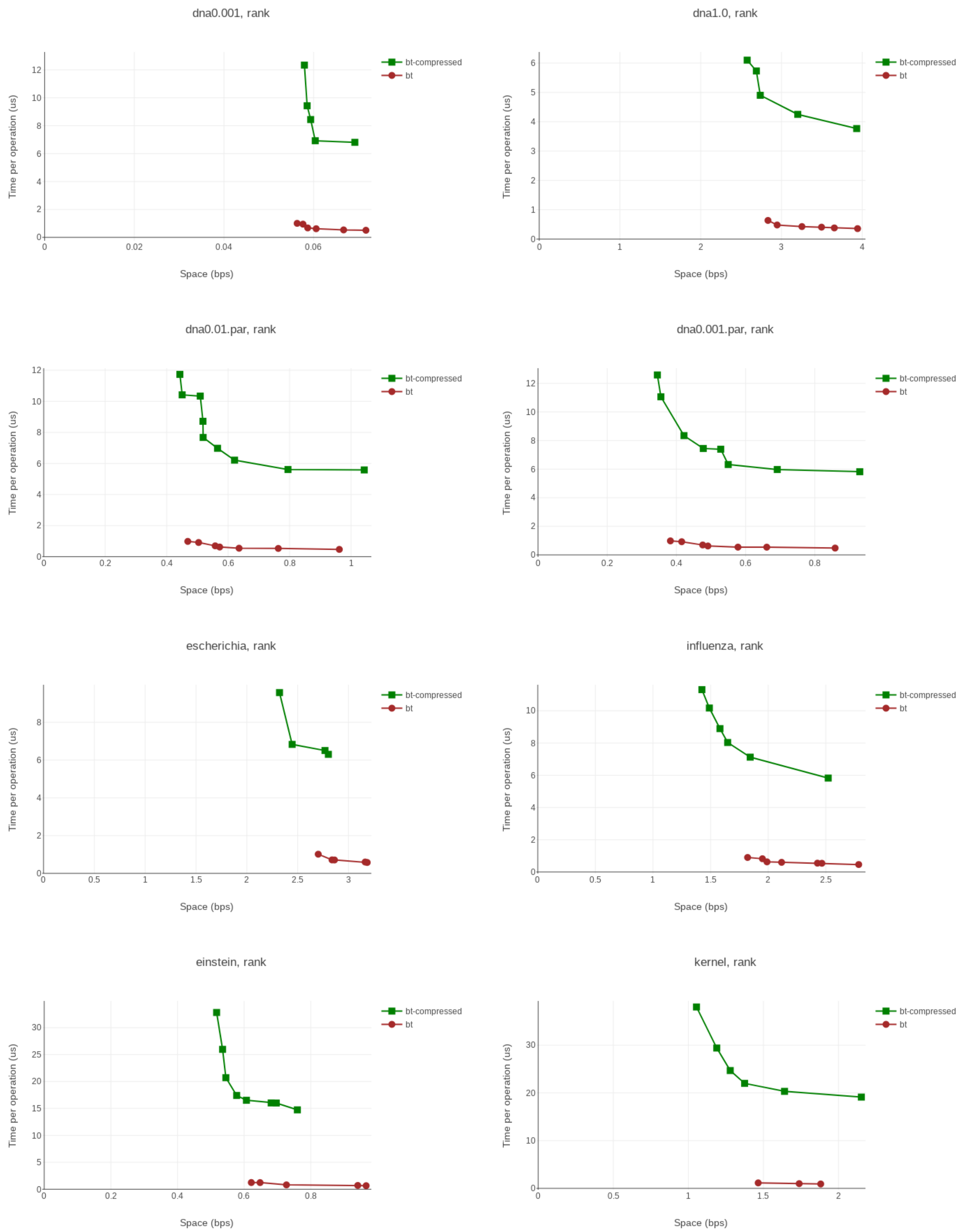


Figure 3.6: Performance of *rank* in Block Trees using plain and compressed components. The y-axis is time in microseconds in log-scale.

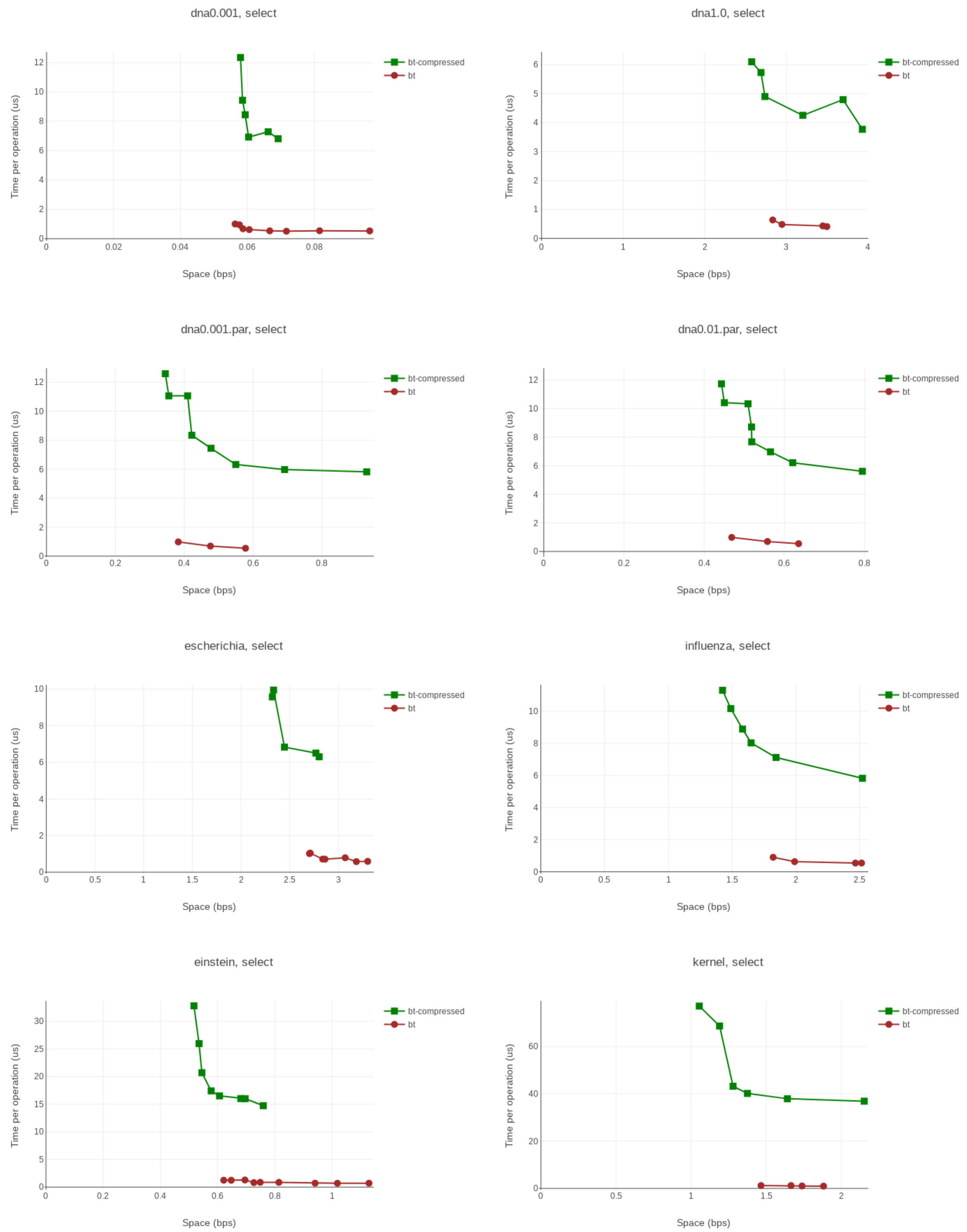


Figure 3.7: Performance of *select* in Block Trees using plain and compressed components. The y-axis is time in microseconds in log-scale.

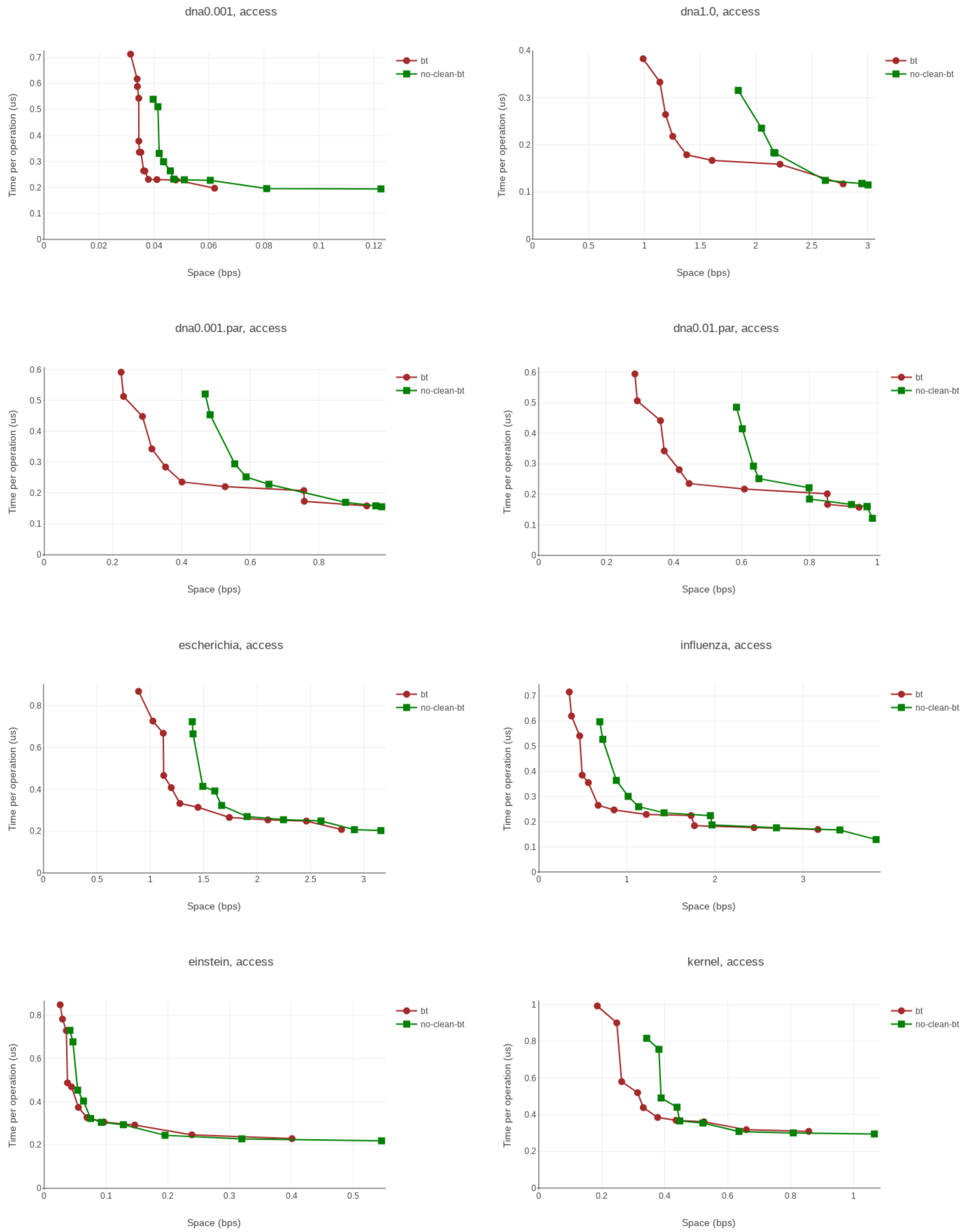


Figure 3.8: Performance of *access* in Block Trees using the pruning algorithm and not using it. The y-axis is time in microseconds in log-scale.

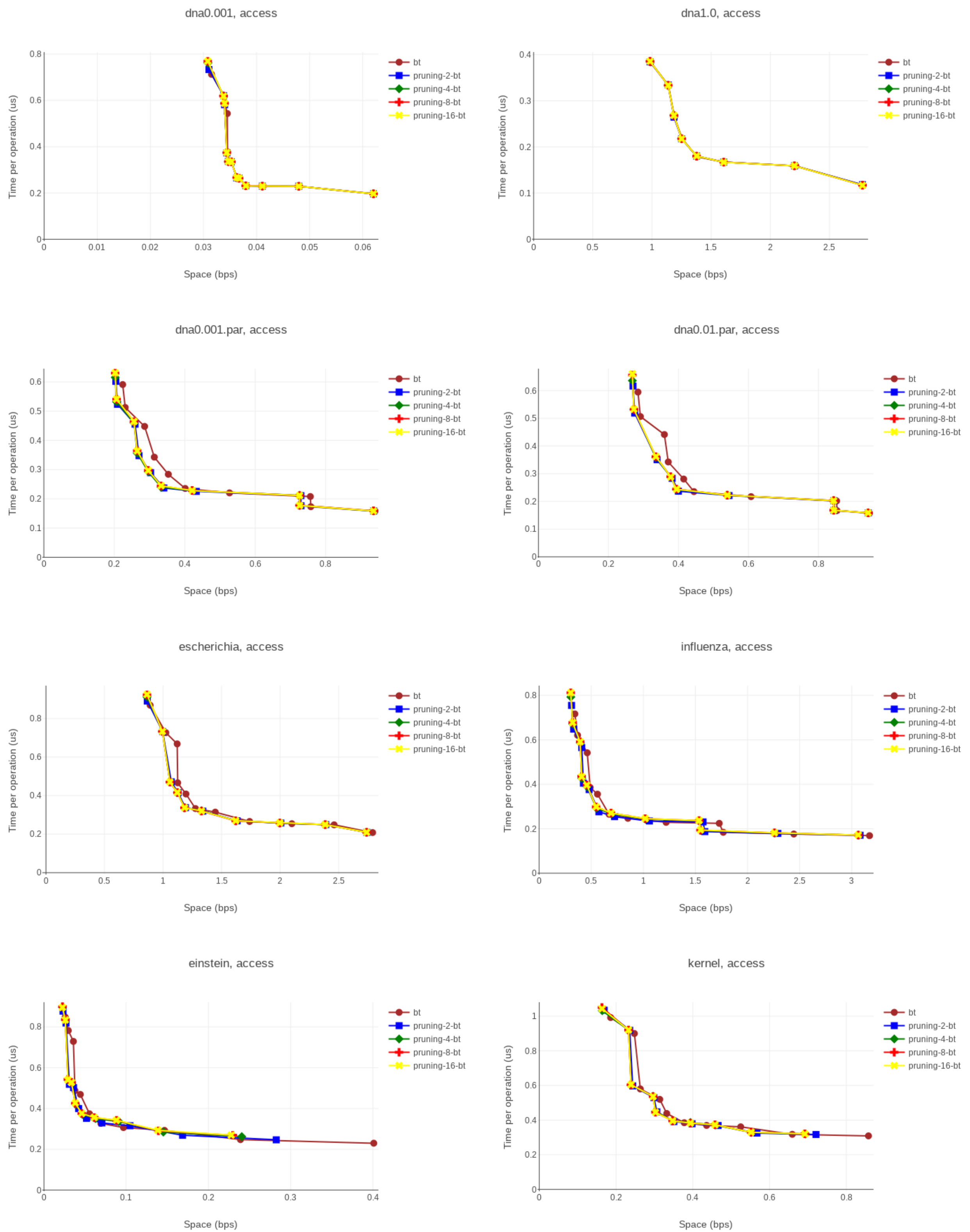


Figure 3.9: Performance of *access* in *pruning-c-bt* for $c \in \{1, 2, 4, 8, 16\}$. The y-axis is time in microseconds in log-scale.

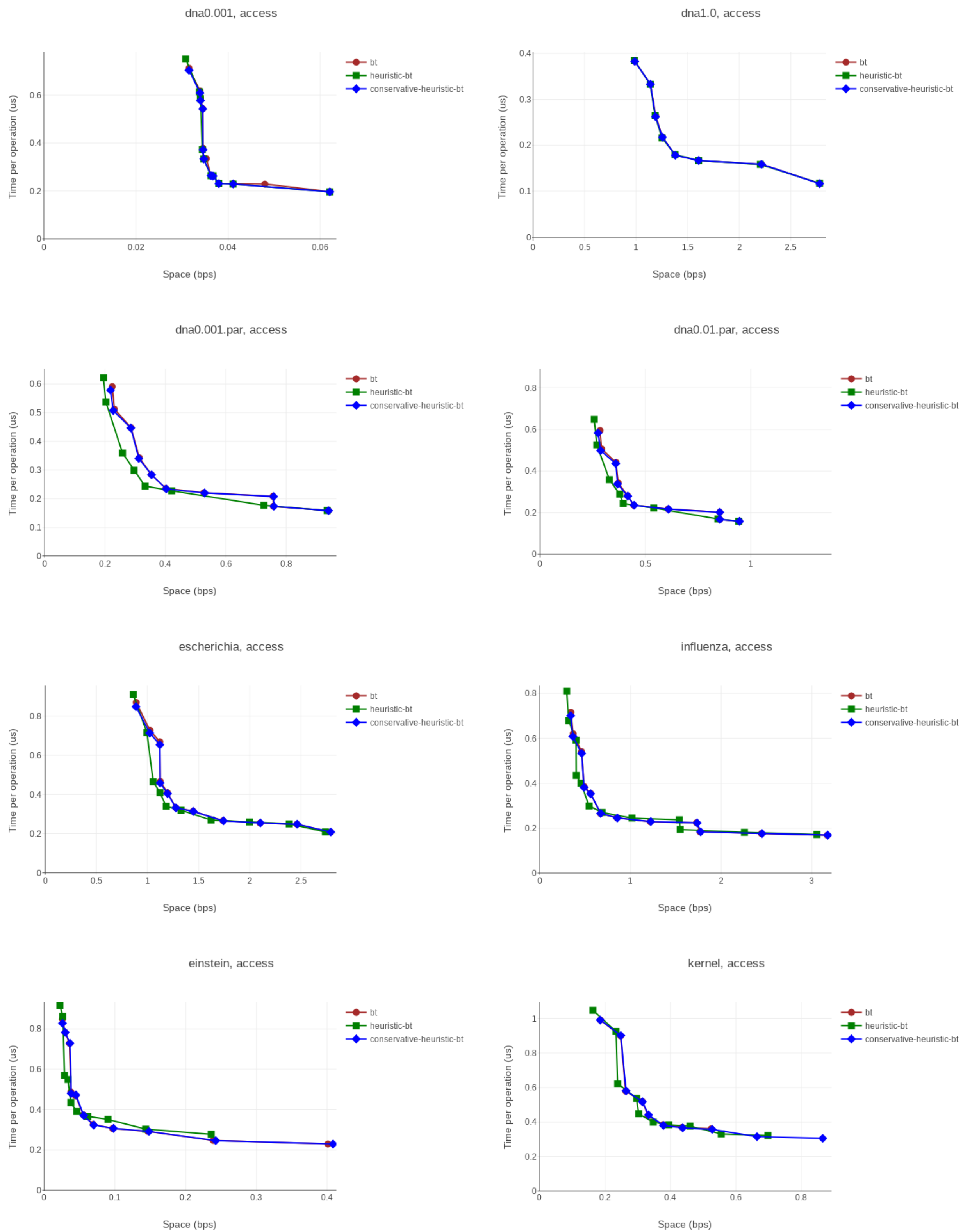


Figure 3.10: Performance of *access* in Block Trees proposed in the thesis of Ordoñez and our Block Tree. The y-axis is time in microseconds in log-scale.

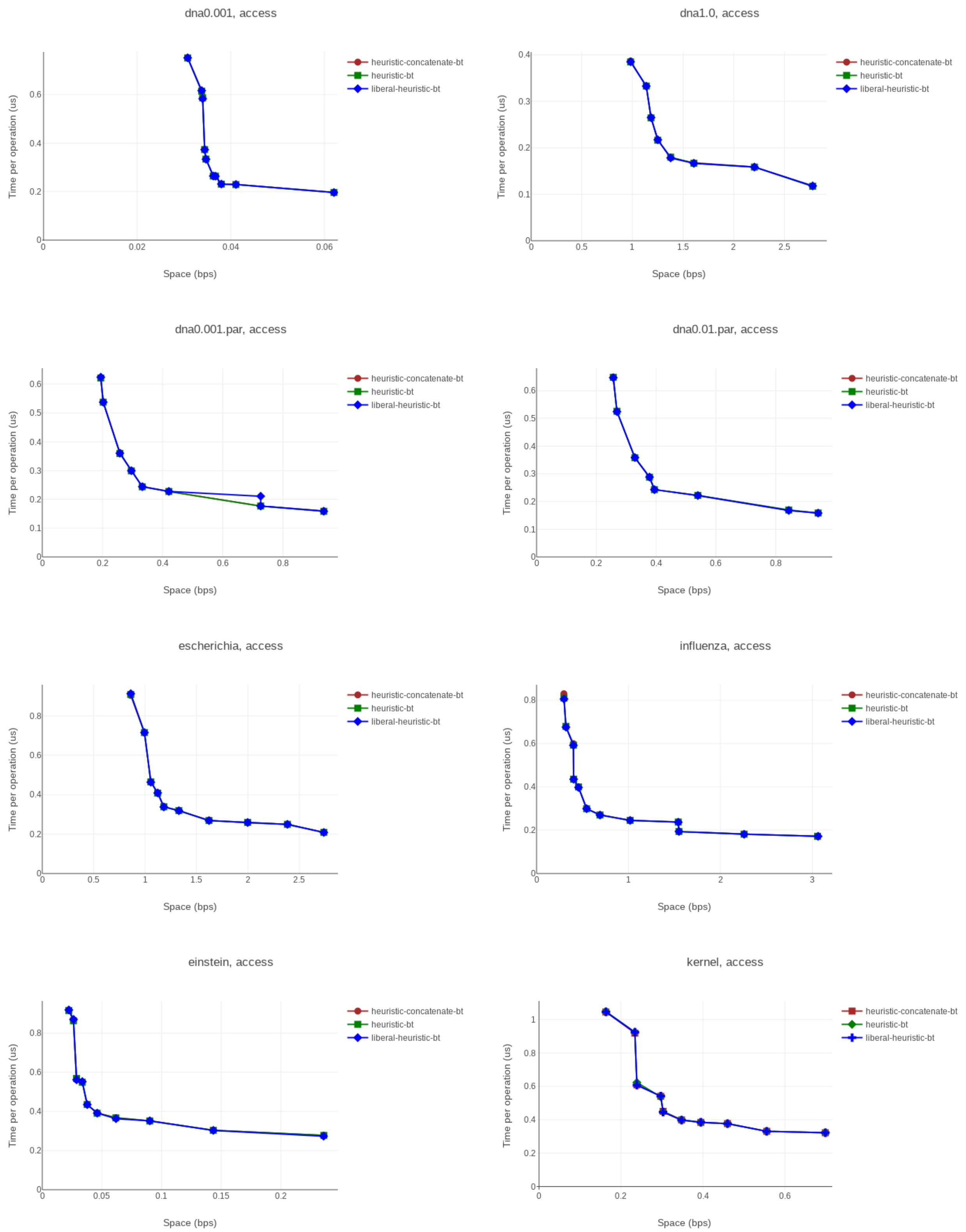


Figure 3.11: Performance of *access* in Block Tree variants of the *heuristic-bt*. The y-axis is time in microseconds in log-scale.

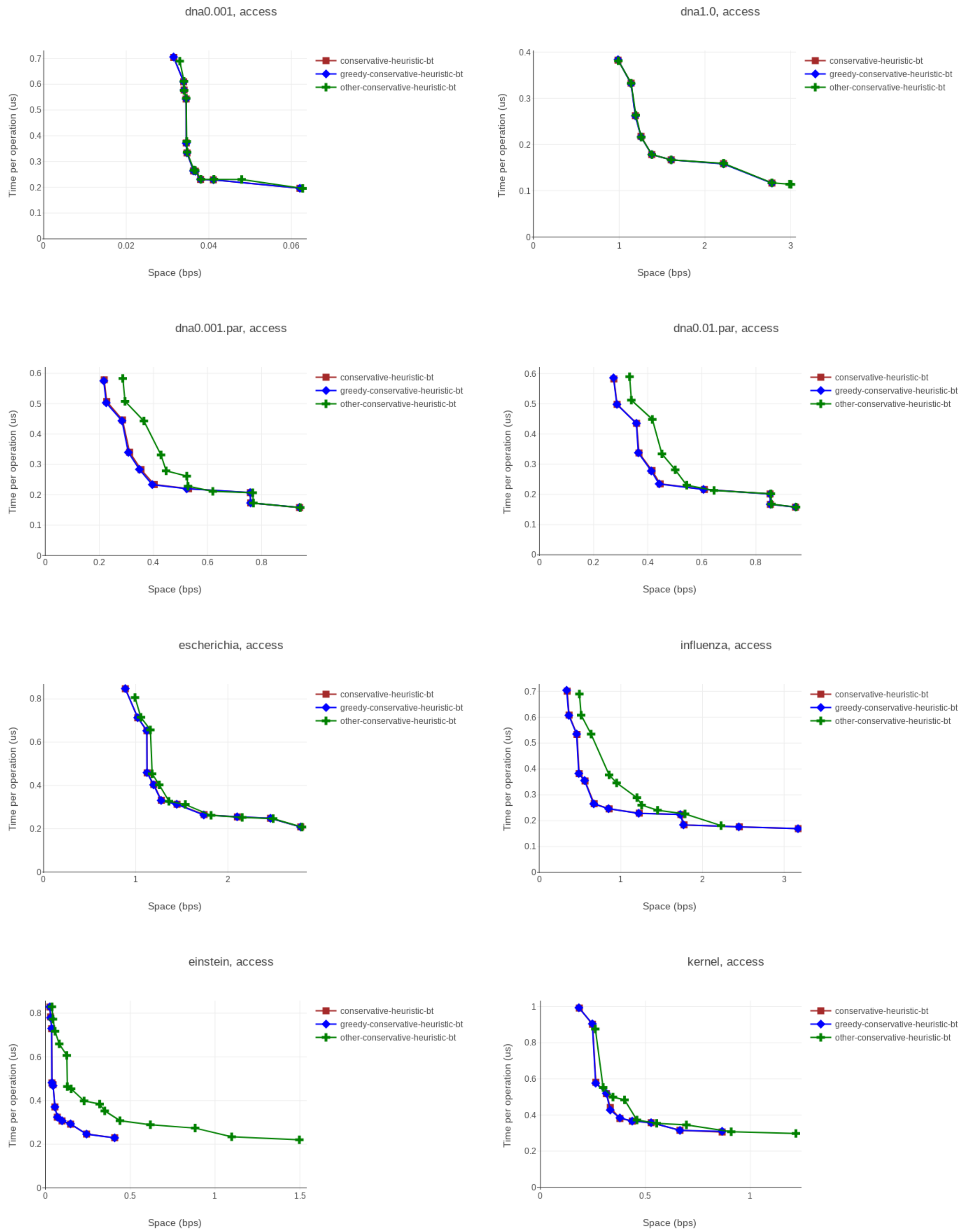


Figure 3.12: Performance of *access* in Block Trees using different criteria to form back pointers. The y-axis is time in microseconds in log-scale.

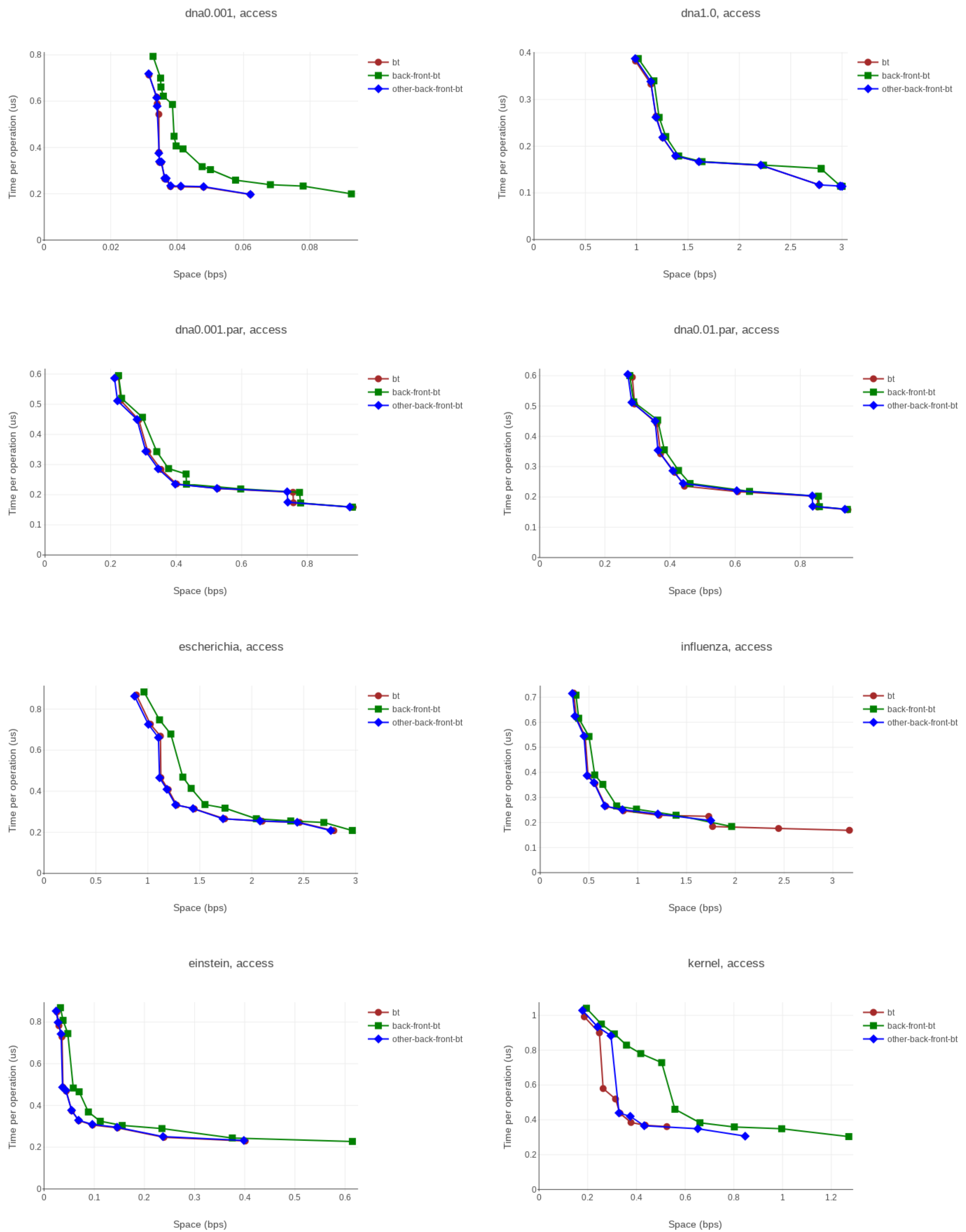


Figure 3.13: Performance of *access* in Block Trees allowing front pointers and our Block Tree. The y-axis is time in microseconds in log-scale.

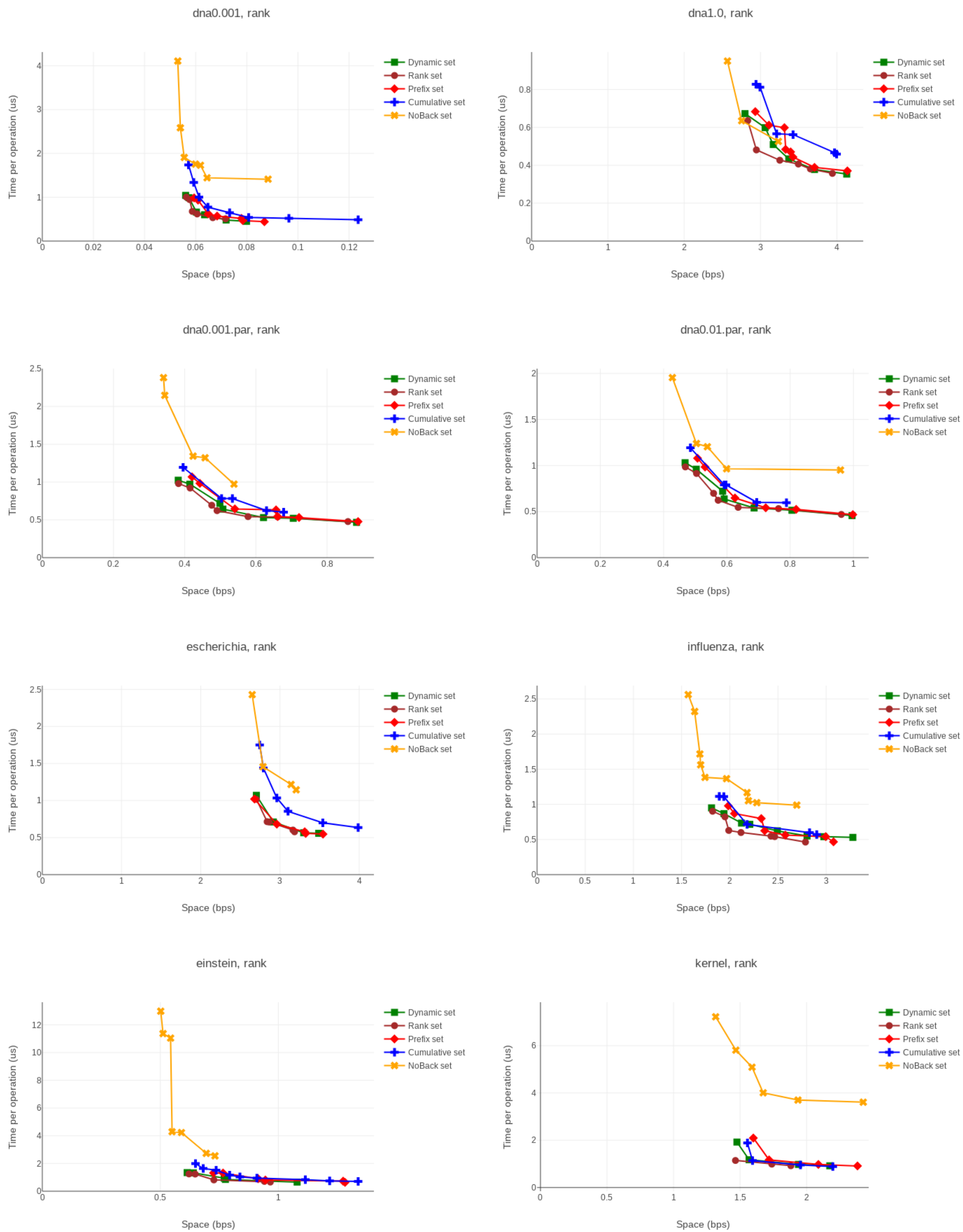


Figure 3.14: Performance of *rank* in Block Trees using different sets of fields to implement the operation. The y-axis is time in microseconds in log-scale.

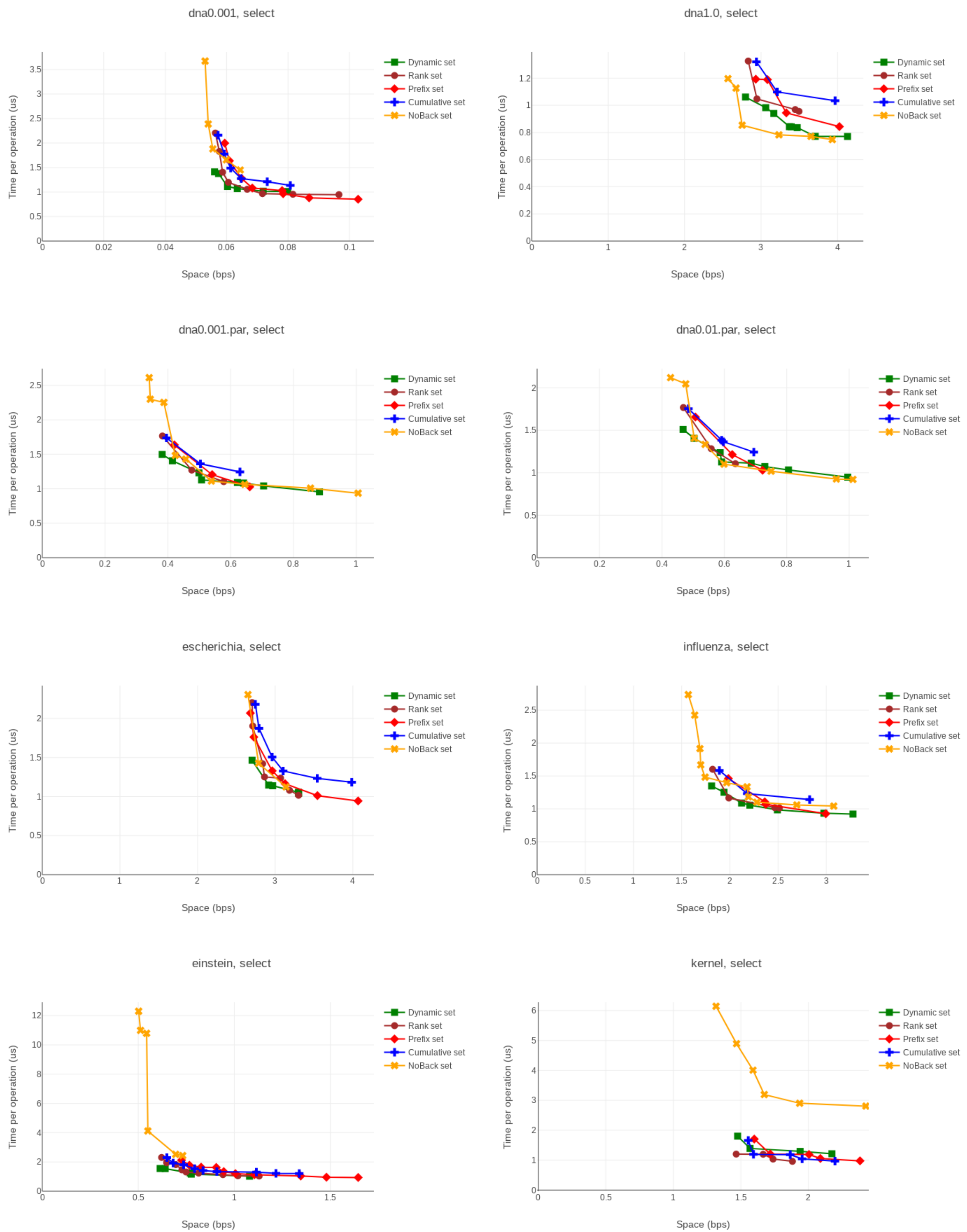


Figure 3.15: Performance of *select* in Block Trees using different sets of fields to implement the operation. The y-axis is time in microseconds in log-scale.

In Figure 3.12, we compare *conservative-heuristic-bt*, *other-conservative-heuristic-bt* and *greedy-conservative-heuristic-bt* (see Section 3.3.7) to show the effect of changing the criteria for forming back pointers from *conservative-heuristic-bt*. Note that differences between *conservative-heuristic-bt* and *greedy-conservative-heuristic-bt* are imperceptible. On the other hand, *other-conservative-heuristic-bt* incurs in a space penalty.

In Figure 3.13, we study the option of allowing pointers to next blocks through *back-front-bt* and *other-back-front-bt* (see Section 3.3.7). These graphs show that differences between *other-back-front-bt* and *bt* are imperceptible. On the other hand, *back-front-bt* incurs in a space penalty. The behavior on kernel is different from the rest; we are not aware of any reason for this.

In Figures 3.14 and 3.15, we study the different options for storing the rank/select information described in Section 3.3.6. From these graphs we note that, for both *rank* and *select*, the theoretical time penalty incurred by the Rank set of fields is not significant, since most of the time this set is Pareto Optimal to the others, except NoBack set. For this reason, we set the Rank set as the default set of fields for Block Trees. The NoBack set achieves a small reduction in space at the cost of an order of magnitude time penalty.

In conclusion, we discovered the following: using the padding technique shows to be Pareto-optimal against not using it; changing plain components to statistical compressed ones is not worth it, as it incurs in an order of magnitude time penalty in exchange for little space improvement; although theoretically inferior, the Rank set of fields, due to its simplicity, is the better approach to implement rank and select; in practice, predicted worst case scenarios are unlikely to happen, as its corresponding variants does not present a significant time penalty, which let us build Block Tree instances using even less space; there is no considerable space improvement on concatenating the levels of the Block Tree on our variants, nor using *front* pointers. Finally, there is a diverse number of variants with similar time/space behavior to the canonical Block Tree *bt*, we offer all of them in the code publicly available at `https://github.com/elarielcl/BlockTrees`. For researchers and practitioners we recommend to use two options: *bt* for theoretical guarantees, and *heuristic-bt* for better compression. It is important to mention that the space difference between these two only happens when the input is not very repetitive, if the input is very repetitive we advice to use *bt*.

Chapter 4

Compression of Repetitive Differential Arrays

In this chapter we present adaptations of previous data structures to represent differential encodings (Section 1.2 on page 5) of arrays. These are repetition-aware data structures, that is, data structures bounded by the size of some dictionary compressor; therefore, our solutions are useful when the differential encoding of the array is repetitive. Our target input will be the suffix array A, the inverse suffix array A^{-1}, and the *longest-common-prefix* array LCP of repetitive sequences, which inherit the repetitiveness of its input on their differential encodings [23].

Our first adaptation is an improvement of a previous attempt to use grammar compression to encode A differentially, the LCSA. 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. Our second and third solutions are adaptations of RSA-structures, GCC and Block Tree, where we show the similarity between answering a *rank* query and answering *access* to the original array in a differential encoding of it.

4.1 LCSA Adaptation

Recall from Section 2.2 on page 15 that LCSA [27] is a grammar-compressed representation of the suffix array A, where Re-Pair [42] is used to compress the differential encoding of the array and a sampling of the array, with sampling rate r, $A'[1, \lceil n/r \rceil]$, is added on top. In general terms, 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, this because $A[i] = A[s] + \sum_{j=s+1}^{i} A[j] - A[j-1]$. They also add a bitvector $L[1, n]$ that indicates the positions in T where each symbol of the initial rule $S \rightarrow C[1]...C[c]$ 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 $(A[s+1]-A[s])...(A[i]-A[i-1])$ from the grammar and add it to $A[s]$. This process is on-the-fly, that is, we are adding differential values to the result at the time we are decompressing them.

   a. Identify the symbols $C[x]$ and $C[y]$ containing $(A[s+1]-A[s])$ and $(A[i]-A[i-1])$, respectively. For this, *rank* on L is used.

   b. Expand $C[x]...C[y]$ 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 $A \rightarrow BC$ they first expand B and then, they expand C, so we do not expand C in case $(A[i]-A[i-1])$ was previously expanded by B. We note we could apply the same optimization on $C[x]$ if we do a right-to-left expansion. Our first adaptation to LCSA add this optimization; we call it LCSA.

Despite these optimizations, 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 adaptation, 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 this adaptation LCSA-*lengths*.

Finally, our third adaptation consists in replacing the uniform sampling of A for a sampling of A in the 1s of the bitvector L, that is, a sampling in the start position of the symbols of the initial rule $S \rightarrow C[1]...C[c]$, $A'[1,c]$. Note that in this case, the lengths of the expansions do not help, since the sampled positions are always at the beginning of the symbols of the initial rule; we call LCSA-*c-sampling* to this adaptation.

Note that, although these solutions were built to represent suffix arrays, they can represent any array A, because it is not assumed that A is a suffix array in the design of the solution.

4.2 GCC Adaptation

Recall from Section 2.1 on page 14 that GCC [60] is an RSA-structure that grammar-compresses its input with Re-Pair [42] and augments the nonterminals B with information of the length of their expansion $l(B)$ and the number of as in their expansion, $r_a(B)$ for every $a \in \Sigma$. They also store the same bitvector L that LCSA uses and a sampling rnk_a of the number of as in the prefix of T before the sampled position. This sampling is stored uniformly in T (GCCN) or in the starting positions of the symbols of the initial rule $S \rightarrow C[1]...C[c]$ (GCCC).

To answer $rank_a(i)$, GCCC works as follows:

– Find the rule containing the text position i, $C[p]$. This is done by answering *rank* on L.
– Initialize a variable as, which is a counter of the number of as, with the sampled rank

for the rule $C[p]$, $rnk_a[p]$.

– Start to expand $C[p]$ recursively:

  – If we are expanding a rule $B \rightarrow b$: If $b = a$ we increment as by one. Return as.

  – If we are expanding a rule $A \rightarrow BC$: We use $l(B)$ and the information of previous expansions to decide whether $T[i]$ is in the expansion of B or in the expansion of C. In the first case we continue expanding B recursively, and in the second case we increment as by $r_a(B)$ and continue expanding C recursively.

Note the similarity between this procedure and the one used for LCSA-lengths: Changing the sampling of rnk_as by prefix sums of the differential encoding we get exactly a sampling of the original array: $A[i] = A[1] + \sum_{j=2}^{i} A[j] - A[j-1]$. The only difference in the analogy is that in LCSA-lengths the $r_a(B)$ fields attached to the rules do not exist. If we add the field *partial-sum* to the nonterminals, which stores the sum of the symbols in the expansion of the rule, the analogy is complete.

Our adaptation of GCC to represent arrays consists on creating the GCC representation of the differential encoding of the array and change:

– The samplings of rnk_a by a sampling of the original array A.

– The fields $r_a(B)$ by a field *partial-sum*(B), which stores the sum of the differentials expanded by B.

The method to access a position of the differential array is the *access* method for GCC and the method to access the original array A is the same method used by GCC to answer *rank*, replacing the *rank* information by the new field. That is, for GCCC:

– Find the rule containing the text position i, $C[p]$. This is done by answering *rank* on L.

– Initialize a variable r, which will contain the partial sum from the sampling to i, as the sampled value of A for the rule $C[p]$.

– Start to expand $C[p]$ recursively:

  – If we are expanding a rule $B \rightarrow b$: We increase r by b and return r.

  – If we are expanding a rule $A \rightarrow BC$: We use $l(B)$ and the information of previous expansions to decide whether $A[i] - A[i-1]$ is in the expansion of B or in the expansion of C. In the first case we continue expanding B recursively, and in the second case we increment r by *partial-sum*(B) and continue expanding C recursively.

We take both GCCN and GCCC, and create the corresponding differential encodings named in the same way. We showed the adaptation in the case of GCCC, the adaptation for GCCN is analogous but considering a regular sampling of values in the input sequence, instead of the sampling in the initial rule of GCCC.

4.3 Block Tree Adaptation

Following the ideas of our GCC adaptation from the previous section, we use Block Trees to compress the differential encoding of the array A and change the fields of nodes used to answer *rank* by fields storing the sum of the 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 encoding of A, and the adaptation of *rank* using these new fields corresponds to an *access* to the original array A. This adaptation to retrieve a value of A is called *accessA* and is recursive in a node v for the ith element, as follows:

- If v is a LeafBlock, we add the symbols in $v.blk[1, i]$ and return it.
- If v is a BackBlock the query is translated into $v.ptr$ or the next block according to i and $v.off$. If $i \leq b - v.off$ then the we answer $accessA(i + v.off)$(in $v.ptr$) $-v.pfb\text{-}sum$, otherwise we return $accessA(i - b + v.off)$ (in the block next to $v.ptr$) + $v.fb\text{-}sum$.
- If v is an InternalBlock the query is translated into some of its children according to i. Suppose w is the $\left(\lfloor\frac{i-1}{r}\rfloor + 1\right)$-th child of v, then we answer $accessA((i-1) \mod r+1)$ (in w) plus the sum of the *partial-sum* fields of its left siblings.

We call this adaptation *DABT* (Differential Array Block Tree). We also propose a version where instead of storing the differences at LeafBlocks we store the prefix sums of its corresponding block; thus, we can recover this value in one access. We call this version *DABT-prefix*.

4.4 Experimental Results

4.4.1 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 (see Section 2.6 on page 20) for their internal components, GCCC and GCCN are adaptations of the implementations used in its original publication [60]. The compiler is gcc version 6.3.0, with -O9 optimization flag set.

We use the datasets *dna{0.001, 0.01, 0.1, 1.0}*, *escherichia*, *influenza*, *einstein* and *kernel* described in Section 2.5 on page 19.

4.4.2 Experiments and Structures

We run *access* experiments on the suffix array A, its inverse A^{-1} and the LCP array explained in Section 1.6 on page 11. Our experiments build the corresponding arrays for each of the inputs and average 100,000 random *access* queries in each of them.

We compare the following structures:

LCSA, LCSA-lengths & LCSA-c-sampling. Our adaptations of LCSA [27] explained in Section 4.1. The first two vary *sampling-rate* $\in \{16, 32, 64, 128, 256, 512, 1024, 2048\}$, for the sampling of the absolute values.

GCCC & GCCN. Our adaptations of GCC [60] explained in Section 4.2. We vary the same parameters as the authors suggest for GCC in their original publication.

DABT & DABT-prefix. Our adaptations of Block Tree (Chapter 3). We use the variants *bt* (theoretical guarantee) and *heuristic-bt* (low space usage). We vary the parameters $r \in \{2, 4, 8\}$ and $mll \in \{4, 8, 16, 32, 64, 128\}$.

RLCSA (with compressed-H). The RLCSA [45] explained in Section 2.2 on page 15. Same implementation of RLCSA used by Ordoñez el al. [53]; *access* to A is done with the help of a sampling on A, *access* to A^{-1} is done without sampling. We vary *sa-sampling* $\in \{32, 64, 128, 256\}$ and *block-size* $\in \{16, 32, 64\}$. This implementation also includes the compressed version of the bitvector H from Section 2.3.3 on page 17. Recall that this array stores the LCP array in text-order and an *access* to A is needed to recover the original value. The space and time shown does not consider space used for A nor its *access* time.

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

4.4.3 Results and Analysis

We show graphs of the structures answering *access* for A, A^{-1} and LCP for each of the inputs. The graphs are in 2D, where the x-coordinate represents the space in bps (bits per symbol) and the y-coordinate is the time per operation in microseconds; the input and corresponding array are indicated in the title of every graph.

Figure 4.1 shows that the LCSA structures, in general, dominate to GCC structures, both using RePair grammar compression in their internal workings. This can be explained because $LCSA$ has a simpler implementation and does not store/use an extra field of partial sums. DABT structures are the fastest, but they are the largest too, achieving as much as 5 bps for some inputs in their smallest version. RLCSA is the structure achieving the

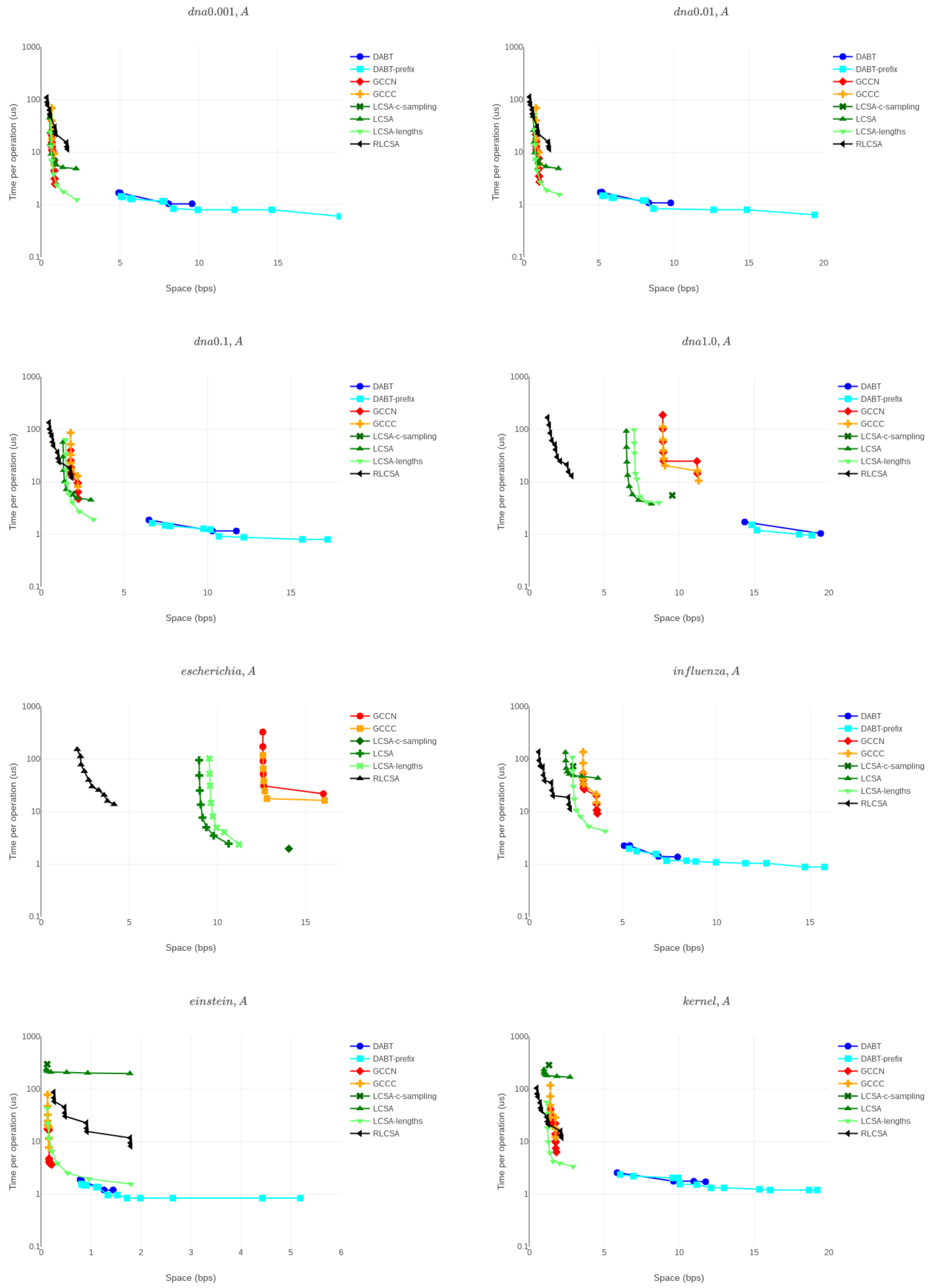


Figure 4.1: Performance of *access* in different representations of the suffix array. The y-axis is time in microseconds in log-scale.

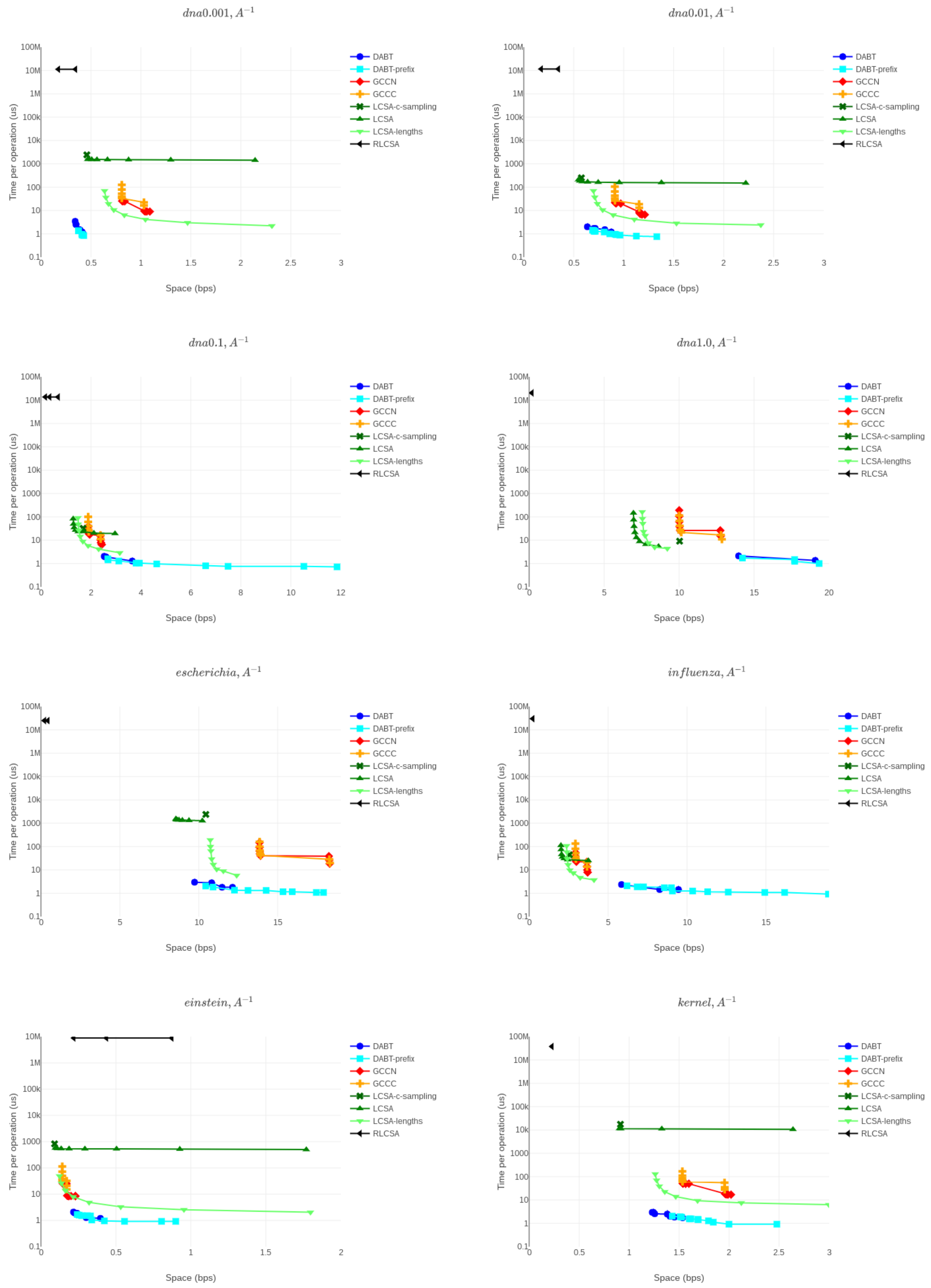


Figure 4.2: Performance of *access* in different representations of the inverse suffix array. The y-axis is time in microseconds in log-scale.

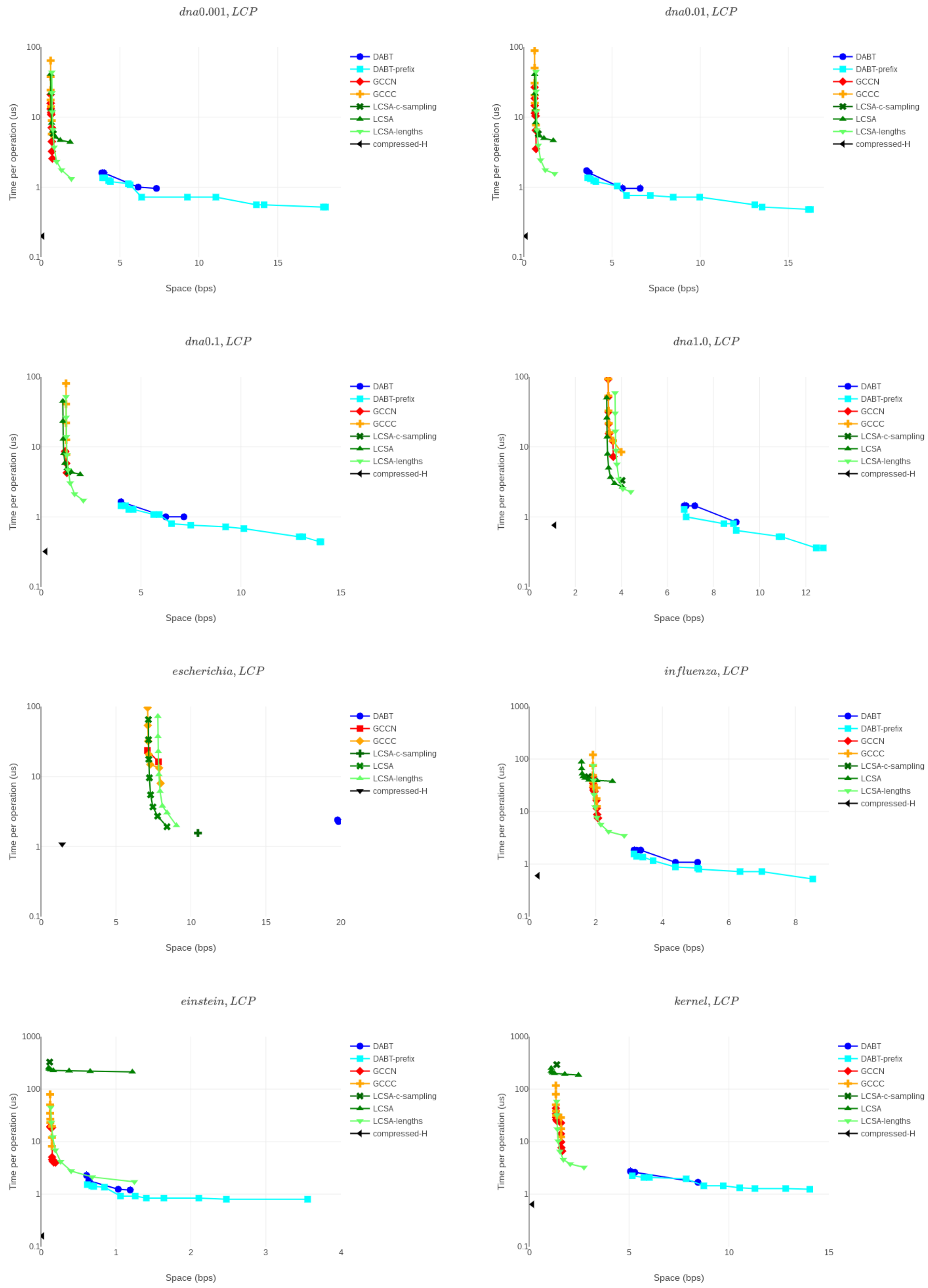


Figure 4.3: Performance of *access* in different representations of the LCP array. The y-axis is time in microseconds in log-scale.

Table 4.1: Recommended structures for practitioners depending on the accessed array and the repetitiveness of the input sequence.

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

smallest 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 added captures the bad case of LCSA. For repetitive inputs, LCSA-c-sampling is small and slow, and for non-repetitive inputs it is fast and large, what is consistent with the fact that its sampling is proportional to the repetitiveness of the input. GCCC is slower than GCCN as in the structures they are based on, but similar in space. DABT-prefix and DABT are similar in space and time, but the first presents a broader trade-off.

In the case of A^{-1}, Figure 4.2 shows that RLCSA is several orders of magnitude slower that other structures, this because this implementation does not store a sampling for A^{-1}. RLCSA keeps being the smallest structure in the majority of inputs, however, the other options present (on their smallest versions) similar space or even smaller for the most repetitive inputs.

Figure 4.3 shows that compressed-H dominates all the other structures representing LCP, which makes it the best option when LCP must be used together with A in the case of repetitive inputs.

In Table 4.1, we summarize recommended structures depending on the accessed array and the repetitiveness of the input sequence. In the case of the suffix array A, we recommend RLCSA for repetitive inputs, because it is better at capturing the repetitiveness of these inputs, and in the case of highly repetitive inputs we recommend LCSA-lengths. For the inverse suffix array A^{-1}, we recommend LCSA-lengths for repetitive inputs, because the smallest structure, the RLCSA, it is implemented without sampling and times obtained are out of competition. For highly repetitive inputs, DABT turns out to be space-competitive and is the structure of choice in this case for its superior time performance. Finally, for LCP H is the alternative of choice for its dominance in the space$\times$time map.

Chapter 5

Block Tree Compressed Topology

In this chapter we show how to use Block Trees to compress the balanced parentheses (BP) representation of the topology of an ordinal tree. This structure is aimed to handle tree topologies that present repeated patterns on their structures, because the BP representation inherits this repetitiveness. For example, repeated subtrees in the topology map to repeated substrings in the BP representation. This repetitiveness has been exploited before in a BP representation based on grammar compression called *Grammar Compressed Tree (GCT)* [53], presented in Section 2.4.2 on page 19.

Recall from Section 1.4 on page 7 that a comprehensive list of topology operations can be implemented by solving a set of basic operations or primitives on its BP representation. We implement these primitives on a Block Tree whose input is a BP representation. To make them more efficient we follow the same basic ideas used to build the rmM-tree [55], which were later applied to build the GCT, that is, we augment the nodes of the compressed representation with information that allows us to solve the primitives more efficiently.

Our representation is called *BT-CT* (Block Tree Compressed Topology). Although we are unable to prove useful upper bounds on the operation times, the BT-CT performs very well in practice, as we show in Section 5.4.

We first present the Block Tree augmentation, then the implementation of the operations with some further optimizations on them. Finally, Section 5.4 tests tree topology operations, which were implemented on top of the BT-CT using the methodology explained in Section 1.4 on page 7.

5.1 Block Tree Augmentation

Below we present the added fields to implement the primitives on BT-CT. We start from a simple Block Tree representing a BP sequence P, without the fields needed to support *rank* and *select*; note that in this case $\sigma = 2$, so we only need to store rank information for one of the symbols, because the other can be inferred. We note that many fields can be computed

from other fields. For this reason, we separate the description into *stored fields*, which are actually stored in the compact representation of BT-CT (recall Section 3.2.3 on page 27) and *fields computed on the fly*, which can be inferred from the stored fields.

5.1.1 Stored Fields

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

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

5.1.2 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 $v.blk = P[i, i+b-1]$
  - $rank_0$, the number of 0s in $v.blk$, that is, $b - v.rank_1$.
  - *excess*, the excess of 1s over 0s in $v.blk$, that is, $v.rank_1 - v.rank_0 = 2 \cdot v.rank_1 - b$.

- 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 $v.off = o$:
  - *fb-rank*$_0$, the number of 0s in $O \cap v.blk$, that is, $(b-o) - v.$*fb-rank*$_1$.
  - *pfb-rank*$_{0|1}$, the number of 0s|1s in the prefix of $u.blk$ that precedes O $(u.blk - O)$, that is, $u.$*rank*$_{0|1} - v.$*fb-rank*$_{0|1}$.
  - *fb-excess*, the excess in $O \cap u.blk$, that is, $v.$*fb-rank*$_1 - v.$*fb-rank*$_0$.
  - *sb-excess*, the excess in $O - u.blk$ (second block spanned by O), that is, $v.excess - v.$*fb-excess*.
  - *pfb-lrank*, the number of 10s that finish in $u.blk - O$, that is, $u.lrank - v.$*fb-lrank*.
  - *sb-mexcess*, the minimum excess in $O - u.blk$, that is, *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 $v.mexcess$. If $v.$*fb-mexcess* is stored ($v.$*m-fb* is 0), then $v.$*sb-mexcess* $= v.mexcess - v.$*fb-excess*. If $v.$*sb-mexcess* is stored ($v.$*m-fb* is 1), then $v.$*fb-mexcess* $= v.mexcess$.
  - *sb-mcount*, the number of times $v.$*sb-mexcess* is reached in $O - u.blk$, that is, *min-count*$(j+b, j+b+o-1)$ in P. We also store only $v.$*fb-mcount* or $v.$*sb-mcount*, the one that counts other minimum than $v.mexcess$, or store $v.$*sb-mcount* if both count the same minimum. If $v.$*fb-mcount* is stored ($v.$*m-fb* is 0), then $v.$*sb-mcount* $= v.mcount$. If $v.$*sb-mcount* is stored ($v.$*m-fb* is 1), we check whether $v.mexcess = v.$*fb-excess*$+v.$*sb-mexcess*, if that is the case $v.$*fb-mcount* $= v.mcount - v.$*sb-mcount*, otherwise, $v.$*fb-mcount* $= v.mcount$.
  - *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. For example, $v.$*fb-Mexcess-suf* $= v.$*fb-excess* $- v.$*fb-mexcess*.

For the algorithms described in the next section we use both, stored fields and fields computed on the fly, indistinctly.

5.2 Description of Primitives

We obtain the basic operations *access*, *rank*, *excess* and *select* from the Block Tree on its RSA-structure described in Chapter 3. The rest of the operations are *leaf-rank*(i), *leaf-select*(j), *fwd-search*(i, d), *bwd-search*(i, d), *min-excess*(i, j), *min-count*(i, j) and *min-select*(i, j, t).

We first explain how to solve *leaf-rank* and *leaf-select*, which are extensions of *rank* and *select* operations. Then we describe *fwd-search*(i, d) and *bwd-search*(i, d) only for $d \leq 0$, which suffices to implement topology operations from Section 1.4 [51] on page 7. Finally we describe the algorithms to solve *min-excess*(i, j), *min-count*(i, j) and *min-select*(i, j, t).

5.2.1 *leaf-rank*(i) and *leaf-select*(j)

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.

Recursively, in a node v, the *leaf-rank*(i) operation is as follows:

- If the node is a LeafBlock we do a linear scan on $v.blk$ counting leaves (10s in P) and consider the $v.lbreaker$ into the sum.
- If the node is an InternalBlock, we transform the query into the corresponding child and use the sum of the *lrank* fields of its left siblings to adjust the answer.
- In the case the node is a BackBlock we use $v.off$ to relocate the query into one of the pointed blocks, $v.pfb\text{-}lrank$ and $v.fb\text{-}lrank$ to adjust the result of the new query, and $v.lbreaker$ and $v.fb\text{-}lbreaker$ to fix the error caused by shifting. This error happens when these two bits are different and could increase/decrease the result by one.

Recursively, in a node v, the *leaf-select*(j) operation is as follows:

- If the node is a LeafBlock we initialize a counter in 1 or 0, depending on $v.lbreaker$, then we do a linear scan on $v.blk$ increasing the counter every time we see a 10. If the counter reaches j we return the corresponding position; if this does not happen we return ∞.
- If the node is an InternalBlock we use the *lrank* fields of its children, to look for the corresponding k-th child containing the answer (we return ∞ if none is found), and to relocate the query. Finally, we adjust the query by adding $(k-1)b/r$ to the result.
- In case the node is a BackBlock we use $v.pfb\text{-}lrank$, $v.fb\text{-}lrank$, $v.lbreaker$ and $v.fb\text{-}lbreaker$ to relocate the query into one of the pointed blocks and $v.off$ to adjust the result.

Like $rank_c(i)$ and $select_c(i)$, our operations work $O(1)$ time per level, and then have their same time complexity, given in Chapter 3.

5.2.2 *fwd-search*(i, d) and *bwd-search*(i, d)

We show how to solve both operations for $d \leq 0$. Thus we aim to find the smallest position $j > i$ where the excess of $P[i+1..j]$ is d for *fwd-search*(i, d), and the largest $j < i$ where the excess of $P[j+1..i]$ is $-d \geq 0$ for *bwd-search*(i, d).

We describe our solutions as recursive procedures *fwd-search*(i, j) and *bwd-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 cumulative 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) or *bwd-search*$(1, i)$ and with $e = 0$. If at some point e reaches d for *fwd-search*

or $-d$ for *bwd-search*, we have found the answer to the search. The general idea is to traverse the range of the current node v left to right for *fwd-search*, and right to left for *bwd-search*, using the fields $v.mexcess$, $v.fb\text{-}mexcess$ and $v.sb\text{-}mexcess$ for *fwd-search* and $v.Mexcess\text{-}suf$, $v.fb\text{-}Mexcess\text{-}suf$ and $v.sb\text{-}Mexcess\text{-}suf$ for *bwd-search* to speed up the procedure.

fwd-search(i, j) in a node v is as follows:

- 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 e for each 1 and decreasing e for each 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 cover the children after the k-th and before the m-th completely, and these calls will take constant time in case the answer is not reached in that child). 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 $v.off$ 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 $j \geq b - v.off$). If so, we check whether we can skip $O \cap u.blk$, namely if $e + v.fb\text{-}mexcess > d$. If we can skip it, we just update e to $e + v.fb\text{-}excess$, otherwise we call *fwd-search* recursively on the intersection of $u.blk$ 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 $v.blk$ contained in $u'.blk$, $O - u.blk$ (i.e., $j = b$ and $i \leq b - v.off$). If so, we check whether we can skip $O - u.blk$, namely if $e + v.sb\text{-}mexcess > d$. If we can skip it, we just update e to $e + v.sb\text{-}excess$, otherwise we call *fwd-search* recursively on the intersection of $u'.blk$ and the translated query range. If the answer is not ∞, we adjust and return it. Otherwise, we return ∞.

***fwd-search* examples:** Figure 5.1 shows two examples on the execution of our algorithm for *fwd-search*. 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.

bwd-search(i, j) in a node v is as follows:

- If the search range spans the entire block $v.blk$ (i.e., $j - i + 1 = b$) and the answer is

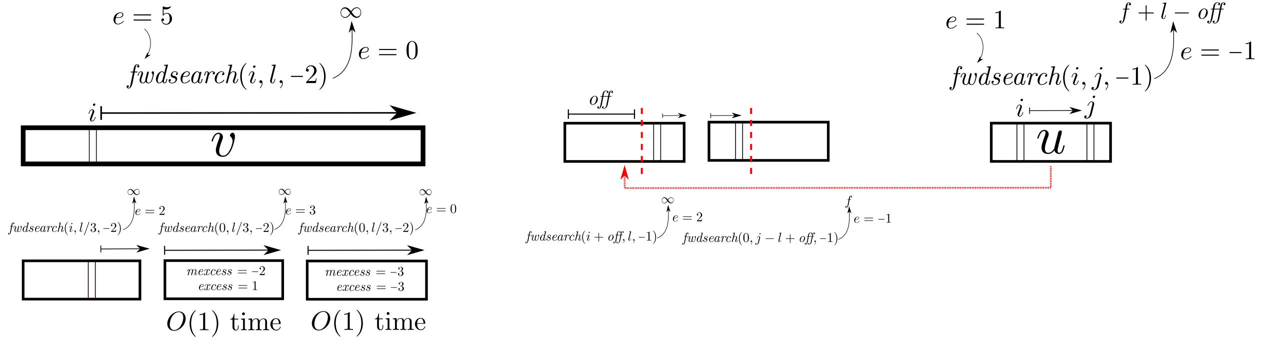


Figure 5.1: Two examples on the execution of our algorithm for *fwd-search*.

not reached inside v (i.e., $e + v.Mexcess\text{-}suf < -d$), then we increase e by $v.excess$ and return $-\infty$.

- If v is a LeafBlock we scan $v.blk$ bitwise (right to left), increasing e for each 1 and decreasing e for each 0. If e reaches $-d$ at some index k, we return $k - 1$; otherwise we return $-\infty$.
- 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 *bwd-search* recursively on the m-th to the k-th children, intersecting the query range with the extent of each child (the search range will cover the children after the m-th and before the k-th completely, and these calls will take constant time in case the answer is not reached in that child). As soon as any of these calls returns a non-$(-\infty)$ value, we adjust (i.e., shift) and return it. If all of them return $-\infty$, we also return $-\infty$.
- If v is a BackBlock we must translate the query to the original block O, which starts at offset $v.off$ in $u.blk$, where $u = v.ptr$. We first analyze the node u' next to u and check whether the query covers the suffix of $v.blk$ contained in $u'.blk$, $O - u.blk$ (i.e., $j = b$ and $i \leq b - v.off$). If so, we check whether we can skip $O - u.blk$, namely if $e + v.sb\text{-}Mexcess\text{-}suf < -d$. If we can skip it, we just update e to $e + v.sb\text{-}excess$, otherwise we call *bwd-search* recursively on the intersection of $u'.blk$ and the translated query range. If the answer is not $-\infty$, we adjust and return it. Otherwise, we turn our attention to the node u. Again, we check whether the query covers the prefix of $v.blk$ contained in $u.blk$, $O \cap u.blk$ (i.e., if $i = 1$ and $j \geq b - v.off$). If so, we check whether we can skip $O \cap u.blk$, namely if $e + v.fb\text{-}Mexcess\text{-}suf < -d$. If we can skip it, we just update e to $e + v.fb\text{-}excess$, otherwise we call *bwd-search* recursively on the intersection of $u.blk$ and the translated query range. If the answer is not $-\infty$, we adjust and return it. Otherwise, we return $-\infty$.

***bwd-search* examples:** Figure 5.2 shows two examples on the execution of our algorithm for *bwd-search*. On the left, *bwd-search* is queried on a substring of an InternalBlock v with a cumulative excess of $e = -2$ (recall our goal is to get $e = 2$). The query is translated to a prefix of the third child of v, which returns no answer and changes the cumulative excess to $e = 1$. Then the query continues, covering the second child of v completely, where it returns f (the *Mexcess-suf* field is used to realize the answer resides inside this block), which is relocated and returned. On the right, *bwd-search* is queried on a prefix of a BackBlock u

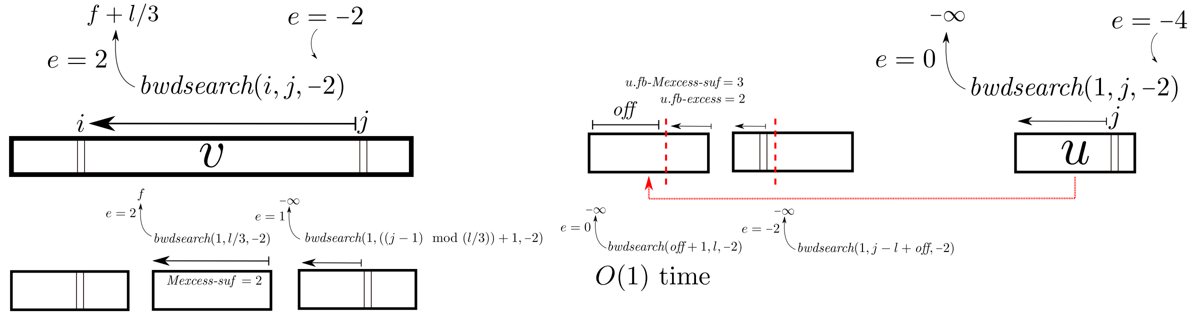


Figure 5.2: Two examples on the execution of our algorithm for *bwd-search*.

with a cumulative excess of $e = -4$. The query is translated to a prefix of the second pointed block, where it returns no answer and changes the cumulative excess to $e = -2$. Then the query continues to the suffix of the first block corresponding to the first part of $u.blk$, where it instantly returns no answer and updates the cumulative excess to $e = 0$ (using the fields *fb-Mexcess-suf* and *fb-excess*).

5.2.3 *min-excess*(i, j), *min-count*(i, j) and *min-select*(i, j, t)

These operations seek the minimum excess in $P[i..j]$, the number of times this excess is reached and the position of the t-th of these minima. 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 for *min-excess* and *min-count*. For *min-select* we will assume that m contains *min-excess*(i, j), which is the first step of this operation. In the case of *min-count* we also initialize a local variable $c = 0$, which contains the number of times the minimum is reached in that node. For *min-select* we also set the parameter t as a global variable that decreases every time a minimum is found. The idea is analogous to that of *fwd-search*: traverse the node left to right and use the *mexcess* and *mcount* fields to speed up the traversal.

min-excess(i, j) in a node v is as follows:

- If the query covers the entire block $v.blk$ (i.e., $j - i + 1 = b$), we increase e by $v.excess$ and return $v.mexcess$.
- If v is a LeafBlock we record the initial excess in $e' = e$ and scan $v.blk$ 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 cover the children after the k-th and before the m-th completely, so these will take constant time). We return the minimum between all their answers (composed with their corresponding prefix excesses).
- If v is a BackBlock we translate the query to the original block O, which starts at offset $v.off$ in $u.blk$, where $u = v.ptr$. We first check whether the query covers the prefix of

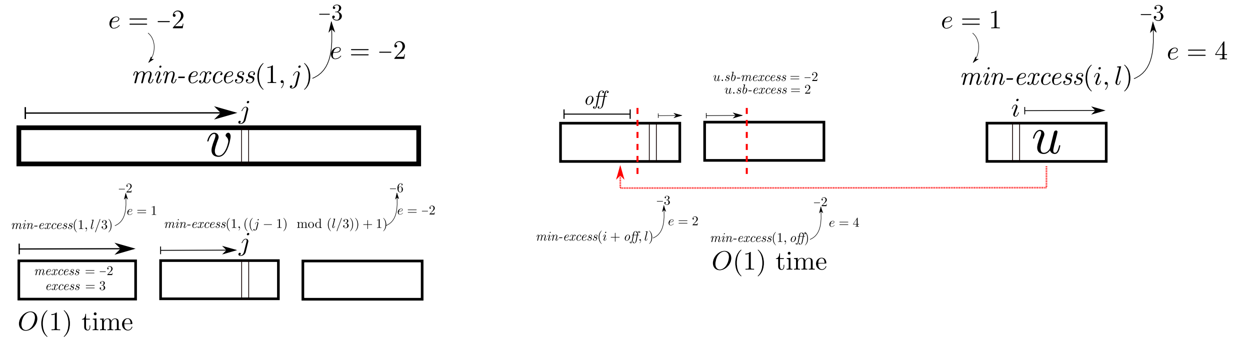


Figure 5.3: Two examples on the execution of our algorithm for *min-excess*.

$v.blk$ contained in $u.blk$, $O \cap u.blk$ (i.e., if $i = 1$ and $j \geq b - v.\mathit{off}$). 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 $u.blk$ 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 $v.blk$ contained in $u'.blk$, $O - u.blk$ (i.e., if $j = b$ and $i \leq b - v.\mathit{off}$). 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 $u'.blk$ 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.

***min-excess* examples:** Figure 5.3 shows two examples on the execution of our algorithm for *min-excess*. 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 covers completely, 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.

min-count(i, j) will return both the minimum in $v.blk[i..j]$ and the number of times this minimum is reached. Recursively in a node v, *min-count*(i, j) works as follows:

- If the query covers the entire block $v.blk$ (i.e., $j - i + 1 = b$), we increase e by $v.\mathit{excess}$ and return $v.\mathit{mexcess}$ and $v.\mathit{mcount}$.
- If v is a LeafBlock we record the initial excess in $e' = e$ and scan $v.blk$ bitwise, updating e for each bit read as in operation *fwd-search*. Every time we have $e - e' = m$, we increase c by 1, otherwise if $e - e' < m$ update $m = e - e'$ and set c to 1. At the end of the scan we return m and c.
- 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-count* recursively on the k-th to the m-th children, intersecting the query range with the extent of each child (the search range will cover the children after the k-th and before the m-th completely, so these will take constant time). We return the minimum between all their answers (composed with their corresponding prefix excesses) and the sum of the returned counts from the children where this minimum was reached.

– 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 $j \geq b - v.\mathit{off}$). If so, we simply set $m = v.\mathit{fb\text{-}mexcess}$, $c = v.\mathit{fb\text{-}mcount}$ and update e to $e + v.\mathit{fb\text{-}excess}$. Otherwise, we call *min-count* recursively on the intersection of $u.\mathit{blk}$ and the translated query range, and record its answer in m and c. 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 $i \leq b - v.\mathit{off}$). If so, we just set $m = \min(m, v.\mathit{fb\text{-}excess} + v.\mathit{sb\text{-}mexcess})$ and put on c the corresponding value, that is, $v.\mathit{sb\text{-}mcount}$ if the minimum reached there was less than m, keep c intact if the minimum was greater, and increase c by $v.\mathit{sb\text{-}mcount}$ if the minimum was the same as m, and update e to $e + v.\mathit{sb\text{-}excess}$. Otherwise, we call *min-count* on the intersection of $u'.\mathit{blk}$ and the translated query range, record its answer in m' and c', and set $m = \min(m, v.\mathit{fb\text{-}excess} + m')$ and c correspondingly as before. Finally, we return m and c.

min-select(i, j) (recall t is a global variable to the procedure) in a node v is as follows:

– If the search range spans the entire block $v.\mathit{blk}$ (i.e., $j - i + 1 = b$) and the answer is not reached inside v (i.e., $e + v.\mathit{mexcess} > m$ or $e + v.\mathit{mexcess} = m$ but $v.\mathit{mcount} < t$), then we increase e by $v.\mathit{excess}$, decrease t by $v.\mathit{mcount}$ if the minimum is reached, and return ∞.

– If v is a LeafBlock we scan $v.\mathit{blk}$ bitwise, updating e for each bit read as in operation *fwd-search*. If e reaches m at some index k, then we decrease t by 1; if $t = 0$ we return k. If t does not reach 0 in this scan, we return ∞.

– 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-select* recursively on the k-th to the m-th children, intersecting the query range with the extent of each child (the search range will cover the children after the k-th and before the m-th completely, and these calls will take constant time in case the answer is not reached in that child). 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 $u.\mathit{blk}$, 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 $j \geq b - v.\mathit{off}$). If so, we check whether we can skip $O \cap u.\mathit{blk}$, namely if $e + v.\mathit{fb\text{-}mexcess} > m$ or if $e + v.\mathit{fb\text{-}mexcess} = m$ but $v.\mathit{fb\text{-}mcount} < t$. If we can skip it, we just update e to $e + v.\mathit{fb\text{-}excess}$ and decrease t by $v.\mathit{fb\text{-}mcount}$ if the minimum is reached there, otherwise we call *min-select* 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 $e + v.\mathit{sb\text{-}mexcess} > m$ or $e + v.\mathit{sb\text{-}mexcess} = m$ but $v.\mathit{sb\text{-}mcount} < t$. If we can skip it, we just update e to $e + v.\mathit{sb\text{-}excess}$ and decrease t by $v.\mathit{sb\text{-}mcount}$ if the minimum is reached there, otherwise we call *min-select* 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 ∞.

Note that, although we look for various opportunities of using the precomputed data to skip parts of the query range, the operations *fwd-search*, *bwd-search*, *min-excess*, *min-count* and *min-select* 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.

5.3 Further Optimizations on Primitives

In this section we present versions and optimizations made for the structure presented in the previous section.

5.3.1 r-ary rmM-tree on First Level

Recall from Section 3.3.4 on page 30 that we get rid of the first levels not containing Back-Blocks. Then the first level of the tree is a sequence of blocks instead of just the root as assumed in the description of the operations from the previous section.

In the case of *leaf-rank* and *leaf-select* we use the same techniques used to solve this problem for *rank* and *select* in Chapter 3, that is, we have a sampling on top of the blocks in the first level indicating the number of leaves in the prefix of blocks previous to each block. We use this sampling to accommodate the result in case of *rank* and to find the entry block in case of *select* (with a binary search on the samples).

For the rest of the operations, *fwd-search*, *bwd-search*, *min-excess*, *min-count*, and *min-select*, the entry block is easy to find. However, it could happen that we have to traverse other blocks neighboring this entry block. In the case of the *search* operations this could happen because the answer is not inside the entry block; in the case of the *min* operations it could happen because the limits i and j fall in different blocks.

Note that for all these operations we can identify a sequence of blocks in the first level $b_x ... b_y$ in which the operation must be applied to be solved. Then a naive approach to solve the problem caused by for the elimination of the first levels is:

- Run the operation on b_x.
- Scan linearly the blocks between b_x and b_y. This scan takes constant time for each visited block, because the information required is stored as a field of the block.
- In case $x \neq y$, run the operation on b_y.

To make the linear scan faster we add an r-ary rmM-tree [55] on top of the first level, where r is the arity of the Block Tree. Recall from Section 1.5 on page 9 that the rmM-tree is an augmented binary heap built on top of a partition of a BP representation to efficiently handle the primitive operations.

In our case the partition of the BP corresponds to the blocks in the first level, and instead of building a binary heap we build an r-ary heap preserving, in this way, the arity of the Block Tree.

With this structure the linear scan from the naive approach is replaced with a traversal of the rmM-tree, which makes it exponentially faster (cost proportional to the height of the rmM-tree versus cost proportional to the leaves). This approach turned out to be orders of magnitude faster in preliminary experiments and does not use a significant amount of space compared to the whole structure.

5.3.2 BT-CT Versions

We explore getting rid of some of the fields without having a significant time penalty. For this we create two versions of the BT-CT.

BT-CT-nobackmin, which corresponds to BT-CT but without the fields *fb-mexcess*, *fb-mcount*, *sb-mexcess* and *sb-mcount* in BackBlocks. For this version the optimization attempts in BackBlocks are removed.

BT-CT-noback, which is a more extreme version of BT-CT-nobackmin where the *rank/select* fields for BackBlock are removed, that is we use the NoBack set of fields from Section 3.3.6 on page 32 for both ones and leaves.

5.4 Experimental Results

5.4.1 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 (see Section 2.6 on page 20) for their internal components, GCT corresponds to the implementation used on its original publication [53]. The compiler is gcc version 6.3.0, with -O9 optimization flag set.

We use the datasets *dna{0.001, 0.01, 0.1, 1.0}.par*, *escherichia.par*, *influenza.par*, *einstein.par* and *kernel.par* described in Section 2.5 on page 19.

5.4.2 Experiments and Structures

We compare the following structures:

SADA. The BP implementation explained in Section 1.5 on page 9. We use the implementation of the *sdsl*, `cst_sada`, on 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 [53] explained in Section 2.4.2 on page 19. We use the same implementation used on its original publication, but only considering the topology space. We vary parameters *rule-sampling* and *C-sampling* as the authors suggest.

BT-CT, BT-CT-nobackmin & BT-CT-noback. Our Block Tree based topologies explained in this chapter. We use the variants *bt* (theoretical guarantee) and *heuristic-bt* (low space usage). We vary the parameters $r \in \{2, 4, 8\}$ and $mll \in \{16, 32, 64, 128, 256, 512\}$.

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

We implemented all topology operations from Table 1.1 on page 8 for our structures (BT-CT, BT-CT-nobackmin and BT-CT-noback) but GCT and SADA only have implemented a subset of them, so 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, similar to the scheme used in previous work on Compressed Suffix Trees [1, 53] 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 $tree\text{-}depth(v) = td \geq 10$, and choose a random $d \in [1, td - 1]$. For *lca* we choose random leaf pairs.

5.4.3 Results and Analysis

We show graphs of the structures answering each of the operations, for each of the inputs. The graphs are in 2D, where the x-coordinate represents the space in bps (bits per symbol) and the y-coordinate is the time per operation in microseconds; the input BP and operation are indicated in the title of every graph.

The previous figures show that the BT-CT structures are, in general, one order of magnitude faster than the GCT, and similar to SADA. SADA uses $\sim$ 1.4 bps independently of the repetitiveness of its inputs, which shows the insensitivity of statistical techniques on capturing this kind of compressibility. GCT is the smallest structure, followed by the BT-CT structures, both with low space usage for very repetitive inputs. For example, for *einstein* GCT uses $\sim$ 0.03-0.35 bps and the BT-CT structures $\sim$ 0.11-0.25 bps. BT-CT-nobackmin presents a small reduction in space but does not produce a significant impact on its time performance. BT-CT-noback further reduces the spaces but it produces up to one order of magnitude time penalty in some cases.

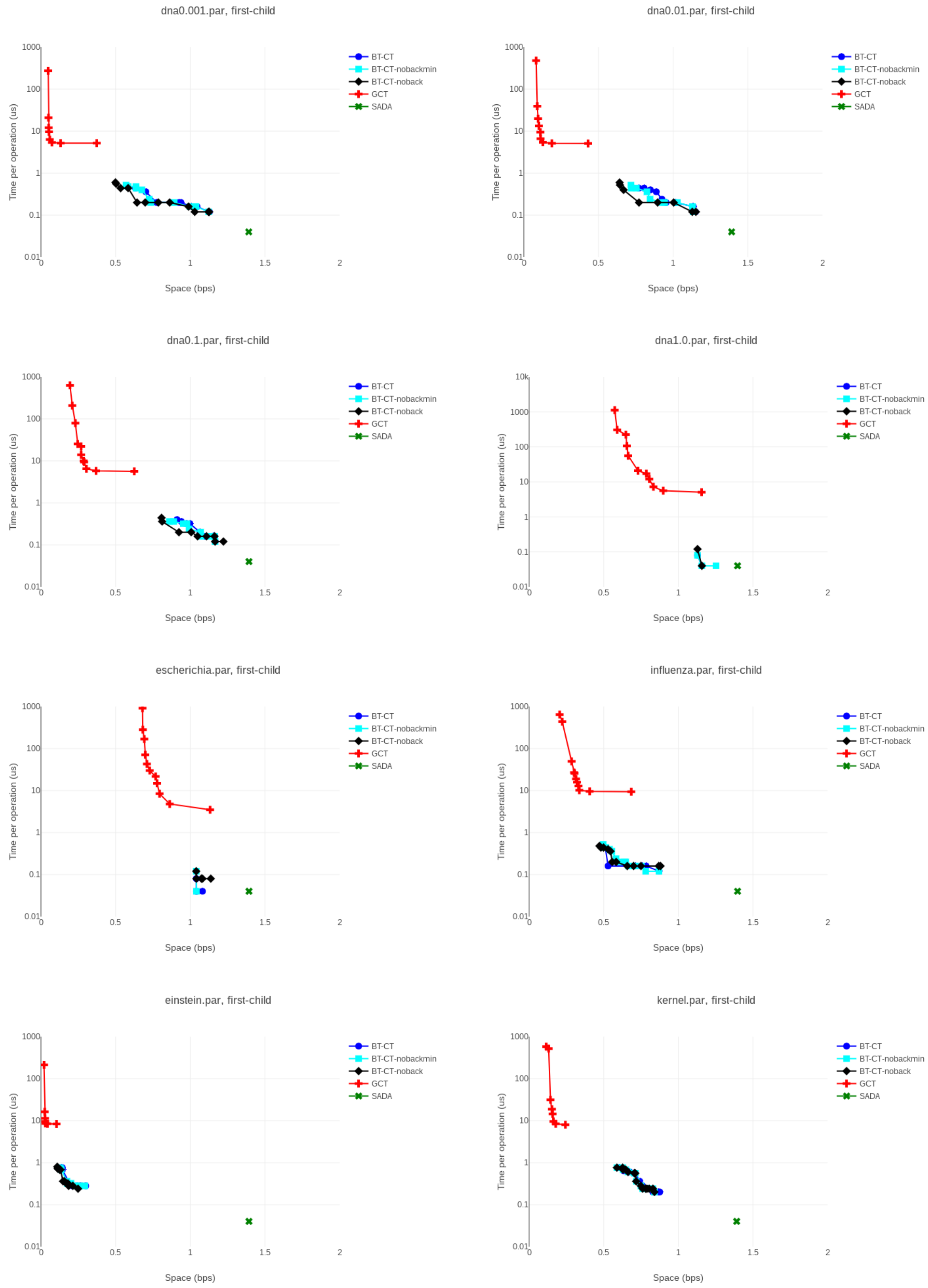


Figure 5.4: Performance of *first-child* in different BP representations. The y-axis is time in microseconds in log-scale.

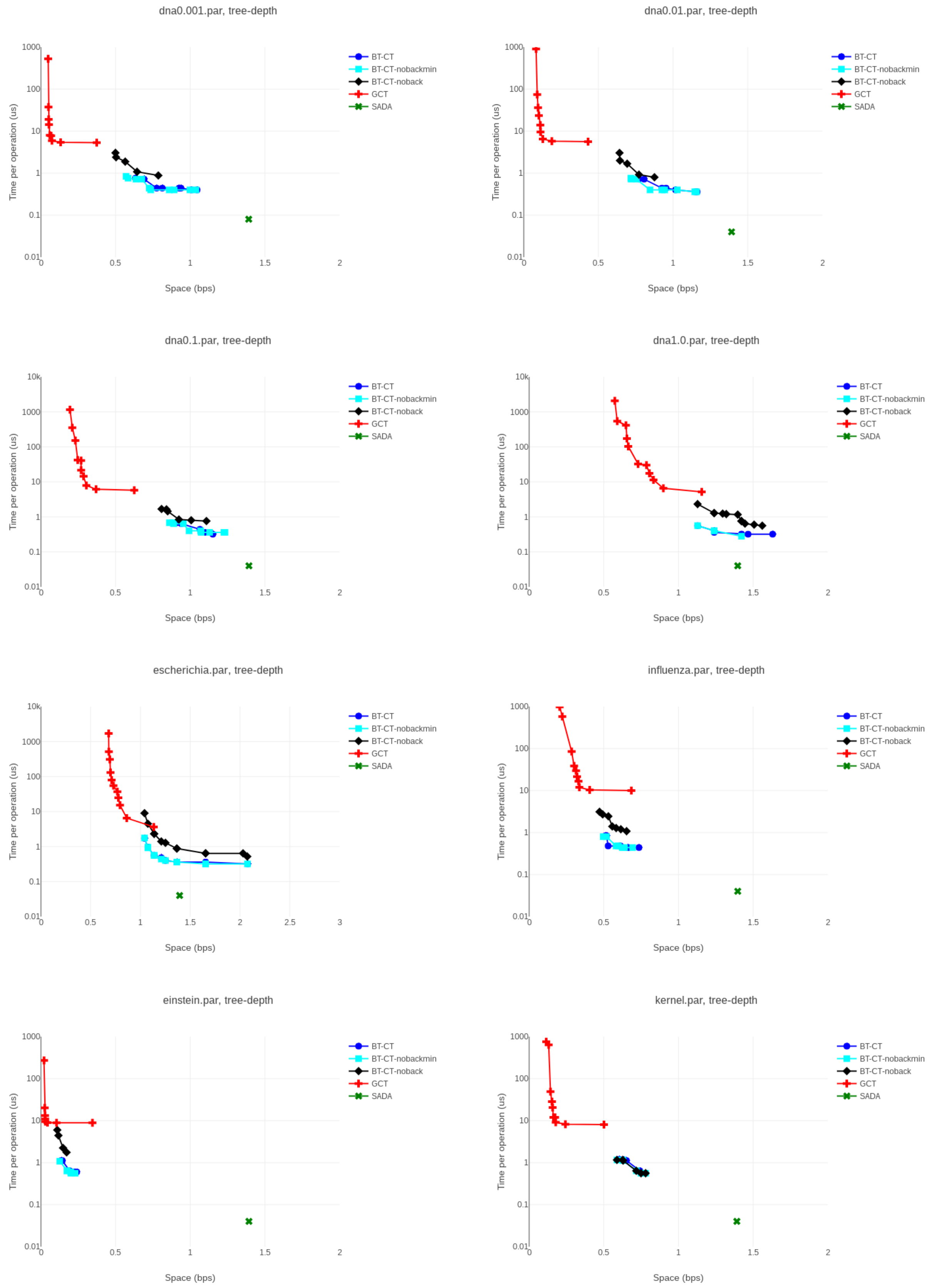


Figure 5.5: Performance of *tree-depth* in different BP representations. The y-axis is time in microseconds in log-scale.

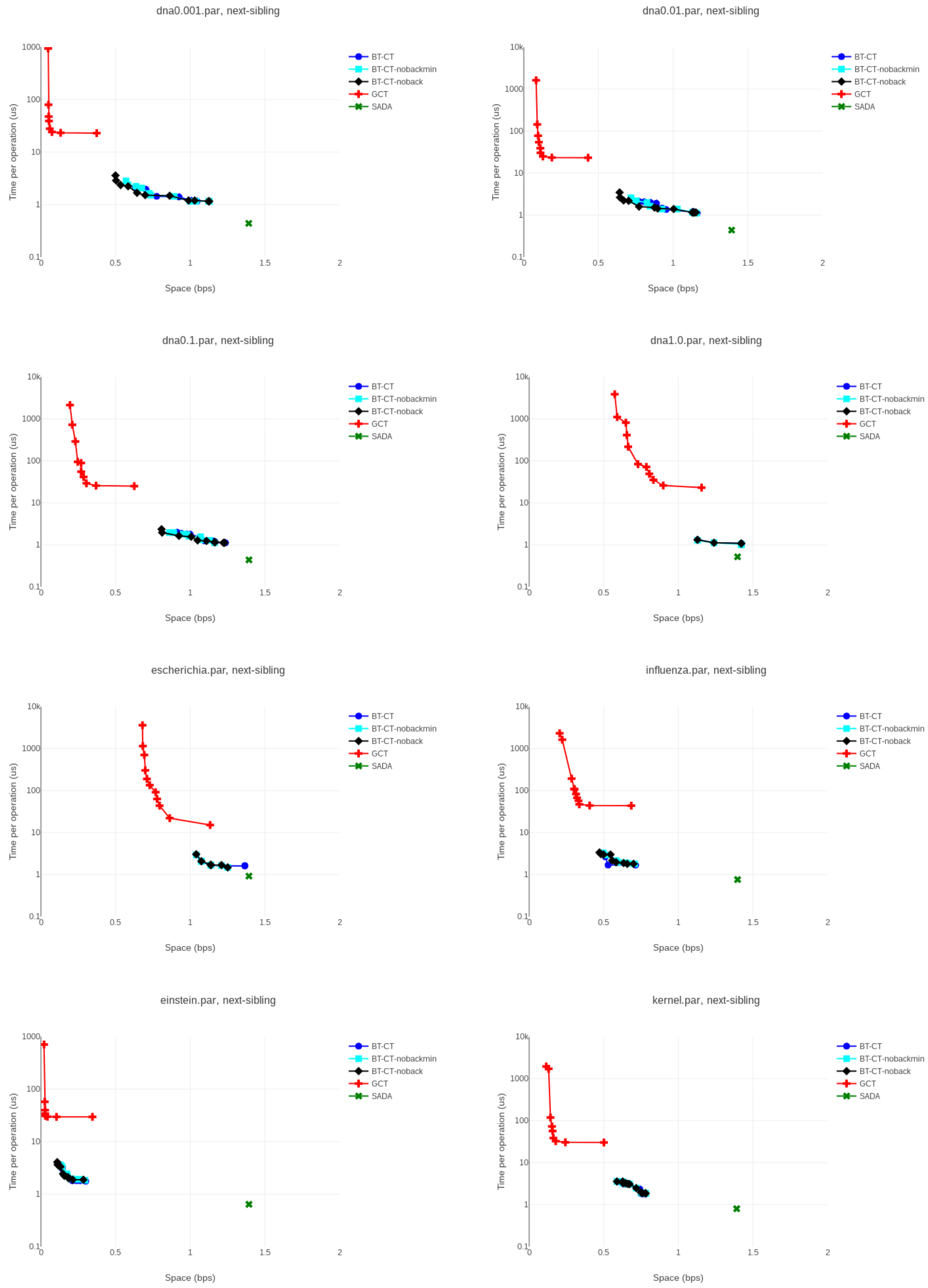


Figure 5.6: Performance of *next-sibling* in different BP representations. The y-axis is time in microseconds in log-scale.

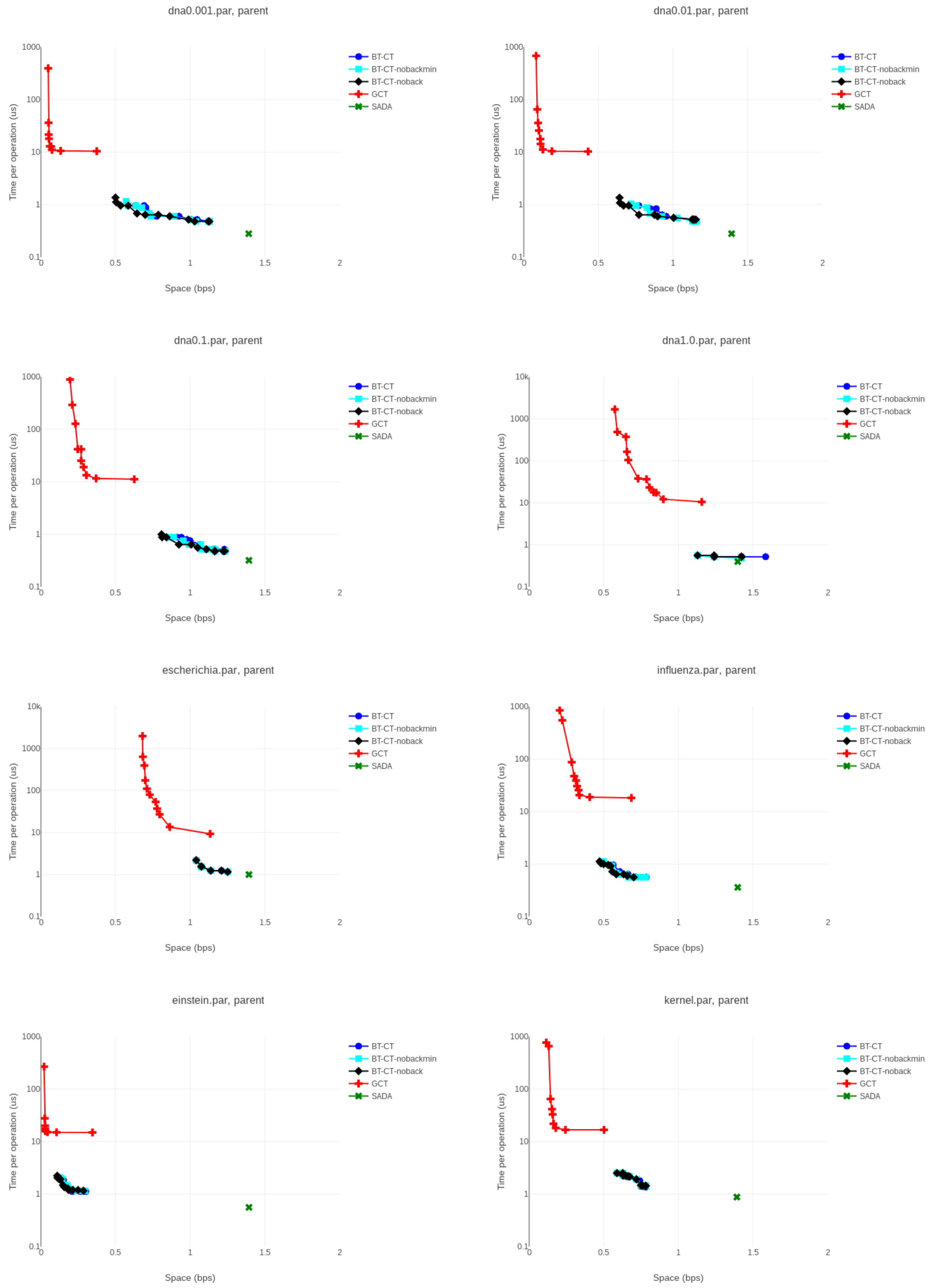


Figure 5.7: Performance of *parent* in different BP representations. The y-axis is time in microseconds in log-scale.

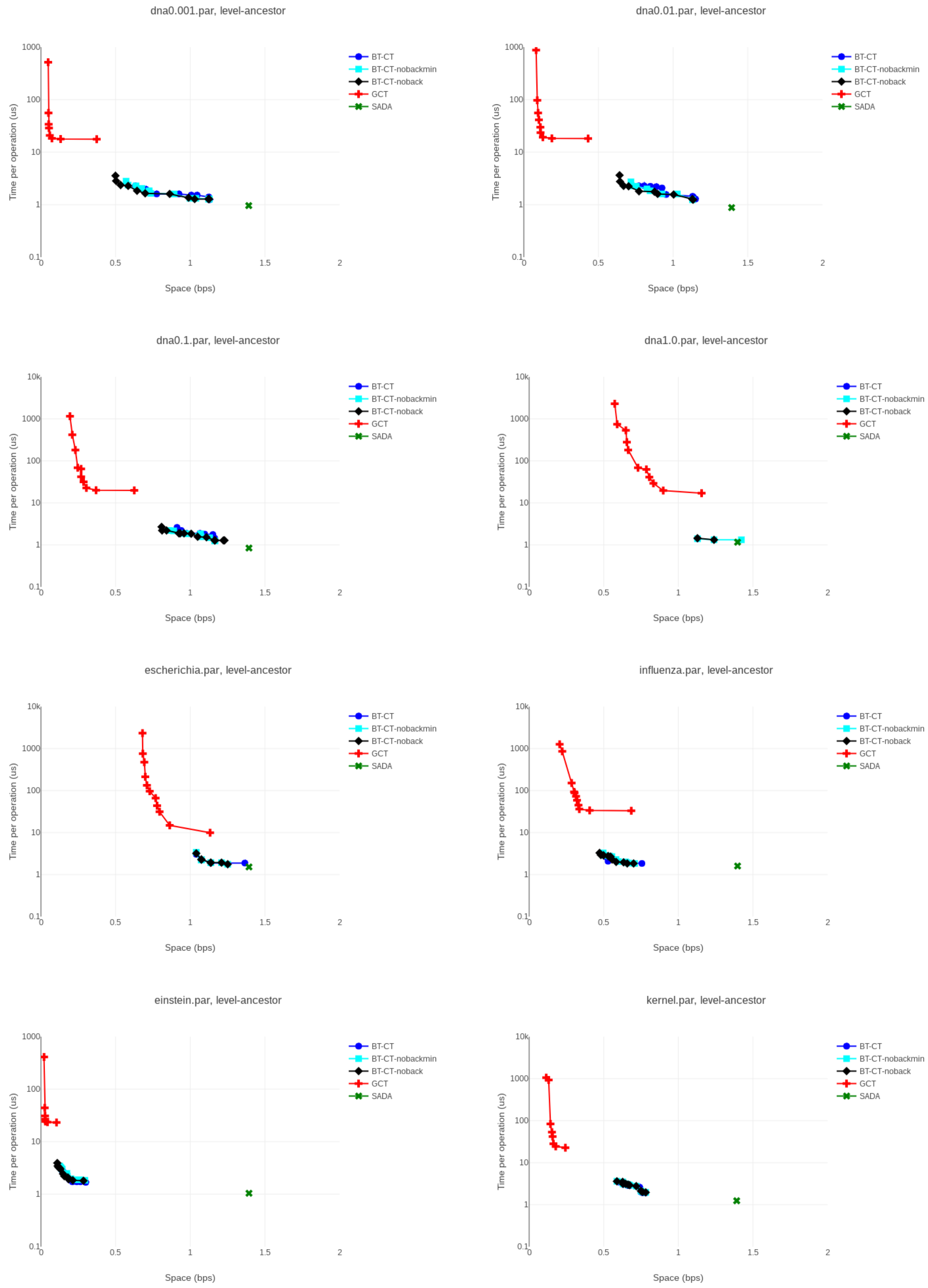


Figure 5.8: Performance of *level-ancestor* in different BP representations. The y-axis is time in microseconds in log-scale.

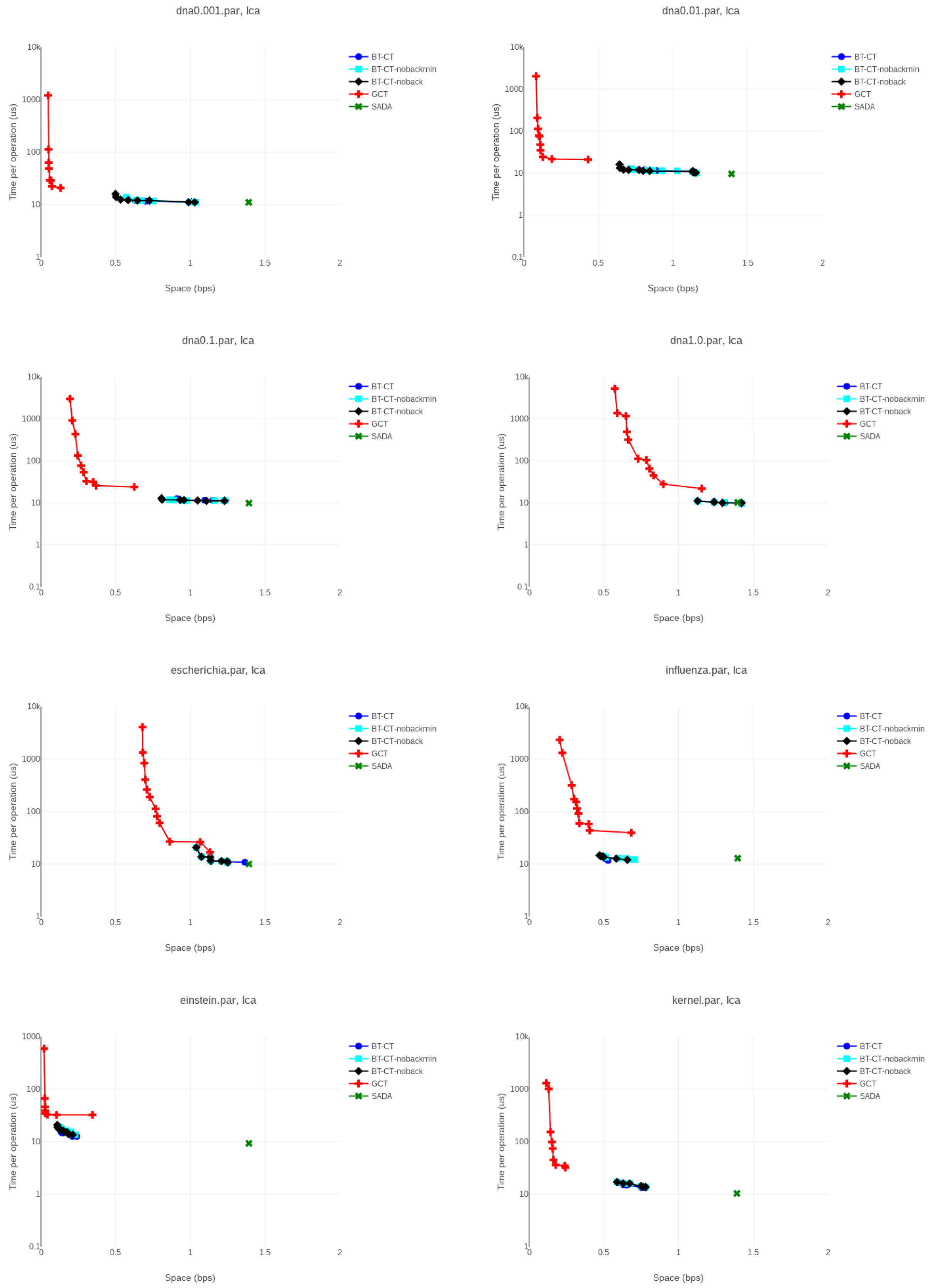


Figure 5.9: Performance of *lca* in different BP representations. The y-axis is time in microseconds in log-scale.

Figures 5.4 and 5.5 show the operations *first-child* and *tree-depth*, which uses the basic *access* and *rank* primitives. As SADA solves these primitives using techniques of plain bitvectors (see Section 1.3.1 on page 6), it is an order of magnitude faster than BT-CT and two orders over GCT.

Figures 5.6, 5.7 and 5.8 show the operations *next-sibling*, *parent* and *level-ancestor*, which are solved by using the searches: *fwd-search* and *bwd-search*. For these operations the BT-CT structures reach times very similar to SADA, both being one order of magnitude superior to GCT. That is, the BT-CT works in times of the rmM-tree but uses space aware of the repetitiveness of its inputs, even when we could not show a good theoretical bound for the searches.

Figure 5.9 shows the operation *lca*, which uses the primitives *min-excess*, *bwd-search* and *fwd-search*. In this case, the BT-CT structures reach space very similar to SADA again, but this time the GCT does not present a very different performance (less than one order of magnitude) on its fastest versions.

Our BT-CT presents a new topology representation, time competitive to the fastest known compressed representation, but using space that decreases with repetitiveness. As a consequence, we advise to use BT-CT when dealing with very repetitive inputs, but if memory resources are very limited we recommend GCT instead, losing up to 2 orders of magnitude in time performance.

Chapter 6

Our Repetition-Aware Compressed Suffix Trees

In this chapter we present our new Repetition-Aware Compressed Suffix Trees, which follow the three-component scheme from Sadakane's proposal [67] seen in Section 2.3.1 on page 16.

For the BP topology representation, we use the BT-CT described in Chapter 5, in fact we note that some of the fields and primitives used in the augmentation are not required to implement the necessary operations for suffix trees. Our results with BT-CT let us improve the operation child, which is a bottleneck among the operations of CSTs.

For the LCP, we use the run-length compressed version of the bitvector H [23], and the RLCSA [45] as our CSA. We explain how to use the differential array representations of the suffix array A and its inverse A^{-1} shown in Chapter 4 to speed up and replace some of the components of the RLCSA.

6.1 BT-CT Adaptation

We note that the *mcount* fields are only required to solve the primitives *min-count* and *min-select*, which are only used to implement the topology operations child-num, child-rank and child (see Table 1.1 on page 8). These three topology operations do not show up in the suffix tree operations given in Table 1.2 on page 12. Note that the child operation offered in topology operations recovers a child by its ordinal position, while the child operation required by suffix trees consists on taking a child by a letter, which is currently implemented with a linear scan on the children of the node, using the CSA to query the letters pointing to each of these nodes. Hence, we get rid of the fields *mcount*, *fb-mcount* and *sb-mcount* from nodes in the BT-CT.

6.1.1 Operation child

The fast operations enabled by our BT-CT structure give space for an improved algorithm to solve operation child(v, a). Previous CSTs first compute $d = \text{string-depth}(v)$ and then linearly traverse the children of v from $u = \text{first-child}(v)$ 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 $O(\sigma)$ operations next-sibling, but only $O(\log \sigma)$ operations letter.

6.2 RLCSA Speed Up

As said, we choose the RLCSA [45] as our CSA. Our first proposal of Repetition-Aware Compressed Suffix Tree uses the RLCSA with no changes.

Recall from Section 2.2 on page 15 that the RLCSA uses $O(r(2\log(n/r)+\log\sigma)(1+o(1)))$ bits of space and can support access to the suffix array A with the help of a regular sampling of $O((n/s)\log n)$ 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, 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 [45], so we opt for replacing these samplings by differential encodings of A and A^{-1} using LCSA or DABT from Chapter 4 to capture the repetitiveness of these encodings, which let us speed up access to both arrays. The new versions of CSTs are called BT-CST-{LCSA, NONE}-{LCSA, DABT, NONE}, where the last word corresponds to the implementation of A^{-1} and the previous to the implementation of A. Finally, BT-CST-NONE-NONE is renamed to BT-CST, which is our basic CST.

6.3 Experimental Results

6.3.1 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 (see Section 2.6 on page 20) for their internal components, GCST corresponds to the implementation used on its original publication [53]. The compiler is gcc version 6.3.0, with -O9 optimization flag set.

We use the datasets *dna{0.001, 0.01, 0.1, 1.0}*, *escherichia*, *influenza*, *einstein* and *kernel* described in Section 2.5 on page 19.

6.3.2 Experiments and Structures

We compare the following structures:

CST_SADA ,CST_SCT3, CST_FULLY. Adaptation and improvements from the *sdsl* library on the indexes of Sadakane [67], Fischer et al. [23] and Russo et al. [64], respectively. CST_SADA maximizes speed using Sadakane's CSA [66] 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 [62] 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 [45] and use a run-length-compressed representation of bitvector H [23].

GCST. The Grammar-based Compressed Suffix Tree [53]. We vary parameters *rule-sampling* and *C-sampling* as they suggest.

BT-CST-{LCSA, NONE}-{LCSA, DABT, NONE}. Our new Compressed Suffix Tree with the described components. For the BT-CT component we vary $r \in \{2, 4, 8\}$ and $mll \in \{4, 8, 16, 32, 64, 128, 256\}$. For the versions using LCSA we use LCSA-lengths to get better time performance and set the sweet point *sampling-rate* $= 128$. For the versions using DABT we use DABT-prefix on a low space version, that is, *heuristic-bt* $r = 2$ and $mll = 16$. For the versions using LCSA for the suffix array A we do not consider the space used by the sampling of the RLCSA although it is present on the implementation. Recall from Section 4.4.3 on page 55 that the implementation of RLCSA does not use sampling on A^{-1} and it is very slow. Then we do not apply $\Psi^d[i] = A^{-1}[A[i] + d]$ when a representation of A^{-1} is not present.

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 [53].

We implemented all the suffix tree operations of Table 1.2 on page 12. From those, we present the performance comparison with other CSTs on ten important operations: *first-child*, *tree-depth*, *next-sibling*, *parent*, *level-ancestor*, *lca*, *suffix-link*, *string-depth*, *string-ancestor* and *child*. Data points are the average of 100,000 random queries, similar to the scheme used in previous work on Compressed Suffix Trees [1, 53] to choose the nodes on which the operations are called. For *first-child*, *tree-depth*, *next-sibling*, *parent* and *string-depth* we collect the nodes in leaf-to-root paths starting from random leaves. For *level-ancestor* we

choose random leaves v whose *tree-depth*$(v) = td \geq 10$, and choose a random $d \in [1, td-1]$. For *lca* we choose random leaf pairs. 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 Substrings

To test our suffix trees in more complex scenarios we implemented the suffix-tree-based algorithm to solve the "maximal substrings" problem [53] on all of the above implementations except for CST_FULLY (because of its poor time performance).

The problem is as follows. We want to find all the maximal substrings (i.e., adding a neighbor letter breaks the property) of a pattern $S[1, m]$, which are also substring of a text $T[1, n]$. This problem can be solved in $O(m)$ time using the suffix tree of T to find the requested substrings. More precisely, the algorithm maintains two pointers, which indicate the borders of the current processed substring of S, $S[i, j]$, and work by iteratively applying the following two steps: try to increase j by descending in the suffix tree by the letter $S[j+1]$. The algorithm first descends by nodes as much as possible, outputs the corresponding maximal substring, and continues to the next step. In the next step, the algorithm increases i by taking suffix links, until the algorithm can descend by a node again, returning to the previous step. This iteration of two steps is repeated until S is completely processed.

We use the same setup of the GCST publication [53], 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 $r = 2$ and $mll = 128$ 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.

6.3.3 Results and Analysis

We show graphs of the structures answering each of the operations, for each of the inputs. The graphs are in 2D, where the x-coordinate represents the space in bps (bits per symbol) and the y-coordinate is the time per operation in microseconds; the input and operation are indicated in the title of every graph.

In the case of the "maximal substrings" algorithm, the time is shown divided by the length m of the pattern.

The figures show the space and time for all the indexes and all the operations. The smallest structure is GCST. The next smallest indexes are BT-CST, and CST_FULLY. The compressed indexes not designed for repetitive collections use 2–3 bps less if combined with

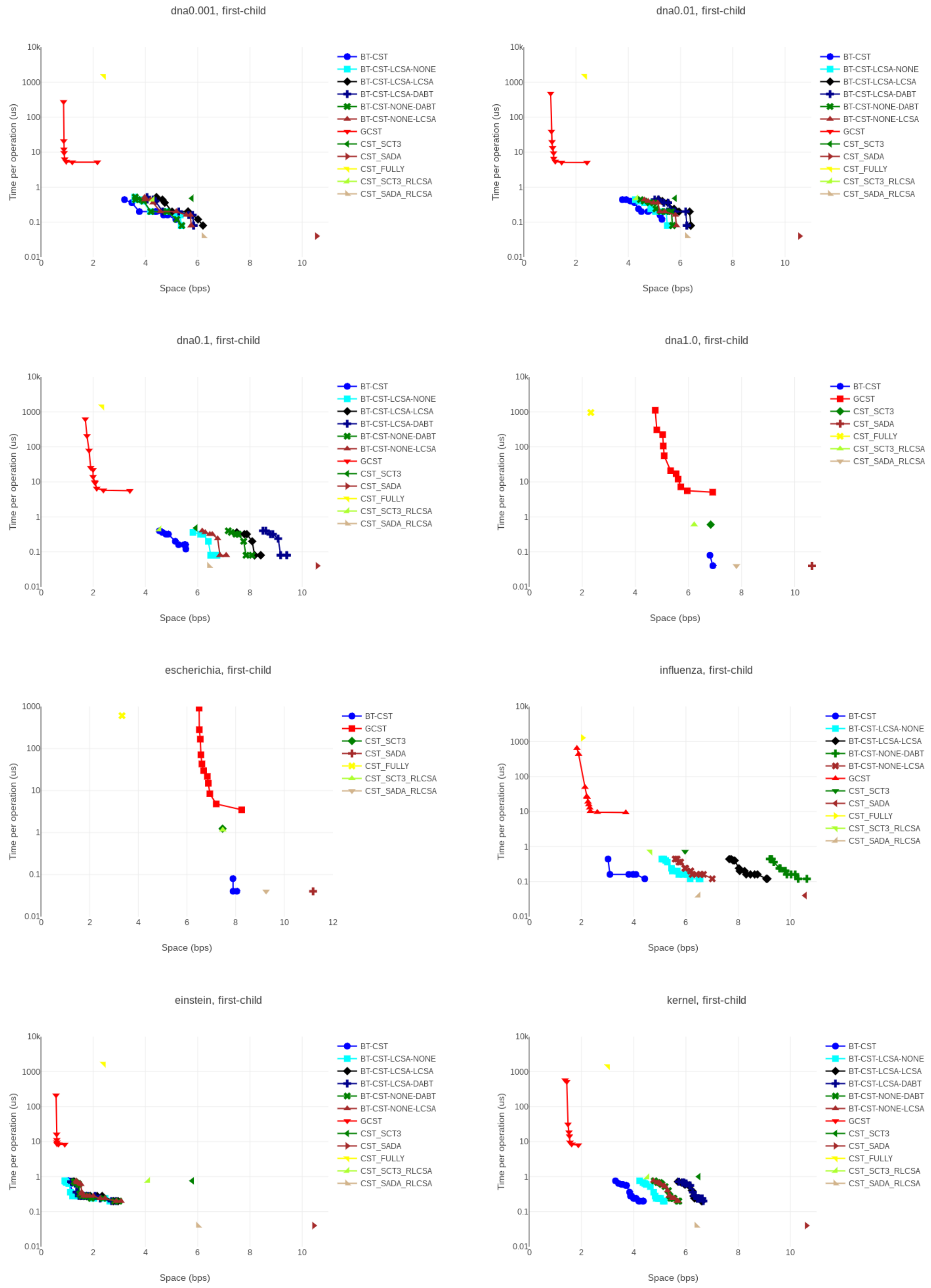


Figure 6.1: Performance of *first-child* in different CSTs. The y-axis is time in microseconds in log-scale.

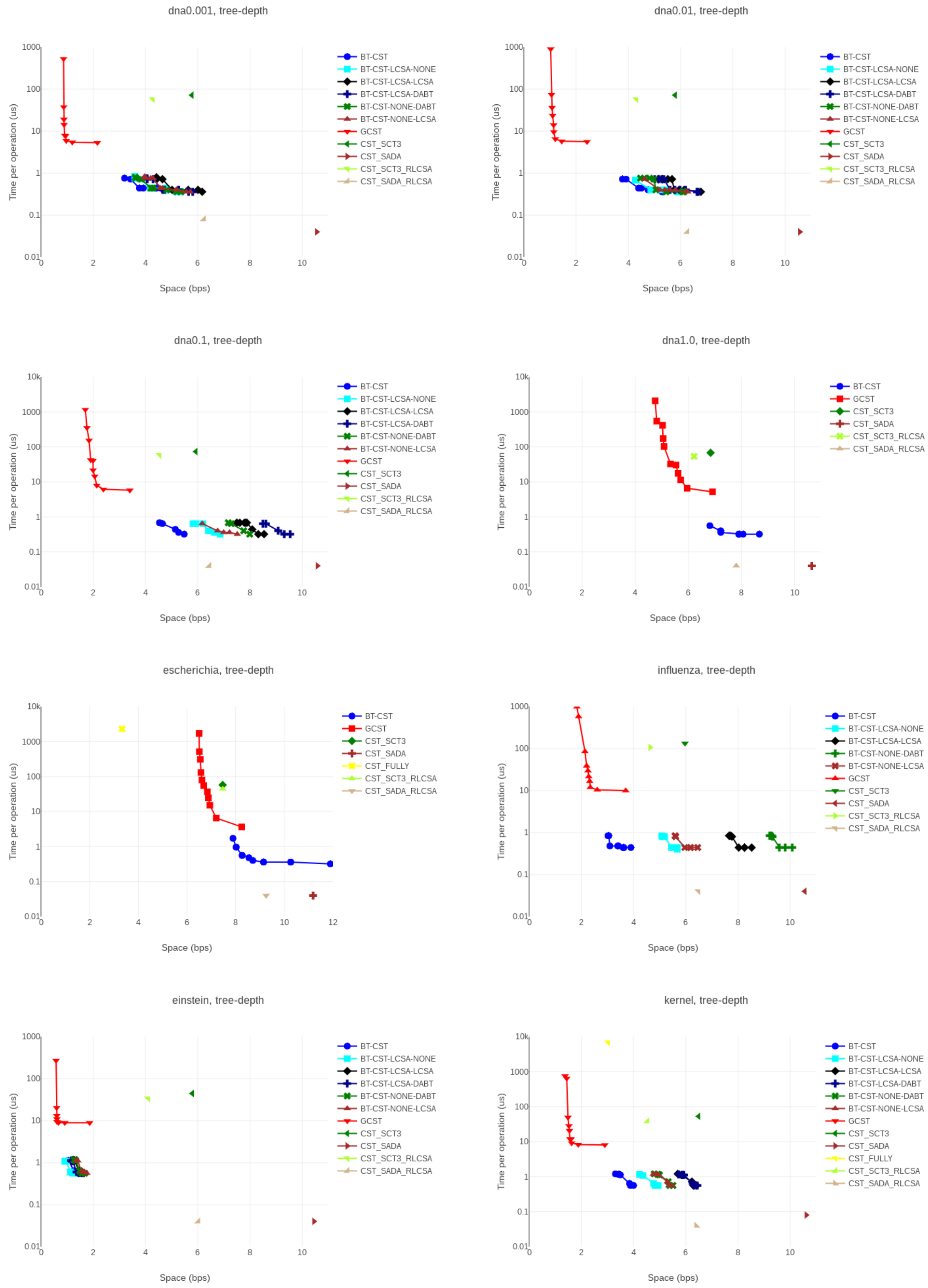


Figure 6.2: Performance of *tree-depth* in different CSTs. The y-axis is time in microseconds in log-scale.

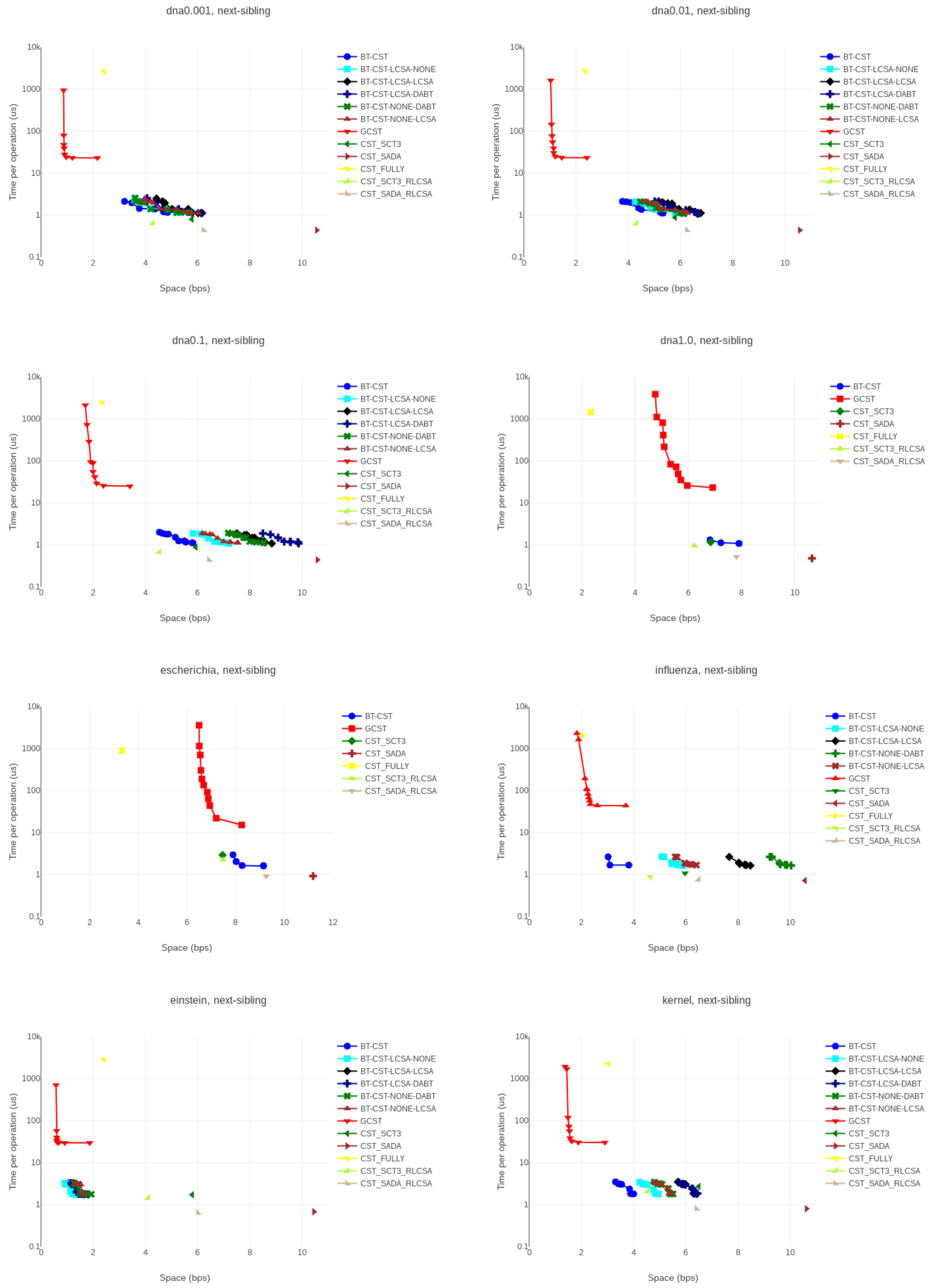


Figure 6.3: Performance of *next-sibling* in different CSTs. The y-axis is time in microseconds in log-scale.

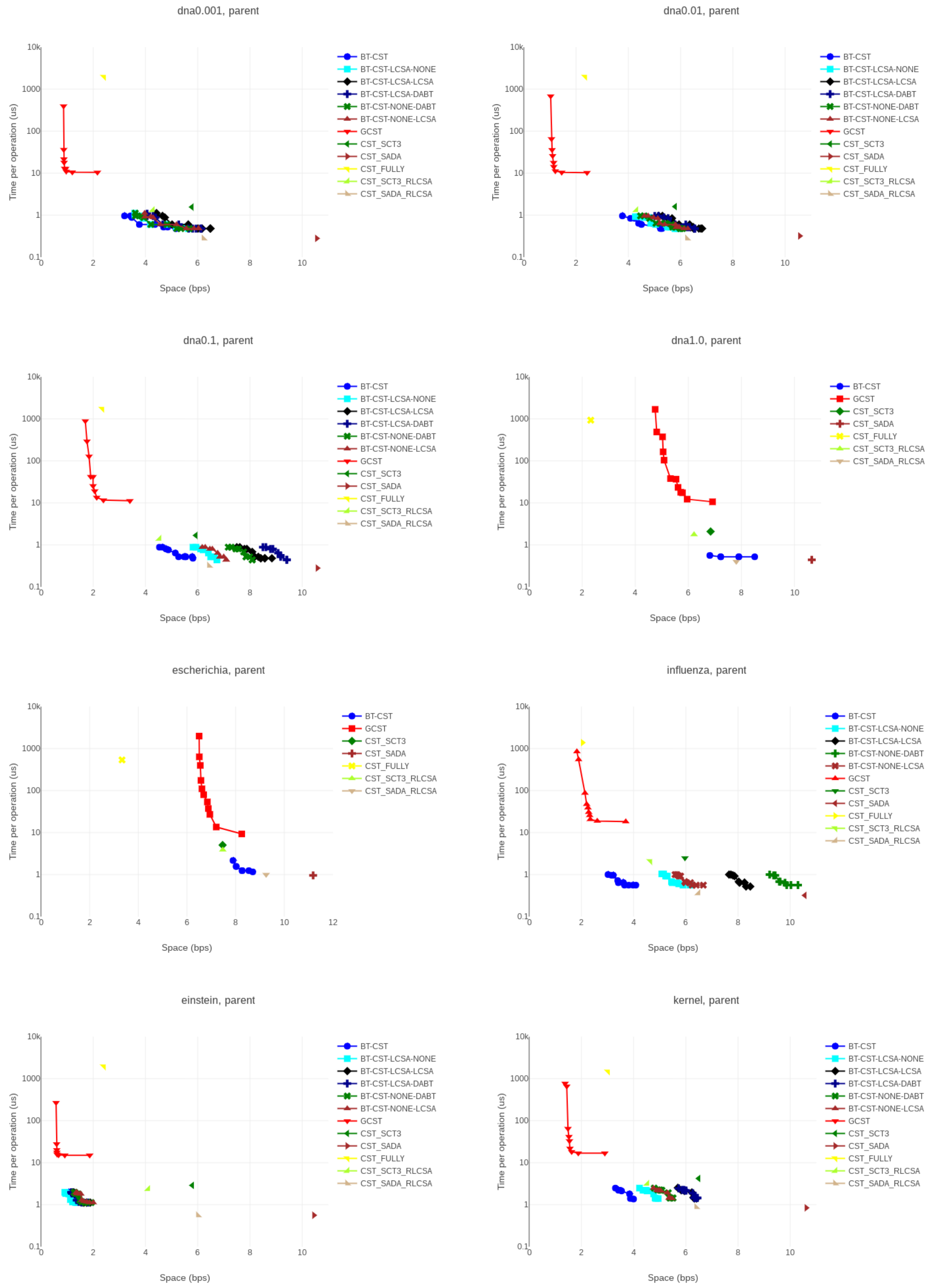


Figure 6.4: Performance of *parent* in different CSTs. The y-axis is time in microseconds in log-scale.

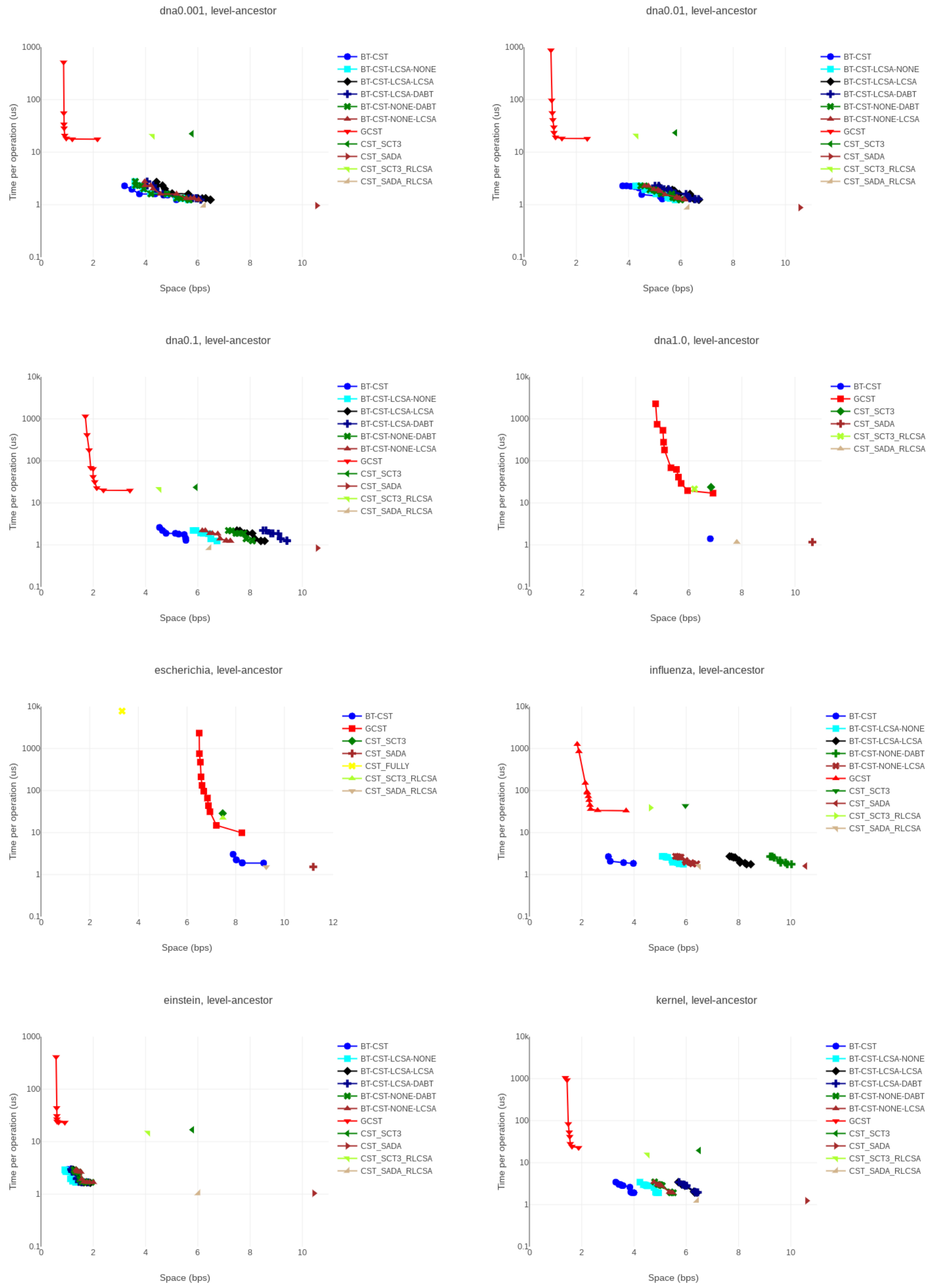


Figure 6.5: Performance of *level-ancestor* in different CSTs. The y-axis is time in microseconds in log-scale.

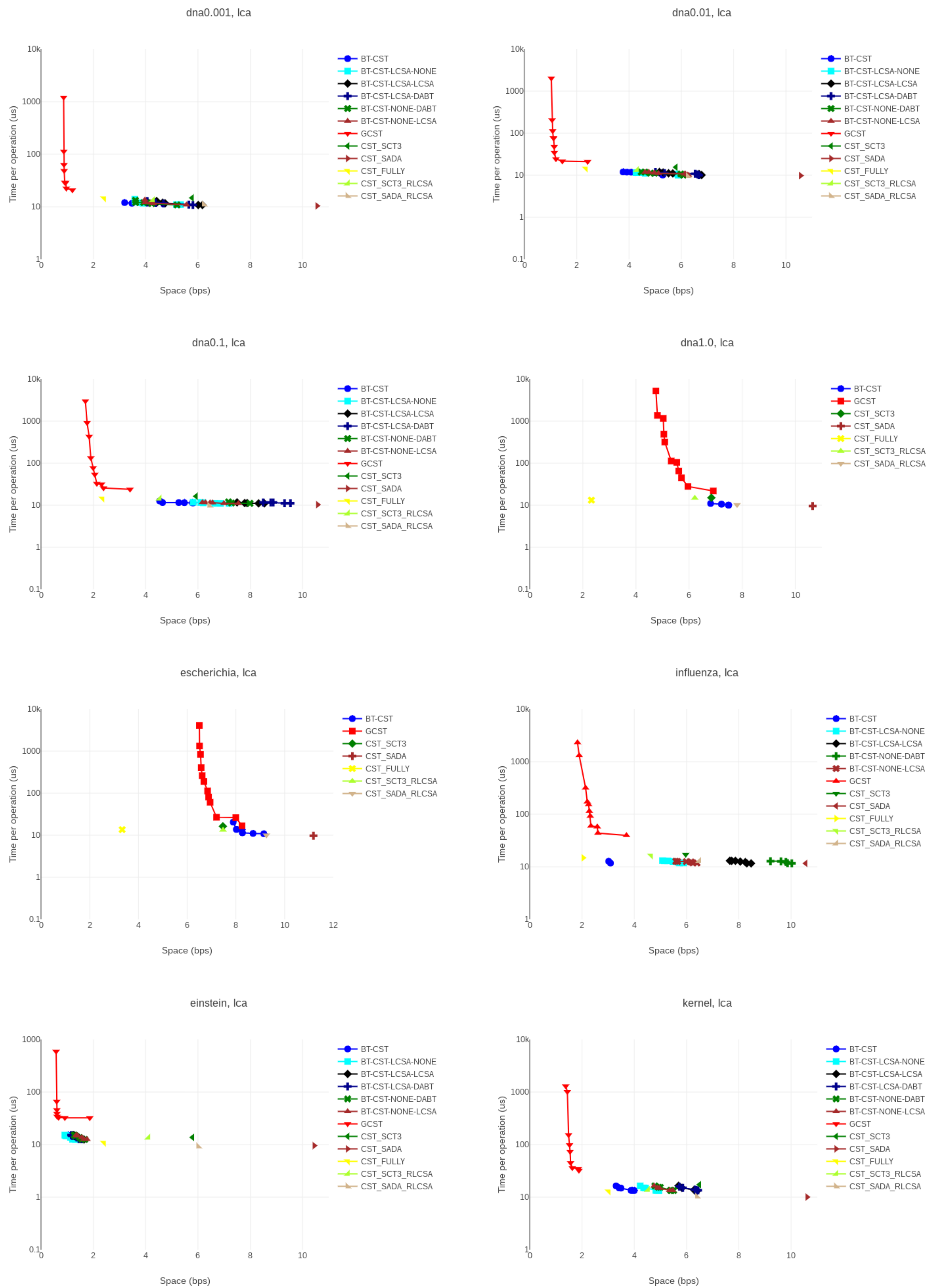


Figure 6.6: Performance of *lca* in different CSTs. The y-axis is time in microseconds in log-scale.

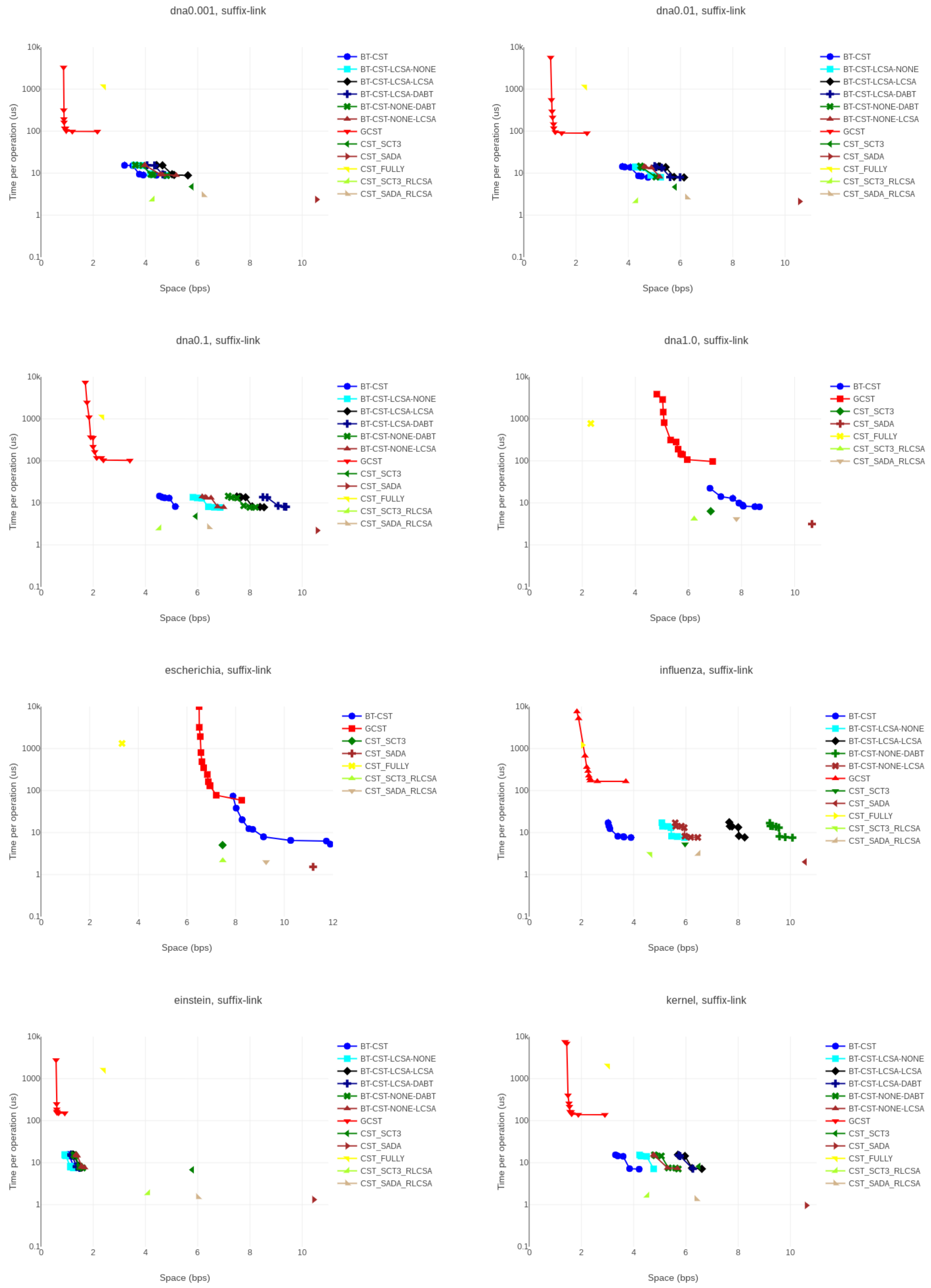


Figure 6.7: Performance of *suffix-link* in different CSTs. The y-axis is time in microseconds in log-scale.

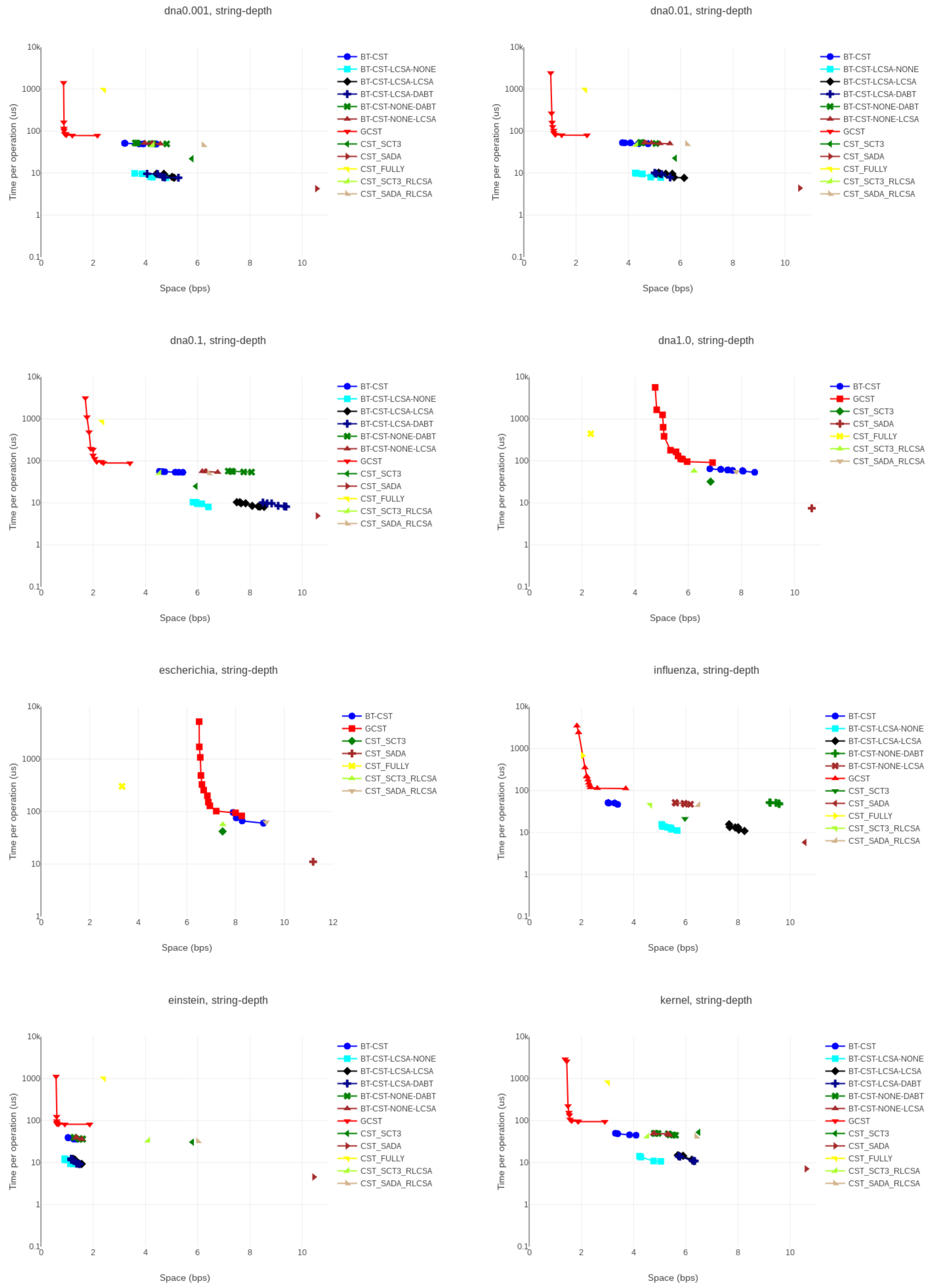


Figure 6.8: Performance of *string-depth* in different CSTs. The y-axis is time in microseconds in log-scale.

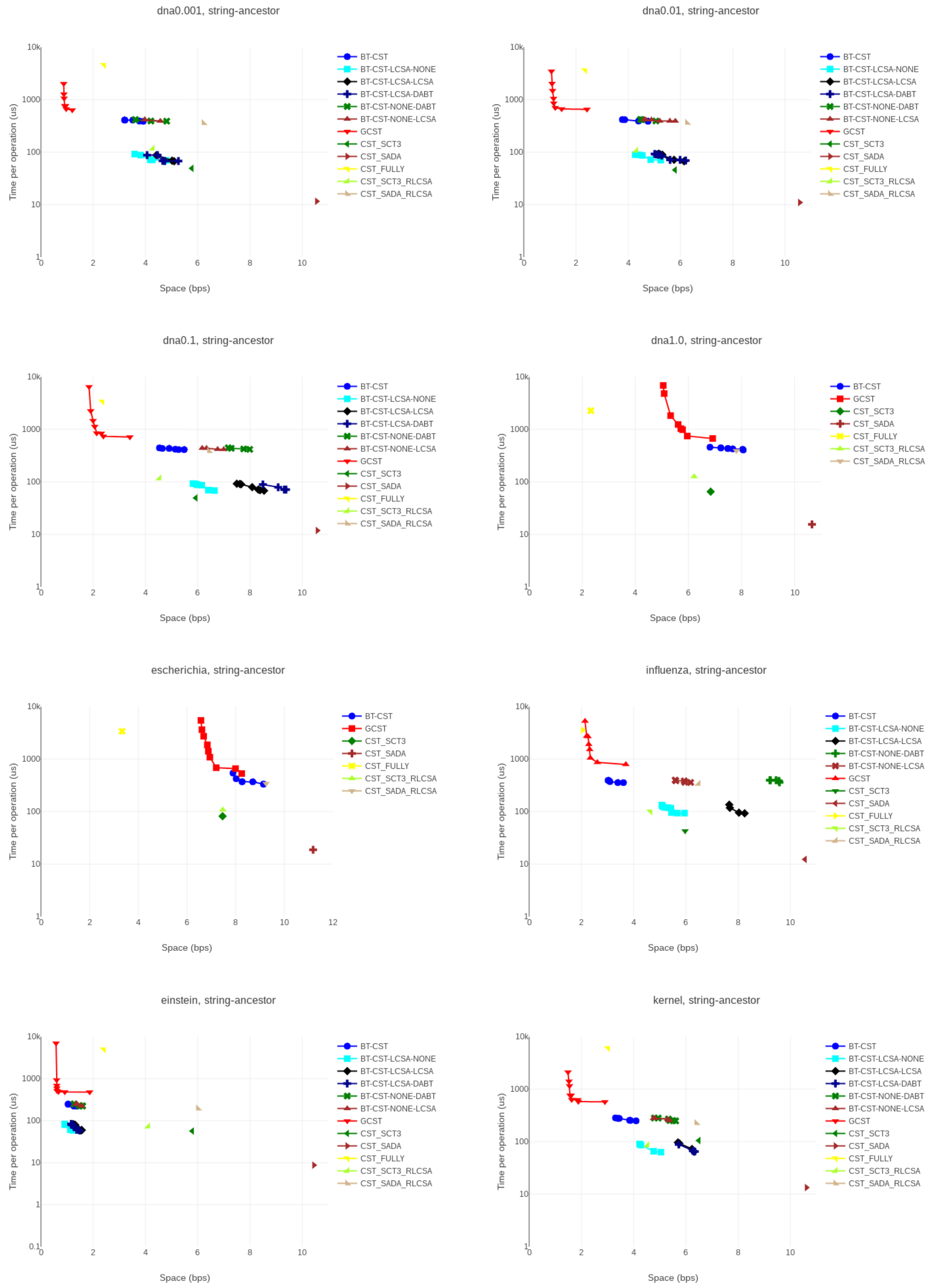


Figure 6.9: Performance of *string-ancestor* in different CSTs. The y-axis is time in microseconds in log-scale.

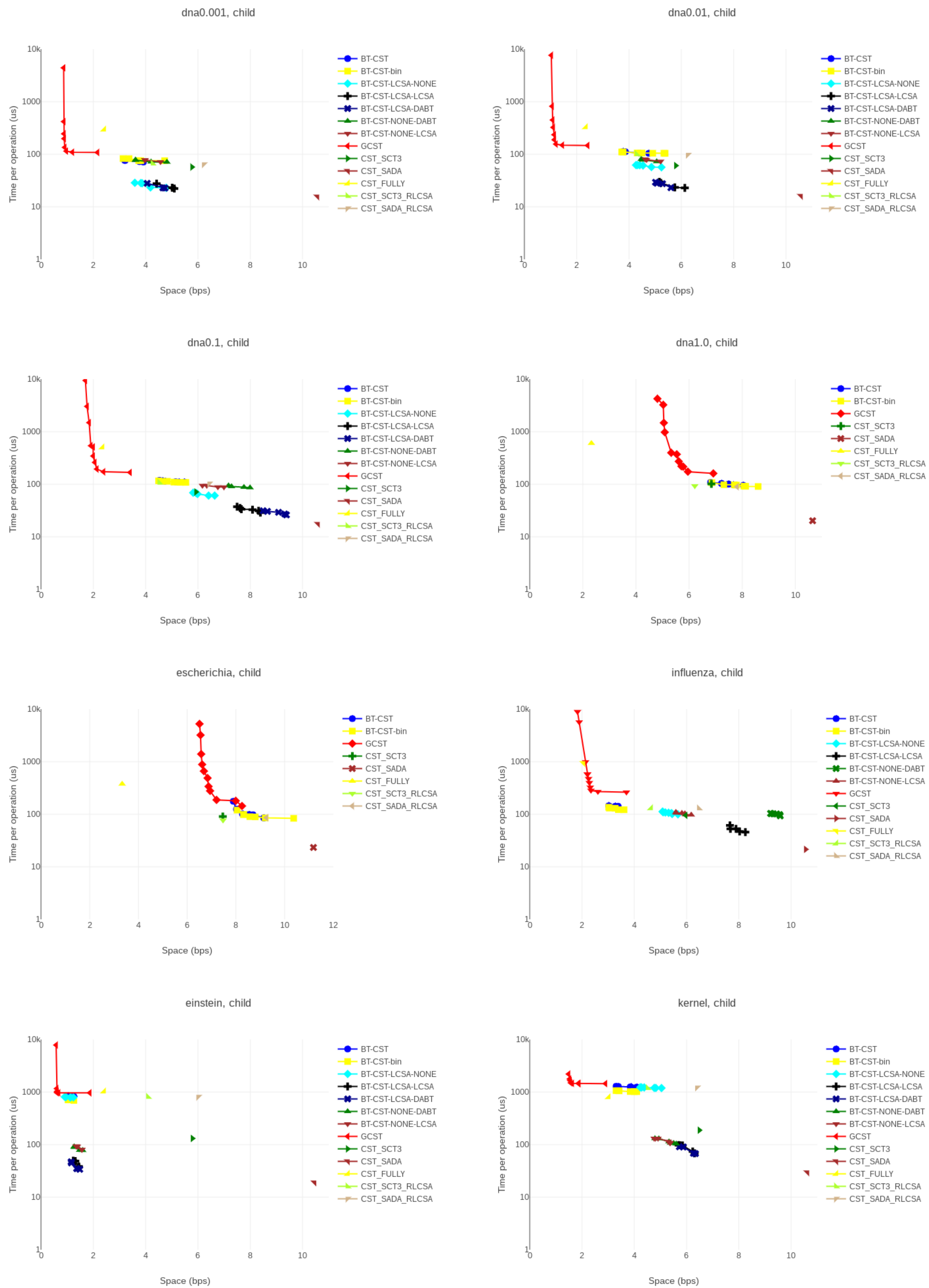


Figure 6.10: Performance of *child* in different CSTs. The y-axis is time in microseconds in log-scale.

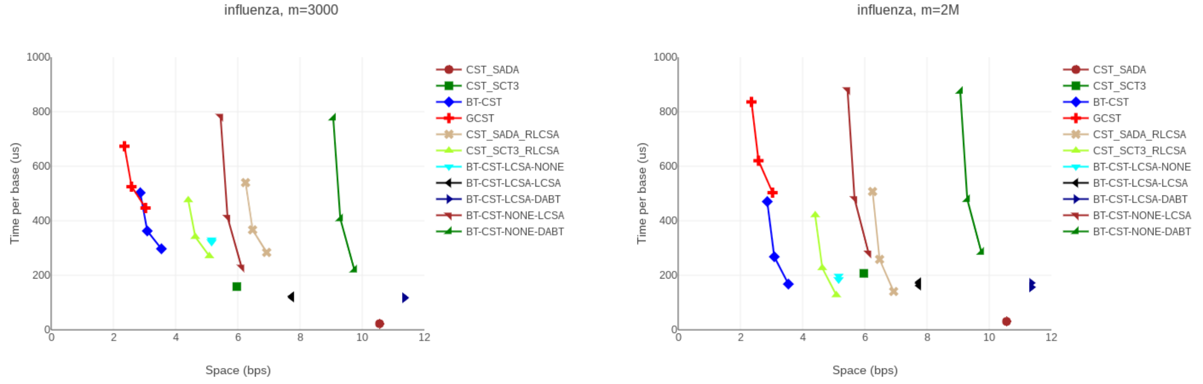


Figure 6.11: Performance of CSTs when solving the maximal substrings problem. The y-axis is time in microseconds per base in the smaller sequence (of length m).

a RLCSA. Our smallest BT-CST variants are BT-CST and BT-CST-LCSA-NONE and the largest are BT-CST-LCSA-LCSA and 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 6.1 and 6.2), which uses the basic *access* and *rank* primitives, CST_SADA[_RLCSA] solves these primitives using techniques of plain bitvectors (see Section 1.3.1 on page 6), which yields an order of magnitude of advantage over BT-CT and two orders over GCT. CST_SCT3[_RLCSA] and CST_FULLY are orders of magnitude slower in these operations as they do not store the topology explicitly.

In operations *next-sibling*, *parent* and *level-ancestor* (Figures 6.3, 6.4 and 6.5), 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. 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.

Operation *lca* (Figure 6.6), 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.

Note that for tree topology operations (*first-child*, *tree-depth*, *next-sibling*, *parent*, *level-ancestor* and *lca*), the time behavior of our BT-CST variants is the same, because the difference between them is their underlying CSA. The same occurs between CST_SADA and CST_SADA_RLCSA, and between CST_SCT3 and CST_SCT3_RLCSA.

Operation *suffix-link* (Figure 6.7) 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 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. Our BT-CST variants have the same behavior as only Ψ is applied, so all of them use the Ψ function from the RLCSA, which is faster.

Operations *string-depth* and *string ancestor* (Figures 6.8 and 6.9) 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 one order of magnitude slower in these operations. Our variants using a differential encoding of A present about one order of magnitude time improvement, as the *access* to A is improved.

Finally, operation *child* (Figure 6.10) is the most expensive, 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 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. 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 millisecond. Our variants of BT-CST behave differently depending on the input sequence: on the *dna{0.001, 0.01, 0.1, 1.0}*, 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.

Figure 6.11 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*. Our BT-CST variants using a differential encoding of A present significant time improvements, but at cost of several bps.

We offer all our variants in the code publicly available at `https://github.com/elarielcl/BT-CST`. For researchers and practitioners we recommend to use two representative options: BT-CST as the low space version using BT-CT, and BT-CST-LCSA-NONE as the next low space alternative, achieving orders of magnitude improvements on operations involving A. In the case of highly repetitive inputs, *einstein* in our datasets, we recommend BT-CST-LCSA-NONE, which turns to be both time and space superior to its relatives.

Conclusion 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 [10] 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 [53], 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 2.79-bit-per-node representation that is blind to repetitiveness [55]. 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 [55], 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 [55] and even on balanced-grammar parse trees [53], 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. [10] claiming logarithmic time for *min-excess* does not really solve the operation (as checked with coauthor T. Gagie).

Considering the good performance of BT-CT we proposed a way to improve the operation *child*, replacing the sequential search by a binary search on their children. We got a small time improvement with this new approach, and note that it can be applied to other CSTs representing its topology, for example GCT or CST_SADA[_RLCSA]. 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 get improvements of one order of magnitude on these operations. We also note that this enhancement of the RLCSA could be applied to any CST using RLCSA, for example, GCST, CST_SADA_RLCSA or CST_SCT3_RLCSA, which will give us a new range of possibilities.

Our BT-CT is the first repetition-aware topology representation whose times are similar to the statistical rmM-tree. It remains to compare the performance of the rest of topology op-

erations that were not implemented on its competitors. The BT-CT have them implemented, and moreover, it has implemented more complex operations not mentioned in this work, for example, *height* of *deepest-node*. To further reduce space we note that we could have applied a sampling technique on the augmentation of the nodes similar to the sampling applied in GCT, that is, we do not augment every node in the Block Tree but only a specific sampling of them and compute the fields of no-sampled nodes on the fly. To further reduce time we note that some topology operations internally call various primitives, which do redundant work, for example, the *lca* operation internally calls to *min-excess* and later *fwd-search* for this minimum, but in the traversal for *min-excess* we discover a block containing the position of this minimum, which could be used as precomputed information for *fwd-search*. Note that this technique could be adapted as a cache technique for all the BP representations (GCT, rmM-tree, BT-CT).

It is worth to mention that we obtained on preliminary experiments that Block Trees better compress topologies starting with regular patterns, for example, many open parentheses, which is not the case of suffix tree topologies (as they start with the leaves representing the last suffixes). We think that moving the leaves of suffix trees to the position of the last child could improve the compression given by Block Trees and the algorithms for suffix trees would not suffer dramatic changes.

We also have shown that it is possible to adapt Block Trees to obtain a representation of differential encodings of arrays, however, it uses significantly more space (compared to their Grammar-compressed counterpart) when dealing with suffix arrays and LCP arrays of repetitive sequences; we do not have a clear explanation for this behavior.

Our work includes a series of improvements on the previous work on basic Block Trees. We implemented the first Block Tree faithful to its theoretical proposal, *bt*, but also a lot of different versions and heuristics that behave very similarly in practice to *bt*. It remains still open a better understanding of the worst-case time and space complexities of these heuristics. It also remains open to reduce the construction time and space needed for Block Trees, whose final result can easily reside inside the memory of a typical personal computer, but that requires much more space at construction time.

Finally, all our implementations have been made public to aid practitioners on using and experimenting with the presented structures. The code is available on the GitHub repository `https://github.com/elarielcl/BT-CST`; another GitHub repository containing only the Block Tree can be found at `https://github.com/elarielcl/BlockTrees`. An important object that we think the area of repetition-aware compressed data structures lacks of is the existence of a unified library containing all the structures presented in this area, which would be of great importance to practitioners using them, and for the further development of the research area itself. We think this library could be a self-contained library such as `sdsl` or an extension of itself.

Bibliography

[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] Diego Arroyuelo, Veronica Gil-Costa, Senén González, Mauricio Marin, and Mauricio Oyarzún. Distributed search based on self-indexed compressed text. *Information Processing & Management*, 48(5):819–827, 2012.

[6] Diego Arroyuelo, Senén González, Mauricio Marin, Mauricio Oyarzún, and Torsten Suel. To index or not to index: time-space trade-offs in search engines with positional ranking functions. In *Proc. 35th International ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 255–264, 2012.

[7] Jérémy Barbay and Gonzalo Navarro. On compressing permutations and adaptive sorting. *Theoretical Computer Science*, 513:109–123, 2013.

[8] 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.

[9] 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.

[10] 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.

[11] Nieves R Brisaboa, Susana Ladra, and Gonzalo Navarro. Dacs: Bringing direct access

to variable-length codes. *Information Processing & Management*, 49(1):392–404, 2013.

[12] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

[13] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959.

[14] David Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1997.

[15] 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.

[16] Francisco Claude, Gonzalo Navarro, and Alberto Ordónez. The wavelet matrix: An efficient wavelet tree for large alphabets. *Information Systems*, 47:15–32, 2015.

[17] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. John Wiley & Sons, 2012.

[18] O'Neil Delpratt, Naila Rahman, and Rajeev Raman. Engineering the LOUDS succinct tree representation. In *Proc. 5th Symposium on Experimental Algorithms (SEA)*, pages 134–145, 2006.

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

[20] 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.

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

[22] 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.

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

[24] Travis Gagie. Large alphabets and incompressibility. *Information Processing Letters*, 99(6):246–251, 2006.

[25] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-Time Text Indexing in BWT-runs bounded space. In *Proc. 29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1459–1477, 2018.

[26] 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.

[27] 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.

[28] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

[29] 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.

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

[31] 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.

[32] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[33] Guy Jacobson. Space-efficient static trees and graphs. In *Proc. 30th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[34] Artur Jeż. A really simple approximation of smallest grammar. *Theoretical Computer Science*, 616:141–150, 2016.

[35] Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[36] Dominik Kempa and Nicola Prezza. At the roots of dictionary compression: string attractors. In *Proc. 50th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 827–840, 2018.

[37] 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.

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

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

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

[41] 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.

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

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

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

[45] 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.

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

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

[48] 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.

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

[50] Gonzalo Navarro. Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences. *ACM Computing Surveys*, 46(4):52, 2014.

[51] Gonzalo Navarro. *Compact data structures: A practical approach.* Cambridge University Press, 2016.

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

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

[54] Gonzalo Navarro, Simon J. Puglisi, and Daniel Valenzuela. General document retrieval in compact space. *ACM Journal of Experimental Algorithmics*, 19:2–3, 2015.

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

[56] Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical structure in se-

quences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

[57] 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.

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

[59] Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 9th Workshop on Algorithm Engineering & Expermiments (ALENEX)*, pages 60–70, 2007.

[60] Alberto Ordóñez, Gonzalo Navarro, and Nieves R Brisaboa. Grammar compressed sequences with rank/select support. *Journal of Discrete Algorithms*, 43:54–71, 2017.

[61] Alberto Ordóñez Pereira. *Statistical and repetition-based compressed data structures*. PhD thesis, University of A Coruña, 2015.

[62] 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.

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

[64] 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.

[65] Wojciech Rytter. Application of Lempel–Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1-3):211–222, 2003.

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

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

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

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

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

[71] 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.

[72] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[73] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.