Karlsruhe Institute of Technology

Engineering Block Trees

Master Thesis

of

Daniel Meyer

At the KIT Department of Informatics
Institute of Theoretical Informatics
Algorithm Engineering

Reviewer:	Prof. Dr. Peter Sanders
Advisor:	Dr. Florian Kurpicz

30. June 2022 – 30. December 2022

Institute of Theoretical Informatics
Algorithm Engineering
KIT Department of Informatics
Karlsruhe Institute of Technology
Am Fasanengarten 5
76131 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.
Karlsruhe, April 24, 2023

. . . .
(**Daniel Meyer**)

Abstract

Engineering Block Trees

Block trees are a data structure to store a string S of length n with z Lempel-Ziv factors in $\mathcal{O}(z \log{(n/z)})$ space, while allowing direct symbol access in time $\mathcal{O}(\log{(n/z)})$. While a Lempel-Ziv encoding of S only takes $\mathcal{O}(z)$ space, direct symbol access is not possible without decoding the whole string. Furthermore, block tree also supports other queries like rank and select, that are commonly used in e.g., full-text indexes. In this thesis, we introduce a novel block tree construction algorithm based on a well-researched data structure, the longest previous factor array. Our algorithm constructs the block tree in $\mathcal{O}(n)$ time, but requires $\mathcal{O}(n) \log n$ space. This is more than the $\mathcal{O}(n)$ space of the previous approaches. In our extensive experimental evaluation, we show that our novel approach is 2.6 to 14.75 times faster than previous algorithms on highly repetitive texts.

Zusammenfassung

Engineering Block Trees

Block Trees sind eine Data Struktur, die eine Zeichenkette der Länge n mit z Lempel-Ziv Faktoren in $\mathcal{O}(z \log{(n/z)})$ Platz speichern und direkten Zugriff auf beliebige Symbole in $\mathcal{O}(\log{(n/z)})$ Zeit erlauben. Die bekannte Lempel-Ziv Codierung braucht zwar nur $\mathcal{O}(z)$ Platz, erlaubt aber keinen Zugriff auf beliebige Symbole, ohne zuvor den gesamten Text zu dekodieren. Zusätzlich können Block Trees auch weitere Anfragen, wie zum Beispiel Rang- oder Auswahlanfragen, beantworten, die häufig in Volltextindexen verwendet werden. In dieser Abschlussarbeit stellen wir einen neuen Block Tree Konstruktionsalgorithmus vor, der Block Trees in $\mathcal{O}(z)$ Zeit und $\mathcal{O}(n \log n)$ Platz konstruiert. Der Algorithmus baut auf dem längsten vorherigen Faktor Array auf. In unseren Experimenten zeigte sich, dass unser Ansatz zwischen 2,6 und 14,75-mal schneller Block Trees für sich stark wiederholenden Texten aufbaut als die bisherigen Algorithmen.

Contents

1 Introduction

The rise in the amount of data we aim to handle and the fact that most of the fastest-growing data is highly repetitive [50], has motivated recent research on exploiting repetitiveness to enable space reductions of two orders of magnitude [21]. In contrast to entropy encoders, dictionary encoders like the Lempel-Ziv compression achieve high compression rates by capturing this repetitiveness [34]. Given a string $S[0..n)$, Lempel-Ziv compression parses S into a set of z factors and encodes S in $\mathcal{O}(z)$ space [46]. Although Lempel-Ziv compression exploits repetitiveness the best in practice, it does not allow for random access on S without decoding S from the beginning. Grammar-based compression [32] is based on finding the smallest context-free grammar that generates S and only S. These grammars allow random access in $\mathcal{O}(\log n)$ time, but generating the smallest grammar is NP-complete [47, 9]. Thus, several linear time approximation, which generate grammars of a size $\mathcal{O}(z \log(n/z))$ were introduced [47, 9, 28]. In this thesis, we take a closer look at the block tree data structure [7], that uses $\mathcal{O}(z \log(n/z))$ space and allows access to any symbol of S in $\mathcal{O}(\log(n/z))$ time. Block trees also support the following two queries with additional data structures:

- $rank_c(i, S)$: return the number of occurrences of the character c in $S[0..i]$.
- $select_c(j, S)$: return the position of the jth occurrence of c in S.

Block trees can be built in linear time and space and allow for time-space-tradeoffs [7]. In their experimental evaluation, Belazzougui et al. [7] showed that the block trees take up about the same space as grammar-compressed representations but can answer queries orders of magnitude faster and provide only a bit slower queries than entropy encoded representation [11, 45, 23], which are several times larger.

1.1 Problem Statement

The current implementation of block trees by Belazzougui et al. [7] shows slow construction times in practice. We want to improve these times by applying ideas taken from LZ77-compression algorithms. Although block trees can be constructed in linear time, given the slow construction time of current implementations, an approximation algorithm could have practical value. Therefore, we want to approximate the block tree data structure and compare it to canonical block trees. In addition, we want to research if it is possible to parallelize the construction process of block trees and achieve a notable speed-up.

1.2 Contribution

We have the following main contributions: first, we present a novel block tree algorithm that links two well-researched data structures, the longest previous factor array and previous occurrences array, to the block tree. Furthermore, we present a greedy heuristic for block trees. We compare our algorithm on various texts with different degrees of repetitiveness and alphabet sizes with the preexisting implementation [7] to show that our algorithm constructs blocks trees without rank support between 2.6 and 14.75 times faster when considering a highly repetitive corpus and between 6.2 and 22.8 times faster on average when considering a standard corpus. Second, we compare our heuristic with our canonical block tree implementation, showing that the heuristic is similar in size, but could not determine any construction time improvements over the canonical version. Finally, we ran an initial experiment on highly repetitive texts, observing the impact of using a known shared memory parallelization for LPF-arrays and simply parallelizing the σ depth first searches required for enabling rank support after the canonical block tree construction, here we observe a modest speedup of 3.6-14.4 on 64 cores (with simultaneous multithreading) with respect to one core.

1.3 Structure of Thesis

The remaining content of this thesis is structured as follows. In Chapter 2, we introduce fundamental definitions and data structures from the field of stringology and give a precise formulation of rank, select and access queries. Chapter 3 follows with a broad overview on the Lempel-Ziv factorization and longest previous factor array, as well as the block tree itself. This is the work that we build upon when developing our block tree construction approach. The main content of this thesis is described in Chapters 4, 5 and 6, where we present our approach to constructing block trees, as well as introduce a heuristic to the block tree and perform an experimental evaluation and comparison to previous work. Finally, we conclude our work in Chapter 7.

2 Fundamentals

This chapter introduces some general definitions necessary to understand the previous work and contributions in the context of this thesis.

Let S be a string of n characters over an alphabet $\Sigma = [0..\sigma)$ stored in an array $S[0..n-1]$. We denote the substring $S[i]S[i+1]...S[j]$ in S as $S[i..j]$. Let suf_i or $S[i..]$ be the suffix starting at $S[i]$ and running to the end of S. The **suffix array** [38] of S, denoted as SA, gives the suffixes of S sorted in ascending lexicographical order, that is $suf_{\texttt{SA}[0]} < suf_{\texttt{SA}[1]} < \cdots < suf_{\texttt{SA}[n-1]}$. Another data structure, often used in combination with the suffix array, is the **longest common prefix array** LCP [30], storing the length of the longest common prefix between consecutive suffixes in SA. $\texttt{LCP}[i]$ is the longest common prefix of and $suf_{\texttt{SA}[i-1]}$ and $suf_{\texttt{SA}[i]}$. To simplify edge cases, we set $\texttt{LCP}[0] = 0$. An example for SA and LCP of $abababbbbaba$ is shown in Figure 3.1.

Empirical Entropy

For a string $S[0..n)$ over an alphabet $[0..\sigma)$, the histogram $\texttt{Hist}[0..\sigma)$ and the cumulative histogram $C[0..\sigma)$ are defined as: $\texttt{Hist}[i] = |\{j \in [0,n) : S[j] = i\}|$ and $\texttt{C}[i] = |\{j \in [0,n) : S[j] < i\}|$ [33]. The **zero-order entropy**

$$\mathcal{H}_0(S) = \frac{1}{n}\sum_{i=0}^{\sigma} \texttt{Hist}[i] \log\left(\frac{n}{\texttt{Hist}[i]}\right) \tag{2.1}$$

is the minimum average number of bits needed to represent the symbols of S if we use the same code to represent all occurrences of the same character $c \in [0..\sigma)$ [33]. The **kth-order entropy**

$$\mathcal{H}_k(S) = \frac{1}{n}\sum_{T\in\Sigma^k} |S_T| \cdot \mathcal{H}_0(S_T) \tag{2.2}$$

is the minimum average number of bits needed to represent the symbols of S, but we chose each code depending on the k preceding characters as context. Given a string S over an alphabet Σ and $T \in \Sigma^k$, S_T is the concatenation of all characters that occur in S after T in text order [33]. Any entropy encoder using k-length contexts must use at least $n\mathcal{H}_k(S)$ bits to represent S. A well-researched entropy encoder is the **Huffman Code** [24], whose encoding for S uses less than $n\lceil\mathcal{H}_0(S)\rceil$ bits and is therefore optimal [24, 48, 15].

Rank/Select/Access

In the field of Stringology, efficient algorithms often make use of rank, select, and access queries to solve various problems, such as inverted indices [2, 3, 4], full text indices

[17, 18] and document listing [40]. We define rank, select and access in the following way:

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

Bit Vectors

A bit vector B provides rank, select and access support over a bitmap $\{0, 1\}$ of length n. B supports $rank(i)$ and $select(i)$ queries in constant time using only $o(n)$ bits extra space [10, 25]. Bit vector are an underlying data structure in well-researched data structures like wavelet trees [23] or succinct graph representations [26, 39].

Wavelet Trees

A wavelet tree is a data structure, that provides rank, select and access support and is used in full-text indices (e.g., the FM-index [17] and r-index [18]). It uses $n\lceil\log\sigma\rceil(1 + o(1))$ bits space [23]. Given a string $S[0..n)$ over an alphabet $[0, \sigma)$, a wavelet tree [23] is a binary tree, where each node represents characters in sub-alphabet $[l, r] \subseteq [0, \sigma)$. The root partitions the alphabet into two roughly equal sizes sub-alphabets $[0, \sigma/2)$ and $[\sigma/2, \sigma]$, and which are represented by its left and right child. We repeat this on each node until the associated sub-alphabet reaches size 1. Characters are represented using a bit vector, where an entry is 1 if the character is represented in the right child and 0 if it is represented in the left child.

To solve access and rank queries, we traverse root to leaf, using access and rank queries on the bit vectors to translate the query to the sub-alphabets. For select queries, we traverse leaf to root, using select queries on the bit vectors to translate the index to the current sub-alphabet. For a string S over an alphabet of size σ, the wavelet tree of the string can answer access, rank and select queries in $\mathcal{O}(\log\sigma)$ [23].

The wavelet matrix [12] is an adaption of the wavelet tree, where the bit vectors in a level are permuted and concatenated. Wavelet trees and the wavelet matrix can be compressed using entropy encoders. As wavelet trees cannot handle holes in the alphabet, we use canonical Huffman codes. This reduces the required space for the Huffman-shaped wavelet tree to $n\lceil\mathcal{H}_0(S)\rceil(1 + o(1))$ bits [37].

3 Related Work

3.1 Lempel-Ziv Factorization

Lempel-Ziv Factorization [36] of a string $S[0..n)$ parses S into z phrases where $S = \omega_0\omega_1...\omega_{z-1}$. Each phrase ω_i is either a new symbol, that has not appeared in S before ω_i and $|\omega_i| = 1$, or it is the longest substring that occurs at lest twice in $\omega_0..\omega_i$. Consider the following example: $S = ababababbbbaba$ has a factorization $S = \omega_0\omega_1\omega_2\omega_3\omega_4$, where $z = 5$, $\omega_0 = a$, $\omega_1 = b$, $\omega_2 = abab$, $\omega_3 = bbb$ and $\omega_4 = aba$. `LZ77` [54] represents each factor ω_i as a tuple $(\ell_i, prev_i)$. If a factor is a single character c and $|\omega_i| = 1$ we set ℓ_i to 0 and store c encoded in $prev_i$. Otherwise, we set ℓ_i to $|\omega_i|$ and $prev_i$ to the starting position of a leftwards occurrence of ω_i. A list of tuples for the string $S = ababababbbbaba$ is $[(0, a), (0, b), (4, 0), (3, 5), (3, 0)]$. Note that for $\omega_4 = aba$ we can choose between two leftwards occurrences and can store either $(3, 0)$ or $(3, 2)$. To decode, each tuple i can restore its factor by looking at $prev_i$ and either restore the stored character for $\ell_i = 0$ or otherwise copy the ℓ_i characters starting at position $prev_i$. Lempel-Ziv compression has the problem that it cannot access an arbitrary substring of S from the decompressed state. If we want to access any substring of S, it is necessary to decompress S from the beginning. The Lempel-Ziv compression algorithm reduces S to $\mathcal{O}(z)$ space in $\mathcal{O}(n)$ time and space [46].

Algorithm 1: `LPF` to `LZ77` [13]

```
   Data: LPF$[0..n)$
   Result: LZ$[0..z)$
1  $i \leftarrow 0$;
2  LZ$[0] \leftarrow 0$;
3  while LZ$[i] < n$ do
4      LZ$[i+1] \leftarrow$ LZ$[i]$ + max(1, LPF[LZ$[i]$]);
5      $i \leftarrow i + 1$;
```

3.2 LPF-Array

We denote the Lempel-Ziv factorization as an array `LZ` of size z where `LZ`$[i]$ stores the starting position of ω_i in S. The pair $(\ell_i, prev_i)$ can be computed with the **previous occurrence array** introduced in Section 3.2 [49].

For a string $S[0..n)$ and $i \in [0..n-1)$, Franěk et al. [20] introduced, the **longest previous factor array** `LPF`. `LPF`$[0..n)$ is defined by `LPF`$[i] = max\{k \mid S[i..i+k)$ occurs at a position $j < i\}$. The **previous occurence array** `PrevOcc` [49] stores in `PrevOcc`$[i]$

a starting location of the longest previous factor $S[i..i+k)$ with $k := \texttt{LPF}[i]$ of $S[i..]$ or -1 if $S[i..]$ has no longest previous factor. We can consider this as an extension of the Lempel-Ziv factorization where we determine the longest previous factor for all suffix $S[i..n), i \in [0..n)$ not just for the starting positions of the Lempel-Ziv phrases $\omega_0, \omega_1..\omega_{z-1}$. Algorithm 1 by Crochemore and Ilie [13] computes the LZ77-factorization with a single pass over the LPF-array.

3.2.1 Sequential Construction Using SA and LCP

Due to the lexicographic ordering of suffixes in SA the LCP-array has the following simple property [14]:

Lemma 3.2.1. *The length of the* lcp *value between two position at ranks r and t ,* $(r < t)$ $\texttt{lcp}(r,t)$*, is the minimal value in* $\texttt{LCP}[r+1..t]$.

Therefore, we can calculate $\texttt{lcp}(r,t)$ using range minima queries on LCP. Range minima queries provide the minimal value in $\texttt{LCP}[r+1..t]$ in $\mathcal{O}(1)$ time using $\mathcal{O}(n)$ space and can be computed in $\mathcal{O}(n)$ time [19].

Definition 1. ***(all nearest smaller values (ANSV)** [49]) For each element in an array of totally ordered objects, find the closest smaller element to the left and the closest smaller element to the right of it (if there is no smaller element, then report it).*

An algorithm for **ANSV** returns two arrays prev and next. Where $\texttt{prev}[i]$ ($\texttt{next}[i]$) contains the index of the nearest smallest value to the left (right) of i-th value or -1 if there is no smaller value to the left (right). Barbay et al. [5] propose a simple linear time Algorithm 2 to calculate prev. To simplify edge cases, we assume an artificial minimum at $\texttt{SA}[-1]$. The algorithm follows the already computed values, as nothing in between can be the previous smallest value. Overall, we compare each element at most twice [5]. Calculating next works analogously. Crochemore et al. [14] introduced Lemma 3.2.2

Algorithm 2: Compute prev for SA [5]

```
  Data: $\texttt{SA}[0..n)$
  Result: $\texttt{prev}[0..n)$
1 for $i \leftarrow 0$ to $n-1$ do
2     $j \leftarrow i-1$;
3     while $\texttt{SA}[i] \geq \texttt{SA}[i]$ do
4         $j \leftarrow \texttt{prev}[j]$;
5     $\texttt{prev}[i] \leftarrow j$;
```

stating that $\texttt{LPF}[i]$ can be computed using **ANSV** and range minima queries on SA and LCP. To simplify edge cases, we assume that $S[\texttt{SA}[-1]..]$ evaluates to the empty string and therefore $\texttt{LCP}[0] = 0$ and $S[\texttt{SA}[-1]..]$ has a longest common prefix of 0 with any string.

Lemma 3.2.2. *Let* $\texttt{prev}[i]$ *and* $\texttt{next}[i]$ *be the left and right nearest smaller neighbors of element* i *in* SA*, then* $\texttt{LPF}[i] = max(lcp(suf_{\texttt{SA}[i]}, suf_{\texttt{SA}[\texttt{prev}[i]]}), lcp(suf_{\texttt{SA}}, suf_{\texttt{SA}[\texttt{next}[i]]}))$, *with* $suf_{\texttt{A}[i]} = S[\texttt{A}[i]..]$.

With, Lemma 3.2.1 we can now use range minima queries to compute the lcp-values and therefore the LPF values [29]. PrevOcc[i] is set to $\texttt{prev}[i]$, if $suf_{\texttt{SA}[\texttt{prev}[i]]}$ has a longer lcp with $suf_{\texttt{SA}[i]}$, and $\texttt{next}[i]$ otherwise [49]. The steps for LPF array construction are summarized in Algorithm 3 below, we can compute the LZ77-factorization by applying Algorithm 1 afterwards. Note that each step takes $\mathcal{O}(n)$ time, and therefore calculating the LPF-array takes $\mathcal{O}(n)$ time.

Algorithm 3: Compute LPF

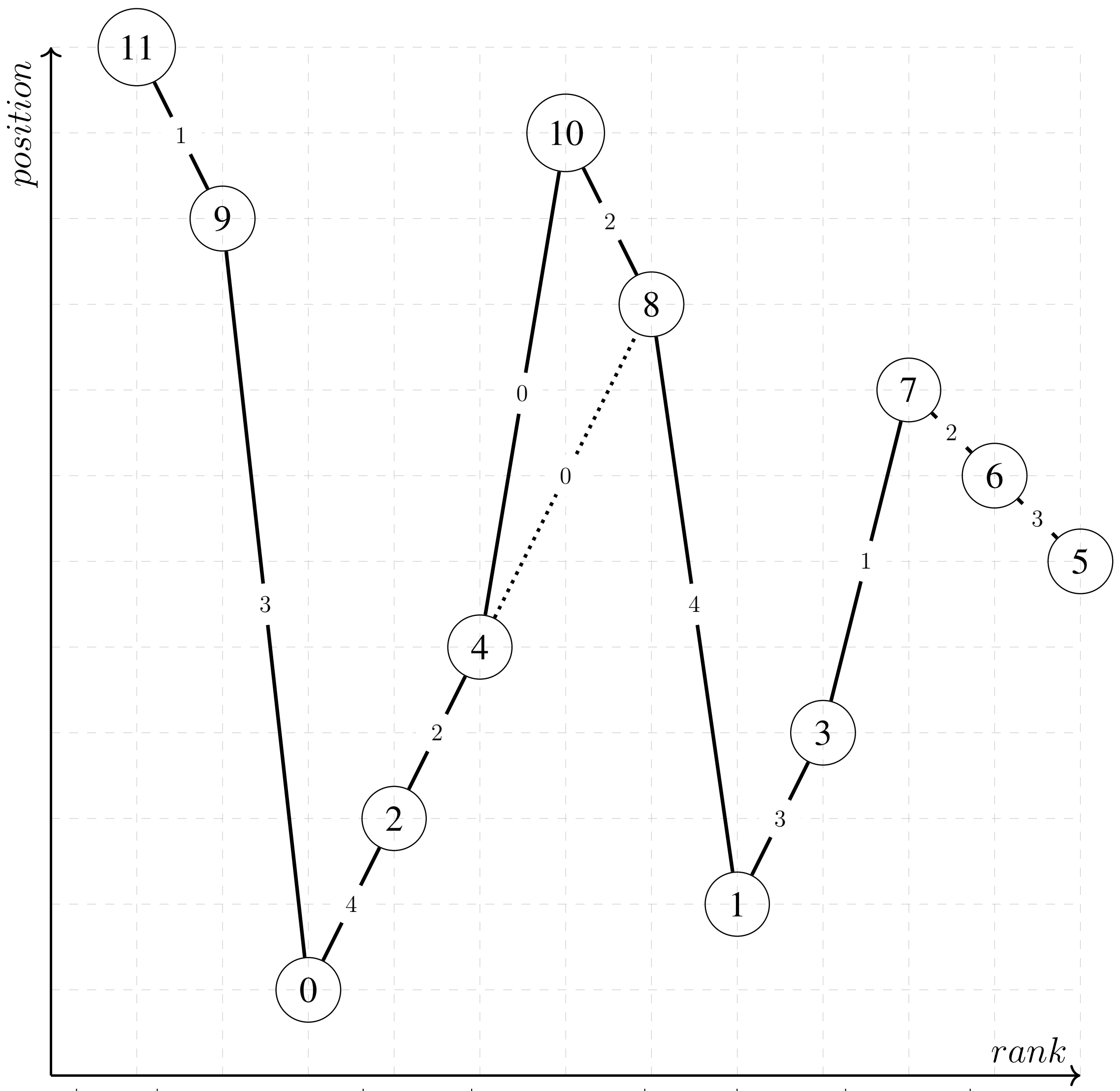
```
  Data: $S[0..n)$
  Result: $\texttt{LPF}[0..n)$, $\texttt{PrevOcc}[0..n)$
1 Compute SA and LCP for $S$;
2 Compute prev and next using Algorithm 2;
3 Compute the LPF and PrevOcc using $RMQs$;
```

The previously shown Algorithm 3 is rather space consuming. At its peak the algorithm needs to allocate six integer arrays (SA, LPF, prev, next, LPF and PrevOcc) of size n, therefore Crochemore et al. [14] proposed Algorithm 4 where prev and next are replaced by a stack, that requires at most $\mathcal{O}(\sqrt{n})$ additional space. Chrochemore

Algorithm 4: Compute LPF of S with SA and LCP [14]

```
   Data: $\texttt{SA}[0..n)$ and $\texttt{LCP}[0..n)$ of $S$
   Result: $\texttt{LPF}[0..n)$ of $S$
1  emptyStack($\mathcal{S}$);
2  for $r \leftarrow 0$ to $n-1$ do
3      $lcp_r \leftarrow \texttt{LCP}[r]$;
4      while notEmpty($\mathcal{S}$) do
5          $(t, lcp_t) \leftarrow$ top($\mathcal{S}$);
6          if $\texttt{SA}[r] < \texttt{SA}[t]$ then
7              $\texttt{LPF}[\texttt{SA}[t]] \leftarrow$ max($lcp_t, lcp_r$);
8              $lcp_t \leftarrow$ min($lcp_t, lcp_r$);
9              pop($\mathcal{S}$);
10         else if $(\texttt{SA}[r] > \texttt{SA}[t])$ and $(r - lcp \leq t - lcp)$ then
11             $\texttt{LPF}[\texttt{SA}[t]] \leftarrow lcp_t$;
12             pop($\mathcal{S}$);
13         else
14             break ;
15     push($\mathcal{S}$, $(r, lcp_r)$);
```

et al. [14] introduced a graphical representation for the suffix array, to better explain the computation of LPF from SA and LCP. Here the x-axis represents the ranks of suffixes in the suffix array i and the y-axis represents their positions $\texttt{SA}[i]$. Suffixes are plotted in lexicographic order by their position ($\texttt{SA}[i]$ for the i-th ranked suffix) and consecutive nodes



i	$S[i]$	$S[i..]$	LPF[i]	PrevOcc[i]	SA[i]	LCP[i]	prev[i]	next[i]
0	a	$abababbbbaba$	0	−1	11	0	−1	1
1	b	$bababbbbaba$	0	−1	9	1	−1	2
2	a	$ababbbbaba$	4	0	0	3	−1	−1
3	b	$babbbbaba$	3	1	2	4	2	7
4	a	$abbbbaba$	2	2	4	2	3	7
5	b	$bbbbaba$	1	3	10	0	4	6
6	b	$bbbaba$	3	5	8	2	4	7
7	b	$bbaba$	2	6	1	4	2	−1
8	b	$baba$	4	1	3	3	7	−1
9	a	aba	3	0	7	1	8	10
10	b	ba	2	8	6	2	8	11
11	a	a	1	9	5	3	8	−1

Figure 3.1: Solid edges form the graph representing the SA and LCP for the string $abababbbbaba$. The dotted edge between 4 and 8 hint at the conceptual transformation that takes place after processing the peak 10. Afterward, SA, LCP, LPF, PrevOcc, prev and next for $abababbbbaba$.

with ranks $i, j = i+1$ are connected with a solid edge labeled with $\texttt{LCP}[j]$. The graphical representation of $S = abababbbbaba$ is shown in Figure 3.1. Algorithm 4 relies on the observation that, if a position $\texttt{SA}[r]$ at rank r is a "peak" in the graphical representation, we get $\texttt{LPF}[\texttt{SA}[r]] = \texttt{max}(\texttt{LPF}[r], \texttt{LPF}[r+1])$ (similar to Lemma 3.2.2) and in the longest common prefix between the position at rank $r-1$ and $r+1$ is $\min(\texttt{LPF}[r], \texttt{LPF}[r+1])$ [14]. These "peaks" are considered first.

For example, "peak" node 10 (rank $r = 5$) and the adjacent nodes 4 and 8 with solid edges labeled 0 and 2 respectively. For "peaks" these adjacent nodes are also the previous and next smallest values in SA. We apply the observation and set $\texttt{LPF}[10]$ set $2 = \texttt{max}(0, 2)$. We can now also conclude that the longest common prefix between 4 and 8 is $0 = \min(\texttt{LPF}[5], \texttt{LPF}[6]$ (dotted edge labeled 0 between 4 and 8 in the graphical representation). In other words because the adjacent suffixes $S[4..]$ and $S[10..]$ have a longest common prefix of length 0, all suffixes with a higher rank than $S[10..]$ also have a longest common prefix of length 0 with $S[4..]$ and suffixes of lower rank than $S[4..]$. We could now proceed and determine $\texttt{LPF}[8]$ with 8 being a "peak" between 4 and 1.

Algorithm 4 considers the nodes by rank (left to right in the graph) and stores non-peak nodes in the stack $\mathcal{S}$. Whenever we encounter a node with a smaller position than the top of $\mathcal{S}$, we can process and pop the top of $\mathcal{S}$. All nodes with a higher position to the left of the top of $\mathcal{S}$ are already processed, and we can consider the top of $\mathcal{S}$ a "peak". For example, 4 has to wait until 10 and 8 are processed. We store a pair (t, lcp_t) on the stack $\mathcal{S}$, containing the position t of a node and the longest common prefix between the $S[t..]$ and the suffix, corresponding to the element right below on the stack $\mathcal{S}$. Algorithm 4 also implements an optimization, where for a suffix of rank r with $\texttt{LCP}[r] \geq \texttt{LCP}[r+1]$, no position to right can provide a larger LCP value, and therefore we get $\texttt{LCP}[\texttt{SA}[r]] = \texttt{LCP}[r]$.

We simplify the algorithm by extending SA to rank n and initialize $\texttt{SA}[n] = -1$ and $\texttt{LCP}[n] = 0$. To increase readability, we exclude calculating PrevOcc. This can be done by adding another if-clause after assigning a new value to $\texttt{LPF}[\texttt{SA}[t]]$ where set $\texttt{PrevOcc}[\texttt{SA}[t]]$ to -1 if $\texttt{LPF}[\texttt{SA}[t]]$ is 0, $\texttt{PrevOcc}[\texttt{SA}[t]]$ to t when $lcp_t > lcp_r$ and too $\texttt{SA}[r]$ otherwise.

3.2.2 Practical CRCW PRAM Parallel Construction

Shun and Zhao [49] proposed a linear-work parallel algorithm for LZ-factorization on the CRCW PRAM. The algorithm is based on the sequential algorithm Algorithm 1 by Crochemore and Ilie [13], which requires the LPF array. To calculate the LPF-array Shun and Zhao [49] parallelize Algorithm 3. With the algorithm by Kärkkäinen and Sanders [29] both, the SA and LCP, can be calculated in parallel using $\mathcal{O}(n)$ work and $\mathcal{O}(\log^2 n)$ time. Berkman et al. [8] show that **ANSV** can be computed in $\mathcal{O}(n)$ work and $\mathcal{O}(\log \log n)$ time on the CRCW PRAM.

Afterwards, we apply Lemma 3.2.2 and compute the LPF values using range minima queries. Range minima queries take $\mathcal{O}(1)$ time and $\mathcal{O}(1)$ work and need $\mathcal{O}(n)$ work and $\mathcal{O}(\log n)$ time for preprocessing. Shun and Zhao [49] show that performing the n queries in parallel takes $\mathcal{O}(n)$ work and $\mathcal{O}(1)$ time. In conclusion, we can compute the LPF-array in $\mathcal{O}(n)$ work and $\mathcal{O}(\log^2 n)$ time on the CRCW PRAM.

A parallelized Algorithm 1 can then compute the LZ-factorization in $\mathcal{O}(n)$ work and $\mathcal{O}(\log n)$ [16] using a parallel leaffix algorithm and a prefix sum [27]. In their experiments, Shun and Zhao [49] observed a speedup between 6.7 and 21.2 on 40 cores compared to a single core.

3.3 Block Tree

Belazzougui et al. [7] introduced the block tree, a data structure, that represents a string $S[0..)$ in $\mathcal{O}(z \log(n/z) \log(n))$ space with z being the number of Lempel-Ziv phrases (see Section 3.1). Unlike a Lempel-Ziv factorization, block trees allow access to any symbol of S in $\mathcal{O}(\log(n/z))$ time. Furthermore, block trees support rank and select queries.

Let $S[0..n)$ be over the alphabet $[0..\sigma)$, and τ and s be integers greater than 1. To simplify edges cases, we assume $n := s \cdot \tau^h$ for some integer h. For other strings with a different length, we round their length up by appending dummy characters $\$$ until the string is of length $s \cdot \tau^h$ for some integer h. Belazzougui et al. [7] define the block tree as a perfectly balanced tree of height h, which may have leaves at higher levels. The root has s children, and every other internal node has τ children.

Each node u represents a substring of S with a position in S, called a "block" B^u. The root represents S and has s children representing s consecutive blocks, of length $\ell := n/s$, that concatenated make up S. We refer to all blocks with the same depth as a block tree level. For example, $B_0, B_1...B_{s-1}$ is the block tree level of depth 0. Two nodes are considered consecutive if they are in the same block tree level and if the substrings they represent are consecutive in S. Let $B_i \cdot B_{i+1}$ be the concatenated substrings of two consecutive blocks. Non-root nodes that are not in the last level represent one of two block types, either marked blocks: internal blocks with τ children in the next level or unmarked blocks: a leaf with a leftwards pointer. Note that sometimes, we will refer indistinctly to the node and the block represented by that node.

Definition 2. *(marked blocks) For any i, two consecutive blocks B_i, B_{i+1} are marked if $B_i \cdot B_{i+1}$ contains the leftmost occurrence of any substring of S.*

All marked blocks B^v are internal blocks, which have τ children. These children represent the consecutive blocks of length $|B^v|/\tau$ that concatenated make up the content of B^v. On the other hand, the unmarked blocks B^u are now leaves and instead of having children in the next level, we store a pointer towards the pair of consecutive blocks $B_i \cdot B_{i+1}$ containing the leftmost occurrence of B^u and the offset of that occurrence in $B_i \cdot B_{i+1}$.

If the block length becomes small enough that storing the content explicitly takes less space than storing the leftwards pointer for an unmarked block, we reach the last level and store each block explicitly. For example, we store blocks once $|B^u| \in \Theta(\log_\sigma n)$ and hence B^u can be encoded in $\mathcal{O}(n)$ bits. The length decreases by a factor of τ each level. Therefore, the block length of blocks B^u in level ℓ_u is $|B^u| = n/(s\tau^{\ell_u - 1})$. If we take the constant 1 for simplicity, leaves store exactly $\log n$ bits and the height of the block tree is $h = \log_\tau \frac{n \log \sigma}{s \log n}$ [7]. Note that, we can decrease the height h by increasing s and thereby adding $\mathcal{O}(s)$ words of extra space. Decreasing the height h allows for time-space-tradeoffs, because the time complexities of rank, select and access are proportional to h (see Section 3.3.1). Figure 3.2 shows an example of a block tree introduced by Belazzougui et al. [7].

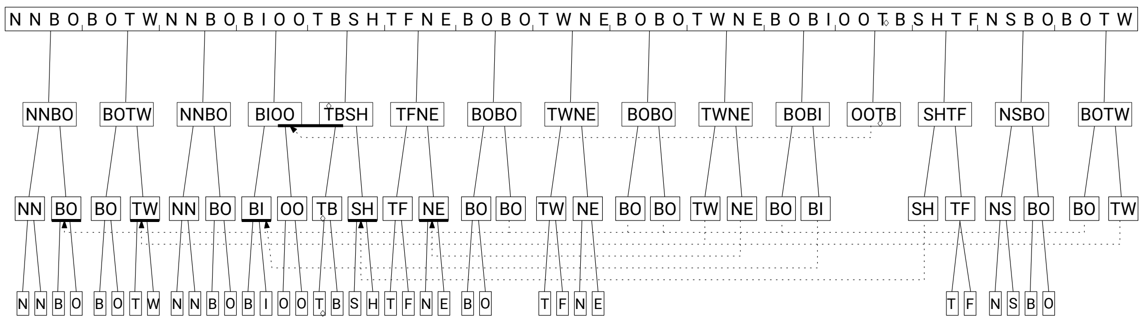

N N B O B O T W N N B O B I O O T B S H T F N E B O B O T W N E B O B O T W N E B O B I O O T B S H T F N S B O B O T W
NNBO BOTW NNBO BIOO TBSH TFNE BOBO TWNE BOBO TWNE BOBI OOTB SHTF NSBO BOTW
NN BO BO TW NN BO BI OO TB SH TF NE BO BO TW NE BO BO TW NE BO BI SH TF NS BO BO TW
N N B O B O T W N N B O B I O O T B S H T F N E B O T F N E T F N S B O

Figure 3.2: Block tree for the example string S introduced by Belazzougui et al. [7] with $s = 15$, $\tau = 2$ and a last level leaf size of 1. Nodes/Blocks are represented as boxes, solid edges are pointers from marked blocks towards their children. Dashed edges are leftwards pointer from unmarked blocks towards the pair of blocks containing its leftmost occurrence, which is highlighted by a thick line above the pair. The process of an exemplary access query on $S[46]$ is illustrated by decorating the T symbols visited during the query with a $\diamond$. Note that, the third block *NNBO* in the first level is marked although *NNBO* already appears in S. This due to that both ***TWNN*** in $B_1 \cdot B_2$ and ***BOBI*** in $B_2 \cdot B_3$ do not have an earlier occurrence.

Belazzougui et al. [7] relate the size of the block tree to the Lempel-Ziv factorization of S by proofing the following properties.

Lemma 3.3.1. *The number of blocks in any level of the block tree (except the first) is at most* $3z\tau$.

Proof. To show this property, Belazzougui et al. [7] argued that for any level ℓ but the first, a concatenation of three consecutive blocks $B_{i-1} \cdot B_i \cdot B_{i+1}$ in level $\ell - 1$, that does not contain any Lempel-Ziv phrase boundary is part of a substring that occurs at least twice rightwards in S (see Section 3.1). Thus, $B_{i-1} \cdot B_i \cdot B_{i+1}$ has a leftwards occurrence in S and none of $B_{i-1} \cdot B_i$ and $B_i \cdot B_{i+1}$ will be marked. Therefore, B_i is unmarked. In conclusion, we only mark $B_{i-1} \cdot B_i \cdot B_{i+1}$, whenever a phrase boundary falls into B_i. Since the Lempel-Ziv factorization has z phrases, there are at most z such blocks. Hence, we mark at most $3z$ blocks in level $\ell - 1$. Each marked block creates τ children in ℓ. Thus, level ℓ has at most $3z\tau$ blocks. □

Theorem 3.3.2. *Given a string S of length n over on alphabet of size σ and integers $s, \tau > 1$, a block tree of S requires $\mathcal{O}((s + z\tau \log_\tau \frac{n \log \sigma}{s \log n}) \log n)$ bits of space, where z is the number of phrases in the Lempel-Ziv factorization of S.*

Proof. With Lemma 3.3.1, there are $\mathcal{O}(z\tau)$ blocks per level. All unmarked blocks (leaves not on the last level) require $\mathcal{O}(\log n)$ bits of space per block for the leftwards pointer. Marked blocks require $\mathcal{O}(\tau \log n)$ for the τ pointers towards their children, but we can charge that to the children itself. The last level contains $\mathcal{O}(z\tau)$ blocks each storing explicit text that is, $\log_\sigma n$ symbols of $\log \sigma$ each, totaling $\mathcal{O}(\log n)$ bits per block. Hence, we need $\mathcal{O}(s)$ pointers to the top-level blocks and $\mathcal{O}(z\tau)$ pointers in each of the $\log_\tau \frac{n \log \sigma}{s \log n}$ other levels. Note that, if $z\tau > n$ the block tree uses $\Omega(n \log n)$ bits, and it is better to store S in plain text as we know it will not compress. □

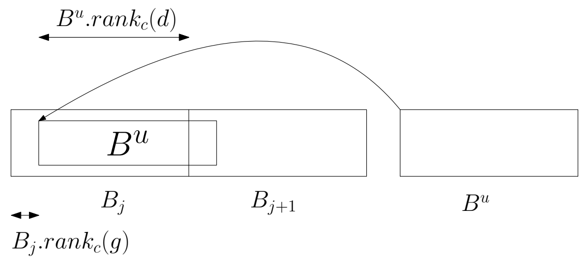


Figure 3.3: To convert a rank query on an unmarked block B^u into a rank query on one of the consecutive marked blocks B_j and B_{j+1} that contain the first occurrence of B^u in the string S at $g+1$, we can store $B_i.rank_c(g)$ in B^u and use the pre-computed samples $pre(c)$ to infer $B^u.rank_c(d)$. Visualization taken from Belazzougui et al. [7].

We retain the asymptotic space complexity for $s = z$, where the block tree is of height $\log_\tau \frac{n \log \sigma}{z \log n}$, leading to $\mathcal{O}(z\tau \log_\tau (\frac{n \log \sigma}{z \log n}) \log n)$ bits of space. Furthermore, using a constant value for τ yields minimum space $\mathcal{O}(z \log (\frac{n \log \sigma}{z \log n})) \subseteq \mathcal{O}(z \log (n/z))$ (measured in $\Theta(\log n)$-bit words) and a logarithmic number of levels $\mathcal{O}(\log (\frac{n \log \sigma}{z \log n})) \subseteq \mathcal{O}(\log (n/z))$ [7].

3.3.1 Queries

Belazzougui et al. [7] introduced the following efficient access, rank and select queries. Block trees can answer access queries in $\mathcal{O}(\log_\tau \frac{n \log \sigma}{s \log n})$ time. To access $S[i]$, we traverse from root to bottom level, by descending into the child containing $S[i]$. If we reach a leaf B^u that does not store its block explicitly, we follow the leftwards pointer towards $B_j \cdot B_{j+1}$, while considering the offset of $S[i]$ in B^u. In $\mathcal{O}(1)$ we can identify a character in $S[i']$ in $B_j \cdot B_{j+1}$ with $S[i'] = S[i]$. We then continue to descend into the child containing $S[i']$. When we reach a leaf that stores its contents explicitly, we can return $S[i]$ immediately. As we can traverse leftwards pointers only once per level, we return $S[i]$ in $\mathcal{O}(\log_\tau \frac{n \log \sigma}{s \log n})$ time, proportional to the height of the block tree.

To support efficient rank queries, it is necessary to store additional information. We store, for each unique character c, the number $B^v.pre(c)$ of occurrence of c in the prefix of S preceding each block B^v. This requires $\mathcal{O}(\sigma(s + z\tau \log_\tau (\frac{n \log \sigma}{s \log n})) \log n)$ bits of space. These samples allow us to translate a rank query $S.rank_c(i)$ into a rank query on a corresponding block. If $S[i]$ is contained at position j in B^u, then $S.rank_c(i) = B^v.pre(c) + B^v.rank_c(j)$. Therefore, we can turn a rank query on a marked block into a rank query on one of its children in $\mathcal{O}(1)$ time. Similar to access queries, we can translate a query on an unmarked B^u, with a leftwards pointer towards $B_j \cdot B_{j+1}$ and B^u starting at position $g+1$ in B_j, into a rank query on one of B_j and B_{j+1}. To implement efficient rank queries, it is necessary to store for each unmarked block B^u and each unique character c $B_j.rank_c(g)$, the number of occurrences of c in the prefix of B_j preceding the occurrence of B^u. As can be seen in Figure 3.3 [7].

Furthermore, let $d = |B_j| - g$ be the length of the prefix of B^u overlapping with B_j. As the number of cs in B_j is $B_{j+1}.pre(c) - B_j.pre(c)$, the number of cs in $B^u[0..d) = B_j[g+1..]$ is $B^u.rank_c(d) = B_{j+1}.pre(c) - B_j.pre(c) - B_{j+1}.rank_c(g)$. If we transform into a query on B_j, $i \leq d$ holds and $B^u.rank_c(i) = B^u.rank_c(d) + B_{j+1}.rank_c(i-d)$, otherwise we transform into a query on B_{j+1} and $B^u.rank_c(i) = B_j.rank_c(g+i) - B_j.rank_c(g)$. Finally, a binary rank data structure [11, 41], for the concatenation of the strings stored explicitly in leaves, answers rank queries on these leaves in $\mathcal{O}(1)$ time and $\mathcal{O}(z\tau \log_\sigma n)$ bits space. Hence, rank queries on S take $\mathcal{O}(\log_\tau \frac{n \log \sigma}{s \log n})$ time [7].

Efficient select queries require additional predecessor data structures on the rank samples $B^v.pre(c)$. We add predecessor data structures for all s top-level blocks, which yield a top-level block B^v. This lets us translate the select query $S.select_c(j)$ onto B^v with $S.select_c(j) = i - 1 + B^v.select_c(j - B^v.pre(c)$. Similar predecessor structures are stored for the ranks sample $\{B^{v_i}.pre(c) \mid i \in [0, \tau)\}$ of the τ children of each marked block B^v. These predecessor structures translate select queries on B^v to the correct child in the same way. Turning select queries on an unmarked block B^u into a select query on B_i and B_{i+1} can be done with the information stored for rank queries. If we want to select a position in B^u that overlaps with B_i then $j \leq B^u.rank_c(d)$ holds and $B^u.select_c(j) = B_i.select_c(j + B_i.rank_c(g)) - g$, else we select a position in B^u overlapping with B_{i+1} and then $j > B^u.rank_c(d)$ holds and $B^u.select_c(j) = B_{i+1}.select_c(j - B^u.rank_c(d)) - d$. For the last level a constant time select data structure [11, 42], for the concatenation of explicitly in leaves stored strings, answers select queries on these leaves in $\mathcal{O}(1)$ time and $\mathcal{O}(z\tau \log_\sigma n)$ bits space. Predecessor structure allow for time-space-tradeoffs [6]. For constant τ, predecessor queries on internal nodes take $\mathcal{O}(1)$ time, therefore for the structure uses $\mathcal{O}(\sigma z \log (n/z))$ space and supports rank and select queries in $\mathcal{O}(\log (n/z))$ time [7].

3.3.2 Construction

The core construction strategy proposed by Belazzougui et al. [7] is shown in Algorithm 5. The first step is to partition S into s blocks of length n/s. Then, with a single pass over S, all blocks are marked. Next, with a second pass, the leftwards pointers for unmarked blocks are set. Afterwards, the next level is created by generating τ children for each marked blocks. The successive passes for each further level decreases the block length by a factor of τ. This reduces the number of string positions still contained inside of blocks, geometrically, with each level. We terminate when the maximum string leaf size m is reached, and we store the last level blocks explicitly. The text necessary to process in each level geometrically decreases because per Lemma 3.3.1, there are at most $3z\tau$ blocks per level and the block size decreases by a factor of τ each level. Finally, Belazzougui et al. [7] proposed a post-processing space optimization, where unnecessarily marked blocks are removed.

Belazzougui et al. [7] proposed both a worst-case-time and an expected time construction strategies that achieve $\mathcal{O}(n)$ construction time when $s = \Theta(z)$.

Worst-case-time construction using $\mathcal{O}(n)$ working space. We follow Algorithm 5 and first partition S into s blocks $B_0, ..., B_{s-1}$ of length n/s.

Identifying marked blocks is split into three parts. First, we create in $\mathcal{O}(n)$ time and space an Aho-Corasick automaton [1] that recognizes all $s-1$ consecutive pairs $B_0 \cdot B_1, B_1 \cdot B_2 ... B_{s-2} \cdot B_{s-1}$ in the first level. Aho-Corasick automata allow us to search for all pairs at the same time in $\mathcal{O}(1)$ amortized time per scanned symbol [1]. Next, we initialize a counter for each block B_i to zero. Afterwards, we traverse S with the

Algorithm 5: Block Tree Construction Scheme [7]

```
Data: String: $S[0..n-1]$, inital level size: $s$, maximum string leaf size: $m$, arity: $\tau$
Require: $n = s \cdot \tau^h$ for some integer $h$, $m = \log_\sigma n$
1 $\ell \leftarrow n/s$;
2 $L := \{B_0, B_1 ... B_{s-1}\} =$ partition($S, s$);
3 while $\ell > m$ do
4     markLevel($L, S, \ell$);
5     setPointers($L$);
6     $\ell \leftarrow \ell/\tau$;
7     generateNextLevel($L, S$);
8 storeStringLeaves($L, S$);
```

automaton. If the automaton recognizes a pair $B_i \cdot B_{i+1}$ for the first time, we increment the counter for both B_i and B_{i+1} by one. Finally, we iterate over $B_0, ..., B_{s-1}$ once more and all blocks B_i with a counter equal to 2 are unmarked, because both $B_{i-1} \cdot B_i$ and $B_i \cdot B_{i+1}$ have an earlier occurrence in S. In addition, we mark B_{s-1} if the associated counter is set to one.

To set the leftwards pointer for unmarked blocks, we first destroy the first Aho-Corasick automaton and create a new one that recognizes all unmarked blocks B^u. Next we traverse S with the automaton again. If the automaton recognizes an unmarked block in a block B_i for the first time, set the leftwards pointer for B^u towards B_i. In addition, we store in B^u the offset g where the occurrence of B^u starts within B_i. If $g > 0$ we also set a leftwards pointer for B^u towards B_{i+1}. Because B_i and B_{i+1} contain the first occurrence of the string B^u, they must also be the leftmost occurrence of $B_i \cdot B_{i+1}$. Therefore, B_i and B_{i+1} are both marked. Next, we delete the second Aho-Corasick automaton again and move on to the second level.

Each marked block is split into τ children, all of size $n/(s\tau)$ and repeat the process until we reach the last level, with blocks of length $\log_\sigma(n)$, where we store the text corresponding to each block explicitly. In the following levels, we build the automaton only considering the existing blocks in the level. Blocks B_i that are surrounded by two blocks and therefore form two consecutive pairs $B_{i-1} \cdot B_i$ and $B_i \cdot B_{i+1}$, need to reach a counter of 2 to be unmarked. Other blocks only require a counter value of one to be unmarked.

Theorem 3.3.3. *Given any string $S[0..n-1]$ of a constant alphabet of size σ, the time to build a block tree with integer parameters s and τ is $\mathcal{O}(n(1 + \log_\tau(z/s)))$.*

Proof. To analyze the construction time complexity, Belazzougui et al. [7] argued the following. The top-level blocks are of length n/s. If $s < 3z\tau$, then after level $\ell = 1 + \lceil log_\tau 3z/s \rceil$ the block lengths are below $n/(3z\tau)$. Up to level ℓ, the lengths of the existing blocks may add up to n per level, but after level ℓ, since there exist at most $3z\tau$ blocks per level (Lemma 3.3.1) and their lengths decrease exponentially, the sum of all the existing block lengths is $\mathcal{O}(n)$. Therefore, the total sum of all block lengths is $\mathcal{O}(n(1 + \log \tau(z/s)))$. Because each symbol is processed in constant amortized time, the total process requires $\mathcal{O}(n(1+\log \tau(z/s)))$ time and $\mathcal{O}(n)$ working space. In addition, we must take the time required to generate the block tree into account. This adds up to $\mathcal{O}(s + z\tau \log_\tau \frac{n \log \sigma}{s \log n}) \subseteq \mathcal{O}(n + z\tau \log_\tau (n/s))$, being $\mathcal{O}(n(1 + \log_\tau (z\tau/s)))$ for compressing block trees with $z\tau = \mathcal{O}(n)$ (Theorem 3.3.2). □

With, $s = z$ we can build the block tree in $\mathcal{O}(n)$ by determining z by calculating the Lempel-Ziv factorizaton for S. This take $\mathcal{O}(n)$ time and space [46].

Expected-time construction with $\mathcal{O}(s + z\tau)$ working space. The Aho-Corasick automata require $\mathcal{O}(n)$ working space, which may be too much on very long strings. Replacing the Aho-Corasick automata by searching with Rabin-Karp fingerprints [31] reduces the space required to $\mathcal{O}(s + z\tau)$. Instead of building the automata, we store the Rabin-Karp fingerprints of all consecutive pairs of blocks $B_i \cdot B_{i+1}$ in a hash table. Afterwards, we traverse S with a sliding window Rabin-Karp scan with a window size of $|B_i \cdot B_{i+1}|$, computing the fingerprints of all the windows each in constant time. Furthermore, it is necessary to verify the strings with matching fingerprints are actually equal, but with high probability (that is, with probability $1 - \mathcal{O}(n^{-c})$ for any constant c) the strings match and a matching block is never verified again, because we have already found its leftmost occurrence. Similarly, in the next step, we find the leftmost occurrences for unmarked blocks using the Rabin-Karp fingerprints of all unmarked blocks. Thus, we spend $\mathcal{O}(n)$ time scanning and $\mathcal{O}(n)$ expected time for hashing and verification at each level. When using the right hashing scheme, the expected time becomes $\mathcal{O}(n)$ w.h.p [52].

In conclusion, the working space per level is now proportional to the number of blocks in that level. Therefore, building the block tree requires $\mathcal{O}(s + z\tau)$.

Pruning. The structure we built meets the space requirements outlined in Lemma 3.3.1, but it may contain more blocks than necessary. We mark the first instance of each pair of consecutive blocks $B_i \cdot B_{i+1}$ to ensure that any block B^u to their right in S within them can point to them without having unmarked blocks pointing to unmarked blocks. However, it is possible that no rightward block B^u points to these blocks and that either or both of the blocks $B_i \cdot B_{i+1}$ appear earlier in the sequence S. In this case, we could replace them with leftward pointers, which would reduce the space needed. For example, the last top-level block *BOTW* in Figure 3.2 meets the criteria and should be able to be set as an unmarked block and have its children removed, with a leftward pointer set towards the second top-level block instead.

To apply the pruning step, we make a modification to our construction and also perform a post-processing space optimization on the block tree once it has been built. Specifically, when finding leftwards pointers, we include all blocks at the level, not just the unmarked ones. We also record the leftmost occurrence of marked blocks found, even though we have not yet replaced them. During post-processing, we traverse the block tree in post-order, starting with the children in reverse order and ending with the parent. All nodes have a counter value of zero at the beginning of the traversal. If we encounter an unmarked block B^u that points leftward to blocks $B_i \cdot B_{i+1}$, we increment the counters of $B_i \cdot B_{i+1}$. If we reach a marked block with a zero counter and its children are either unmarked or last-level leaves storing explicit strings, and its leftmost occurrence in S does not overlap itself, we remove its children and make the block unmarked, creating a leftward pointer. The blocks pointed to by the removed children must then have their counter incremented, and the counters of blocks pointed to by the removed children must be decremented. The post-processing time is proportional to the block tree size $\mathcal{O}(s + z\tau \log_\tau \frac{n \log \sigma}{s \log n})$ and therefore does not increase the asymptotic construction time.

To prune the last top-level block *BOTW* in Figure 3.2 we begin by initializing all counters to zero and start with the last child of the root *BOTW*. We descend into its children

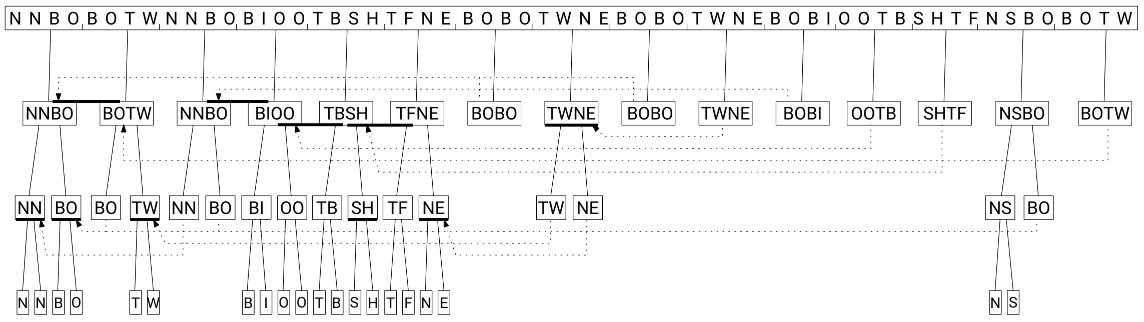


Figure 3.4: Pruned block tree for the block tree in Figure 3.2

TW and *BO* in reverse order, as both children are leaves, we mark the blocks pointed to by them. Afterwards we ascend back to *BOTW*. Here we notice that its counter is zero, both of its children are unmarked, and the leftmost occurrence of *BOTW* is not overlapping. Therefore, we can prune *BOTW*. We remove its children, decrease the pointers of the blocks we just incremented again, unmark *BOTW* and set a leftwards pointer towards the second block. Finally, we increase the counter of the second block to ensure that we later do not prune the second block. The pruned block tree of Figure 3.2 is shown in Figure 3.4.

4 Concept

Recall that Belazzougui et al. [7] presented a level wise construction algorithm, that in its core first identified all marked blocks and then identified the leftmost occurrence for each unmarked blocks in a second pass. In this chapter, we first describe variants to identify marked and unmarked blocks for each level. Then we present an approach to calculate the leftmost occurrence for each block, and lastly we combine these approaches to a complete block tree construction algorithm. For our approach to construct the block tree for a string, S we rely on two well-researched data structures, the `LPF`-Array of S and the corresponding `PrevOcc`-Array (see Section 3.2).

4.1 Marking Blocks

Recall that we mark two consecutive blocks B_i, B_{i+1} if $B_i \cdot B_{i+1}$ contains the leftmost occurrence of any substring of S (Definition 2). Belazzougui et al. [7] showed that there exists a link between marked blocks and `LZ77`-phrases, by proofing that there are at most $3z$ marked blocks in each level and three consecutive blocks B_{i-1}, B_i and B_{i+1} are only marked if a `LZ77`-phrase boundary falls into B_i.

4.1.1 Marking Blocks Using `LZ77`-phrases

The `LPF`-Array allows us to quickly calculate the `LZ77`-phrases of S, which leads to a

Algorithm 6: Mark blocks using `LZ77`-phrase boundaries

```
Data: LZ77-phrase boundaries of String $S$: P, blocks in level: $B_0, B_1, ...B_{s-1}$,
      block length: $\ell$.
Result: Marking for given block tree level
1 $j \leftarrow 0$;
2 for $i \leftarrow 0$ to $|P|$ do
3     while $j < s - 1$ and start($B_{j+1}$) $\leq P_i$ do
4         $j \leftarrow j + 1$;
5     mark($B_j$);
6     if consecutive($B_{j-1}$, $B_j$) then
7         mark($B_{j-1}$);
8     if consecutive($B_j$, $B_{j+1}$) then
9         mark($B_{j+1}$);
```

naive approach to calculate a set of markings that achieves the same theoretical space upper bound as the original block tree definition. Recall Lemma 3.3.1, stating that we only mark the consecutive blocks B_{i-1}, B_i and B_{i+1}, if a LZ77-phrase boundary falls into the sub-string of S represented by B_i. We can now simply calculate the LZ77-phrases boundaries once and apply this rule on all levels. We still only mark at most $3z$ blocks per level, but we might mark more blocks than the marking algorithm presented by Belazzougui et al. [7].

The algorithm to calculate the markings is shown in Algorithm 6. We assume that we have stored the LZ77-phrase boundaries ordered by their position in S and also stored a representation of the current block tree level ordered by the starting text position in S. In an outer loop (lines $2-9$) we iterate over all phrase boundaries and identify all blocks that contain a phrase boundary. During each outer loop iteration, we first search for the block B_j whose content contains the next phrase boundary (lines $3-4$). After that comes the actual marking process (lines $5-9$). We mark B_j and mark B_{j-1} if $B_{j-1} \cdot B_j$ are consecutive and B_{j+1} if $B_j \cdot B_{j+1}$ are consecutive. As we mark at most $3z$ blocks per level, each level, but the first, has at most $3\tau z$ blocks. Recall that the first level is split into s blocks and hence for $s < 3\tau z$ all levels have at most $3\tau z$ blocks.

Lemma 4.1.1. *Algorithm 6 calculates a valid marking in $\mathcal{O}(z\tau)$ time.*

Proof. Although we now have two nested loops iterating both from 0 to z or up to $3\tau z$ respectively, we increase each respective loop variable during each loop execution and never decrease them. Therefore, we execute at most $(3\tau+1)z$ loop iterations, with $\mathcal{O}(1)$ work per iteration. □

4.1.2 Marking Blocks With LPF-Array

Remember that for S, $\texttt{LPF}[i]$ stores the longest previous factor of index i in S, in other words the longest sub-string $S[i..i+x)$ that already occurred in S starting before i has length $x := \texttt{LPF}[i]$ and especially all factors $S[i..i+\ell)$ with $\ell > x$ have not occurred in S before position i.

Algorithm 7: Mark blocks using LPF-Array

```
Data: LPF-Array of String $S$: LPF$[0..n-1]$, blocks in level: $B_0, B_1, ...B_{s-1}$, block
      length: $\ell$.
Result: Marking for given block tree level
1 mark($B_0$);
2 for $i \leftarrow 1$ to $s-2$ do
3     if consecutive($B_{i-1}$, $B_i$, $B_{i+1}$) and (LPF[start($B_{i-1}$)] $< 2\cdot\ell$ or
        LPF[start($B_i$)] $< 2\cdot\ell$) then
4         mark($B_i$);
5 if $s > 0$ and consecutive($B_{s-2}$, $B_{s-1}$) and LPF[start($B_{s-2}$)] $< 2\cdot\ell$ then
6     mark($B_{s-1}$);
```

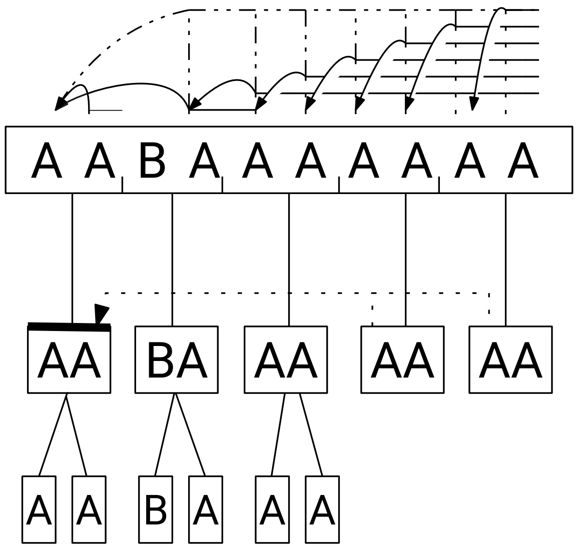
Lemma 4.1.2. *(Marking with LPF-Array) Given the LPF-array we can mark a block tree level faithful to the theoretical proposal in $\mathcal{O}(z\tau)$ time. Recall that a block B_u is unmarked exactly when the sub-strings in S related to both $B_{u-1} \cdot B_u$ and $B_u \cdot B_{u+1}$ have an earlier occurrence in S [7].*

Proof. It is possible to express this condition by considering the related LPF values. Assume we have a partition of blocks $B_0, ..., B_{s-1}$ representing the first level for a block tree of $S[0..n)$ with a block length of $\ell := n/s$ and each block B_i representing the sub-string $S[i \cdot \ell..i \cdot (\ell + 1))$ (short S_{B_i}). In addition, we have calculated the LPF array for S. The algorithm to decide which blocks are marked is outlined in Algorithm 7. We iterate once over all blocks but the first and last block (lines $2-4$). A block B_i is marked exactly when the sub-strings represented by $B_{i-1} \cdot B_i$ and $B_i \cdot B_{i+1}$ are consecutive in S (this is always true in the first level) and at least one of the LPF values for $(i-1) \cdot l$ and $i \cdot l$ is smaller than the combined length of the consecutive sub-strings $S_{B_{i-1}} \cdot S_{B_i}$ or $S_{B_i} \cdot S_{B_{i+1}}$ (or the doubled length of S_{B_i} as all sub-strings have the same length). This holds true as LPF values smaller than $2 \cdot \ell$ implies that there is no former occurrences of either $S_{B_{i-1}} \cdot S_{B_i}$ or $S_{B_i} \cdot S_{B_{i+1}}$ in S and therefore it has to be marked (line 3). We always mark the first block as its content can not occur previously in S (the first block of each level B_0 always represents the sub-string $S[0..\ell)$). Finally, we mark the last block B_{s-1} if the second to last block B_{s-2} and B_{s-1} are consecutive and the LPF$[(s-2) \cdot \ell]$ is smaller than $2 \cdot \ell$ (line 5 and 6). As we have $\mathcal{O}(1)$ work during each loop execution, the total running time for each marking pass is $\mathcal{O}(z\tau)$. □

We can now proceed to determine the first occurrence of each unmarked block and repeat the process for the next level. Note that in all levels, but the first one, a block's B_i index i doesn't necessarily translate to the related sub-string's S_{B_i} starting position. We don't know how many blocks have been unmarked to its parents left in the previous level and don't spawn child blocks in the next level. Therefore, We need to store additional information regarding a block's contents starting position in S. For $S :=$ *AABAAAAAAA* Figure 4.1 shows the block tree and the LPF-Array of S. Let us take the third child of the root B_2 and consider why it is marked and creates 2 children in the next level. First, all blocks on the first level are consecutive, so the condition regarding B_1, B_2 and B_3 being consecutive is met. Next, we take a look at the LPF-values of the starting positions of B_1 and B_2. LPF[2] is 0 and LPF[4] is 6, with LPF$[2] = 0 < 4$ being smaller than the pair size 4. Therefore, B_2 has to be marked, as the pair $B_1 \cdot B_2$ contains the leftmost occurrence of at least one sub-string in S, for example the leftmost occurrence of *B* or *BA*. The next block B_3 on the other hand, although it represents the same text *AA*, can be unmarked as both LPF$[4] = 6 \geq 4$ and LPF$[6] = 4 \geq 4$ are not smaller than the pair size 4. Consequently, all sub-strings contained in $B_2 \cdot B_3 = B_3 \cdot B_4 :=$ *AAAA* have an earlier occurrence, not contained by $B_2 \cdot B_3$ in S. Note that the leftmost occurrence of *AAAA* overlaps into $B_2 \cdot B_3$, but is not contained by, it.

4.2 Identifying Leftmost Occurrences Using LPF- and PrevOcc-Array

After identifying which blocks in a given block tree level are unmarked, it is necessary to find the leftmost occurrences of the content of each unmarked block in S. In the following section, we first present the general idea behind our identification approach and then add modifications, which allows us to give run-time guarantees. On a higher level, we will first find the leftmost occurrences for each unmarked block as a text position in S, then we will link these text positions to the marked block in the current block tree level.



i	$S[i]$	$S[i..]$	LPF$[i]$	PrevOcc$[i]$	FirstOcc$_2[i]$
0	A	AABAAAAAAA	0	-1	-1
1	A	ABAAAAAAA	1	0	0
2	B	BAAAAAAA	0	-1	-1
3	A	AAAAAAA	2	0	0
4	A	AAAAAA	6	3	0
5	A	AAAAA	5	4	0
6	A	AAAA	4	5	0
7	A	AAA	3	6	0
8	A	AA	2	7	0
9	A	A	1	8	0

Figure 4.1: LPF, PrevOcc and FirstOcc$_2$ and the block tree for the string *AABAAAAAAA*, with $s = 5, \tau = 2$ and leaves of size 1. LPF-values are drawn as solid lines above text, pointers to previous occurrence are drawn as a solid edge and pointers in FirstOcc$_2$ are drawn as dashed edges above the text.

Leftmost Occurrence as Text Position

Recall that for S, $\texttt{LPF}[i]$ stores the longest previous factor of index i in S and the corresponding PrevOcc-Array stores a starting position of the longest previous factor in S. We have the following observations on S, LPF and PrevOcc. Consider positions i, j in S with $j := \texttt{PrevOcc}[i]$. If $\texttt{LPF}[j]$ is at least $\texttt{LPF}[i]$, then the longest previous factor of i is a prefix of the longest previous factor of j and therefore said prefix is a previous occurrence of the longest previous factor of i, which is also to the left of the initial previous occurrence pointed to in PrevOcc[i].

Lemma 4.2.1. *(transitive previous occurrence) Let* $i, p := \texttt{PrevOcc}[i]$ *be positions in* S*, if* $0 < \texttt{LPF}[i] \leq \texttt{LPF}[p]$*. Then:* $S[i..i+\texttt{LPF}[i]) = S[p..p+\texttt{LPF}[i]) = S[f..f+\texttt{LPF}[i])$ *with* $f := \texttt{PrevOcc}[p]$*.*

Proof. $\texttt{LPF}[i]$ is defined as $max\{k|S[i..i+k)$ occurs at a position $j < i\}$ and PrevOcc gives a previous position j. Hence, from $0 < \texttt{LPF}[i]$: $S[i..i+\texttt{LPF}[i]) = S[p..p+\texttt{LPF}[i])$ and from $0 < \texttt{LPF}[p]$: $S[p..p+\texttt{LPF}[p]) = S[f..f+\texttt{LPF}[p])$ with $\texttt{LPF}[i] \leq \texttt{LPF}[p]$ follows especially $S[p..p+\texttt{LPF}[i]) = S[f..f+\texttt{LPF}[i])$ and with $S[i..i+\texttt{LPF}[i]) = S[p..p+\texttt{LPF}[i])$ follows $S[i..i+\texttt{LPF}[i]) = S[p..p+\texttt{LPF}[i]) = S[f..f+\texttt{LPF}[i])$. □

Consider the example string *AABAAAAAAA* shown in Figure 4.1. Above the block tree, we visualized the lpf-values and the previous occurrences by drawing a solid black line above the longest previous factor for each text positon and a solid black edge pointing to the previous occurrence. For the indices $7, 8$ and 9 we have $\texttt{LPF}[7] = 3, \texttt{LPF}[8] = 2$ and $\texttt{LPF}[9] = 1$ and $\texttt{PrevOcc}[7] = 6, \texttt{PrevOcc}[8] = 8$ and $\texttt{PrevOcc}[9] = 8$. We can now follow the solid edges and find out that a previous occurrence of $S[9..]$ has also a previous occurrence starting at 7.

Given a fixed factor length ℓ, in our case the current block length, we can conclude that if ℓ is smaller or equal to $\texttt{LPF}[i]$ the factor $S[i..i+\ell)$ also occurs at position PrevOcc[i].

Lemma 4.2.2. *(prefixes of longest previous factor) Let* $i, p := \texttt{PrevOcc}[i]$ *be positions in* S*. If* $0 < \ell \leq \texttt{LPF}[i]$ *then,* $S[i..i+\ell) = S[p..p+\ell)$*.*

Proof. $\texttt{LPF}[i]$ is defined as $max\{k|S[i..i+k)$ occurs at a position $j < i\}$ and PrevOcc gives a previous position j.Therefore, follows from $0 < \texttt{LPF}[i]$: $S[i..i+\texttt{LPF}[i]) = S[p..p+\texttt{LPF}[i])$ with $\ell \leq \texttt{LPF}[i]$ follows $S[i..i+\ell]) = S[p..p+\ell)$. □

The naive algorithm to find the leftmost occurrence of a given unmarked block B^u with content $S^u := S[i..i+\ell)$ is shown in Algorithm 8. At first, we select the previous occurrence pointed to in $\texttt{PrevOcc}[i] := p$. If the value stored at $\texttt{LPF}[p]$ is also larger than ℓ, we can conclude with Lemma 4.2.2 that S^u has at least one more occurrence to the left of position p as the factors $S[p..p+l)$ and $S[f..f+l)$ with $f := \texttt{PrevOcc}[p]$ are both equal to S^u. We set $p := f$ and repeat this until $\texttt{LPF}[p]$ is smaller ℓ and therefore the longest previous factor for p doesn't have $S[i..i+\ell)$ as prefix. In other words, we follow a chain of longest previous factors until the next longest previous factor is smaller than ℓ which means that S^u is not a prefix of it, and we reached the leftmost occurrence of S^u.

Algorithm 8: Naive scan to find the left most occurrence of $S[i..i+l)$

```
Data: LPF-Array of String $S$: LPF$[0..n-1]$, PrevOcc-Array of LPF:
      PrevOcc$[0..n-1]$, block length: $\ell$, position: $i$.
Result: Leftmost occurrence of factor $S[i..i+l]$: $p$.
$p \leftarrow i$ ;
while LPF$[p] \geq l$ do
    $p \leftarrow$ PrevOcc$[p]$;
```

Consider now the example string *AABAAAAAAA* and its LPF- and PrevOcc-Array shown in Figure 4.1. In Section 4.1.1 we argued that the fourth child B_3 (content of B_3 starts at position 6 in S) in the first level has to be unmarked. It is easy to see, that its content *AA* first appears right at the beginning of S as $S[0..1]$. LPF$[6] = 4$ and PrevOcc$[6] = 5$ as the longest previous factor of $S[6..]$ is *AAAA* located at $S[5..8]$. LPF$[5] = 5$ and PrevOcc$[5] = 4$ as *AAAAA* is the longest previous factor of $S[5..]$ at, $S[4..8]$ which also has a prefix of length $4 =$ LPF$[6]$ that is another longest previous factor of our initial position 6 but located to the left of the longest previous factor referenced in PrevOcc$[6]$. We can now repeat that process as long as we find another longest previous factor. In our example, this leads to the following chain: $6 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 0$.

This approach may lead to numerous loop iterations in Algorithm 9. As a worst case, consider the "all-a-text" *aaa...a* of size n and the related LPF and PrevOcc arrays. Here for all but the first string position LPF$[i]$ will store $n-i$. Furthermore, with canonical LPF construction algorithms (see Section 3.2) PrevOcc$[i]$ will point to the nearest previous occurrence. For an "all-a-text" this is $i-1$. Hence, a chain would follow along every previous occurrence $i-1, i-2, i-3...0$ until it terminates at index 0, the leftmost occurrence for factors in an "all-a-text".

We now propose an approach to precompute all necessary information to find the leftmost occurrence for each unmarked block in a given block tree level.

Definition 3. *For $\ell > 0$ we use **ℓ-factor$_i$** to denote the prefix $S[i..i+l)$ of $S[i..]$*

Definition 4. *We use* FirstOcc$_\ell$ *to denote the **first occurrence array**.* FirstOcc$_\ell$ *is an adaption of **previous occurrence array** where,* FirstOcc$_\ell[i]$ *stores the leftmost starting location of either **ℓ-factor$_i$** for $S[i..]$ or an occurrence of the **longest previous factor** of i, when $0 <$* LPF[i] *$< \ell$ or -1 if* LPF$[i] = 0$.

The Algorithm to compute FirstOcc$_\ell$ with LPF and PrevOcc is shown in Algorithm 9 and is based on dynamic programming. For string positions $i, p :=$ PrevOcc$[i]$ we consider the following to two cases when computing the leftmost occurrence of ℓ-factor$_i$:

- Case 1 (Lines $3-4$): LPF$[i] \geq \ell$ and LPF$[p] \geq \ell$. From LPF$[p] \geq \ell$ follows that ℓ-factor$_p$ has an earlier occurrence and with Lemma 4.2.2 and LPF$[i] \geq \ell$, we can conclude that the previous occurrence of ℓ-factor$_p$ is also an occurrence of ℓ-factor$_i$. As we already calculated FirstOcc$_\ell$[p] during the p-th loop iteration, we now set FirstOcc$_\ell[i]$ to FirstOcc$_\ell[p]$.
- Case 2 (Lines $5-6$): LPF$[i] < \ell$ or LPF$[p] < \ell$. In this case, FirstOcc$_\ell[i]$ is set to p. If LPF$[i] = 0$ no factor of $S[i..]$ has a previous occurrence, and therefore we set

$\texttt{FirstOcc}_\ell[i]$ to $p = -1$. For $0 < \texttt{LPF}[i] < \ell$ we set $\texttt{FirstOcc}_\ell[i]$ to p as it points to a previous occurrence of the longest previous factor of i. Note that we still need the LPF-array to interpret $\texttt{FirstOcc}_\ell[i]$. For $\texttt{LPF}[i] \geq \ell$ but $\texttt{LPF}[p] < \ell$ we conclude that ℓ-factor$_i$ has an earlier occurrence at p but no occurrence further left. Hence, we set $\texttt{FirstOcc}_\ell[i]$ to p.

Note that, we can't use dynamic programming for a more general FirstOcc for the longest previous factors, because for a series of string positions $i, p := \texttt{PrevOcc}[i]$ and $f := \texttt{PrevOcc}[p]$, $\texttt{LPF}[i] \leq \texttt{LPF}[p]$ and $\texttt{LPF}[p] > \texttt{LPF}[f]$ doesn't imply $\texttt{LPF}[i] < \texttt{LPF}[f]$. Therefore, the longest previous factor of i can still have an occurrence to the left of p. In other words, $S[p..]$ and $S[f..]$ might have a common prefix equal to the longest previous factor of i. Even if the longest previous factor of p is not a prefix of $S[f..]$.

Algorithm 9: Compute $\texttt{FirstOcc}_\ell$

```
Data: LPF-Array of String $S$: $\texttt{LPF}[0..n-1]$, PrevOcc for LPF: $\texttt{PrevOcc}[0..n-1]$,
      current block length: $\ell$.
Result: first occurrence array: $\texttt{FirstOcc}_\ell[0..n-1]$.
1 for $i \leftarrow 0$ to $n-1$ do
2     $p \leftarrow \texttt{PrevOcc}[i]$;
3     if $\texttt{LPF}[i] \geq \ell$ and $\texttt{LPF}[p] \geq \ell$ then
4         $\texttt{FirstOcc}_\ell[\text{i}] \leftarrow \texttt{FirstOcc}_\ell[\text{p}]$;
5     else
6         $\texttt{FirstOcc}_\ell[i] \leftarrow p$;
```

Furthermore, it is possible to calculate $\texttt{FirstOcc}_{\ell_0}$ with $\texttt{FirstOcc}_{\ell_1}$, for length $\ell_1 \geq \ell_0$ as input instead of PrevOcc. Every occurrence of a ℓ_1-factor contains the related ℓ_0-factor as a prefix, but still there might be an occurrence of ℓ_0-factor further left, and the necessary information is stored in $\texttt{FirstOcc}_{\ell_1}$. Recall that we also stored pointers to a previous occurrence of the longest previous factor if the longest previous factor is shorter than ℓ_1. As it takes $\mathcal{O}(n)$ time to calculate, $\texttt{FirstOcc}_\ell$ it is not feasible to calculate it for each block tree level with block length ℓ.

But circling back to our problem of block tree construction, we can now use this property to identify the leftmost occurrence for each block in a level wise approach. Recall that by definition each pair of marked blocks contains the leftmost occurrence of at least one sub-string of S and therefore also each leftmost occurrence of any sub-string with length smaller or equal to the current level length ℓ is contained in a marked block. All blocks in the current level (except for the first level), are the children of marked blocks in the level before. Hence, all leftmost occurrence of each sub-string of a length equal to the current block length falls into a block in our current level. The adapted algorithm to calculate the leftmost occurrences for a block level is shown in Algorithm 10. Although the leftmost occurrence of each sub-string falls into a block in our current level, we still update all text positions contained int the previous block tree level. This is necessary because we need to consider cases where the lpf-values are smaller than the last level block size ℓ_1 but bigger than the next level block size ℓ_0 and $\ell_0 \leq \texttt{LPF}[i] < \ell_1$ holds for a string position i. Here $\texttt{FirstOcc}_{\ell_1}[i]$ points to a previous occurrence of the longest previous factor of i, this occurrence can be in an unmarked block as it is not necessary the first occurrence of said longest previous factor. Hence, it is also necessary to update

FirstOcc* for string positions k in S that fall into an unmarked block in the previous level (lines $1-6$). We will update FirstOcc$^*[k]$ if one of two conditions is met (line 5).

- Condition 1: LPF$[k] \geq \ell_0$ and LPF$[p] \geq \ell_0$. With Lemma 4.2.2 we conclude that the $p :=$ FirstOcc$^*[k]$ and k share the same ℓ_0**-factor** Therefore, the first occurrence of ℓ_0-factor$_p$ is also the first occurrence of ℓ_0-factor$_i$, which we already calculated in the $p-th$ iteration.
- Condition 2: $0 <$ LPF$[k] \leq$ LPF$[p]$. If condition 2 is met but condition 1 is not, we can differentiate between 2 cases. In any case, FirstOcc$^*[k]$ stores a previous occurrence of the longest previous factor of k.
    1. LPF$[k] < \ell_0$ but LPF$[p] \geq \ell_0$: With Lemma 4.2.1, Lemma 4.2.2 and condition 2 we can conclude that said longest previous factor is a prefix of ℓ_0-factor$_p$ and hence we can update the previous occurrence to FirstOcc$^*[p]$, which, with LPF$[p] \geq \ell_0$, points to the leftmost occurrence of ℓ_0-factor$_p$ and points therefore into a marked block.
    2. LPF$[k] < \ell_0$ and LPF$[p] < \ell_0$: With Lemma 4.2.1 and condition 2 we can conclude that said longest previous factor has a previous occurrence at FirstOcc$^*[p]$ which still points into a marked block. Consider all previous occurrences of said longest previous factor of $p_0, p_1, ..., p$. FirstOcc$^*[p]$ then points to first occurrence p_0, which is the first occurrence of a sub-string smaller than ℓ_0 and hence occurres in a marked block. Or for a previous occurrence p_p, LPF$[p_p] \geq \ell_0$ and FirstOcc$^*[p]$ points to the leftmost occurrence of ℓ_0-factor$_p$ and therefore also points into a marked block

If neither condition is met, k is either the leftmost occurrence for all factors smaller than ℓ_0 or FirstOcc$^*[k]$ points to a first occurrence of a substring smaller than ℓ_0 and hence points into a marked block. Every level but the first level has at most $3z\tau$ blocks, and for each further level the block length decreases by a factor of τ. This reduces the number of string positions still contained inside of blocks, geometrically, with each level (see Section 4.3.1).

Again, consider the example string *AABAAAAAAA* and its FirstOcc$_2$-array shown in Figure 4.1. We draw the FirstOcc$_2$ values above the block tree as dashed edges. Note that, all suffixes starting with *AA*, now point directly to their first occurrence at 0.

Algorithm 10: Compute FirstOcc*

```
Data: LPF-Array of String $S$: LPF$[0..n-1]$, FirstOcc$^*$ for LPF and $\ell_1$:
      PrevOcc$[0..n-1]$, current level block length: $\ell_0$, previous level block
      length: $\ell_1$ previous level number of blocks: $s$, previous block level :
      $B_0, B_1, ...B_{s-1}$.
Result: updated first occurrence array for $\ell_0$: FirstOcc$^*[0..n-1]$.
1 for $i \leftarrow 0$ to $s-1$ do
2     for $j \leftarrow 0$ to $\ell_1 - 1$ do
3         $k \leftarrow$start$(B_i) + j$;
4         $p \leftarrow$ FirstOcc$^*[k]$;
5         if (LPF$[k] \geq \ell$ and LPF$[p] \geq \ell$) or ($p \neq -1$ and LPF$[k] \leq$ LPF$[p]$) then
6             FirstOcc$^*$[k] $\leftarrow$ FirstOcc$^*$[p];
```

Leftmost Occurrence as Block

After updating `FirstOcc`* to store the leftmost occurrence for each string position in the current block level $B_0, B-1...$, it is still necessary to find the marked blocks covering the leftmost occurrence of each unmarked block B^u.

Lemma 4.2.3. *For compressing block trees with $z\tau = \mathcal{O}(n)$, finding the blocks containing the leftmost occurrences of unmarked blocks can be done with Algorithm 11 in $\mathcal{O}(z\tau)$ time and $\mathcal{O}(z\tau(\log(n) + \log(z\tau))$ space.*

Proof. Our algorithm to find the special pointers for unmarked blocks is shown in Algorithm 11 and works as follows. The algorithm is split into three parts. We store for each B^u a pair containing the left most occurrence of its content and its index in our block level in a set U (lines $1-4$). Then, we sort the set by each pair's first element using radix sort. This leads to a sorted list for all left most occurrences In (line 5). Finally, we traverse both our block level and the ordered set U again, checking for a consecutive pair of Blocks $B_i \cdot B_{i+1}$ if the first undiscovered element in U (occ_j, j) is contained by $B_i \cdot B_{i+1}$ (lines $6-13$). In case it is, we found the pair of marked blocks covering the leftmost occurrence of the related unmarked block B_j and we store the special pointer to the two consecutive blocks $B_i \cdot B_{i+1}$ and the offset of the leftmost occurrence of inside $B_i \cdot B_{i+1}$. The times/space complexities of radix sort are $\mathcal{O}(d(z\tau + b))$ time and $\mathcal{O}(n + b)$ space with d being the number of digits/passes and b being the possible values for a digit. For compressing block trees, the with $z\tau = \mathcal{O}(n)$, we set $b = z\tau$ and $d = \mathcal{O}(1)$ to get a run/space complexity of $\mathcal{O}(z\tau)$. Therefore, all steps can be done in $\mathcal{O}(z\tau)$ time and storing the set needs $\mathcal{O}(z\tau(\log(n) + \log(z\tau))$ space. □

Note that is also possible to determine the mapping between leftmost occurrence and blocks for each unmarked block individually. For the first level, we can infer the block directly by text position. For all other level, it is necessary to traverse the already built block tree. In a sense, this mimics an access query, but we stop when we find a current level block and an access query can be done in $\mathcal{O}(\log_\tau \frac{n \log \sigma}{s \log n})$ time (see Section 3.3.1). This adds one additional factor in the height of the block tree for each unmarked block, and calculating the mapping for all unmarked blocks in a level would take $(z\tau(\log_\tau \frac{n \log \sigma}{s \log n}))$ time.

4.3 Block Tree Construction

Now we can put all these building blocks together to form a block tree construction algorithm for a string S, which is shown in Algorithm 12. We first calculate `LPF`-Array and `PrevOcc`-Array for S. First we initialize `FirstOcc`* for $\ell = n/s$ using Algorithm 9 and then partition S into s blocks of length ℓ (lines $1-3$). Then, we then construct each block tree level (lines $4-9$). First, we identify all marked blocks (Algorithm 7). Then we set the special pointers for all unmarked blocks (Algorithm 11). Finally, we update `FirstOcc`* for the next level and split all marked blocks into children blocks, all of size l/τ. We repeat this process until we reach the final level, where the blocks are small enough to store them explicitly.

Algorithm 11: Map occurrences in text to blocks

```
Data: FirstOcc*-Array, current block length: $\ell$, current number of blocks: $s$,
      current block level : $B_0, B_1, ...B_{s-1}$.
Result: All unmarked blocks in our current block level are annotated with special
        pointers towards their leftmost occurence.
1  $U \leftarrow \varnothing$ ;
2  for $i \leftarrow 0$ to $s-1$ do
3      if unmarked($B_i$) then
4          $U \cup \{(\texttt{FirstOcc}^*[\texttt{start}(B_i)], i)\}$ ;
5  radixSort($U$);
6  $k \leftarrow 0$;
7  for $i \leftarrow 0$ to $s-1$ do
8      $(occ_j, j) \leftarrow U[k]$ ;
9      while $B_i$.contains($(occ_j, j)$) do
10         $B_j.firstPointer \leftarrow B_i$;
11         $B_j.secondPointer \leftarrow B_{i+1}$;
12         $B_j.offset \leftarrow occ_j -$start($B_i$);
13         $k \leftarrow k+1$;
```

4.3.1 Complexity Analysis

Recall Theorem 3.3.3 where Belazzougui et al. [7] showed that for $s = \Theta(z)$ block trees can be constructed in $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space. As our block tree construction algorithm 12 follows the same principal procedure outlined in Algorithm 5 as the algorithms proposed by Belazzougui et al. [7], we will show that it achieves the same time complexity $\mathcal{O}(n)$ but has a worse asymptotic space complexity $\mathcal{O}(n \log n)$.

Theorem 4.3.1. *(Time/Space Complexities) Given any string $S[0..n-1]$ of a constant alphabet of size σ, the time, and space complexities to build a block tree with integer parameters s and τ with the algorithm shown in Algorithm 12 are $\mathcal{O}(n(1+\log_\tau(z/s)))$ and $\mathcal{O}(n \log n)$ bits, respectively.*

Proof. Calculating `LPF` and `PrevOcc` take $\mathcal{O}(n)$ time and $\mathcal{O}(n \log n)$ space. Initializing `FirstOcc`* takes $\mathcal{O}(n)$ time. Recall that all but the first block tree level have at most $3z\tau$ blocks (Lemma 3.3.1) and that Belazzougui et al. [7] argued in Theorem 3.3.3, that with $s < 3z\tau$ the sum of all the existing block lengths after level $1 + \lfloor \log_\tau 3z/s \rfloor$ is $\mathcal{O}(n)$ and therefore the total sum of all block lengths is $\mathcal{O}(n(1+\log \tau(z/s)))$. As we update `FirstOcc`* for each position included in a given block level in constant time, the process requires $\mathcal{O}(n(1+\log \tau(z/s))$ time for all levels. Marking blocks, setting pointers for unmarked blocks and generating the next level can be done in $\mathcal{O}(z\tau)$ for each level. This adds up to $\mathcal{O}(s + z\tau \log_\tau \frac{n \log \sigma}{s \log n}) \subseteq \mathcal{O}(n + z\tau \log_\tau (n/s))$, being $\mathcal{O}(n(1+\log_\tau (z\tau/s))$ for compressing block trees with $z\tau = \mathcal{O}(n)$ (Theorem 3.3.2).

`LPF`, `PrevOcc` and `FirstOcc`* require $\mathcal{O}(n \log n)$ space. For compressing block trees with $z\tau = \mathcal{O}(n)$, the space to store the set during Algorithm 11 is $\mathcal{O}(z\tau(\log(n) + \log(z\tau))$ and therefore subsumed by $\mathcal{O}(n \log n)$. □

With the methods described in Algorithm 1 and Algorithm 3 we can determine z in $\mathcal{O}(n)$ time and therefore build the block tree in $\mathcal{O}(n)$ time.

Algorithm 12: Block tree construction

```
Data: String: $S[0..n-1]$, LPF-Array for $S$: $\texttt{LPF}[0..n-1]$, PrevOcc-Array for $S$:
      $\texttt{PrevOcc}[0..n-1]$, inital level size: $s$, maximum string leaf size: $m$, arity: $\tau$
Require: $n = s \cdot \tau^h$ for some integer $h$, $m = \log_\sigma n$
$\ell \leftarrow n/s$;
FirstOcc* =calculateFirstOcc(LPF, PrevOcc, $\ell$);
$L := \{B_0, B_1 ... B_{s-1}\}$ = partition($S$, $s$);
while $\ell > m$ do
    markLevel($L$, LPF, $\ell$);
    setPointers($L$, FirstOcc*, LPF);
    $\ell \leftarrow \ell/\tau$;
    updateFirstOcc($L$, FirstOcc*, LPF, $\ell$);
    generateNextLevel($L$);
storeStringLeaves($L$);
```

4.4 Pruning

We can modify our construction to carry out the post-processing space optimization described in Section 3.3.2. Like, Belazzougui et al. [7] we will also collect leftmost occurrences for all blocks, not just unmarked ones. This can be achieved by simply removing the if condition, where we only calculate leftmost occurrences for unmarked blocks (line 3), during the collection step (lines 2–4) in Algorithm 11. In addition, we need to ensure that for each marked block that the leftmost occurrence doesn't overlap itself. The actual post-processing remains the same.

4.5 Greedy Heuristic

We present a greedy heuristic, that similar to the previous algorithms constructs a block tree level wise. To identify leftmost occurrences, we utilize the concepts presented in Section 4.2. In our greedy heuristic, we do without marking blocks. Instead, we will create leaves as soon as possible. Blocks that don't have a previous occurrence will be considered as a marked block and generate children in the next level. The pseudocode is shown in Algorithm 13. Similar to Algorithm 12, we begin by partitioning S into s blocks of length $\ell := n/s$ and calculating $\texttt{FirstOcc}_\ell$ to reduce the number of future chain operations. For each level in the block tree for S we iterate over all blocks in reverse order. For all unmarked blocks B^u, we try to find the leftmost occurrence of B^u's content with the idea presented in Algorithm 8 (line 9 - 22). Note that we need to ensure that the leftmost occurrence doesn't overlap with B^u (see Section 4.4). If the leftmost occurrence doesn't overlap with B^u, we look up the containing block B_b. This can be done with Algorithm 11 for all blocks or with binary search on L, which reduces construction space, but increases construction time to $\mathcal{O}(z\tau \log z\tau)$. Different from all previous approaches, we can't assume that the leftmost occurrence is still represented in the current level L and therefore have to check if the leftmost occurrence is contained by $B_b \cdot B_{b+1}$ (line 15). If true, we can mark B_b, B_{b+1} and set the special pointers for B^u. Note that we can do without marking B_{b+1} when the leftmost occurrence of B^u is at the start of B_b. If the leftmost occurrence of an unmarked block doesn't fulfill all conditions, we mark B^u (lines 21,23 and 24). Afterwards we generate the next level similar to all other approaches and repeat until it takes less space to store the blocks as text.

Algorithm 13: Greedy Heuristic

```
Data: String: $S[0..n-1]$, LPF-Array for $S$: LPF$[0..n-1]$, PrevOcc-Array for $S$:
      PrevOcc$[0..n-1]$, inital level size: $s$, maximum string leaf size: $m$, arity: $\tau$
Require: $n = s \cdot \tau^h$ for some integer $h$, $m = \log_\sigma n$
1  $\ell \leftarrow n/s$;
2  FirstOcc$^*$ =calculateFirstOcc(LPF, PrevOcc, $\ell$);
3  $L := \{B_0, B_1 ... B_{s-1}\}$ = partition($S, s$);
4  while $\ell > m$ do
5      for $i \leftarrow |L| - 1$ to 0 do
6          if unmarked($B_i$) then
7              $ind$ ←start($B_i$);
8              $replaced \leftarrow$ false;
9              while LPF[$ind$] $\geq \ell$ do
10                 $p \leftarrow$ FirstOcc$^*$[$ind$];
11                 if overlaps($p, ind, \ell$) or $p \leftarrow$ LPF[$p$] $\geq \ell$ then
12                     $ind \leftarrow p$;
13                 else
14                     $b \leftarrow$ binarySearch($p, L$);
15                     if consecutive($B_b, B_{b+1}$) and $B_b$.contains($p$) then
16                         mark($B_b$);
17                         mark($B_{b+1}$);
18                         $B_i.firstPointer \leftarrow B_b$;
19                         $B_i.secondPointer \leftarrow B_{b+1}$;
20                         $B_i.offset \leftarrow p-$start($B_i$);
21                         $replaced \leftarrow$ true;
22                         break;
23             if not $replaced$ then
24                 mark($B_i$);
25     $\ell \leftarrow \ell/\tau$;
26     generateNextLevel($L$);
27 storeStringLeaves($L$);
```

We show in Figure 4.2 an example block tree for S = *ABCD1234ABCDEFDEDE12* where the greedy heuristic cannot link a block to its leftmost occurrence of a sample string S = *ABCD1234ABCDEFDEDE12*. First we can view the third block *ABCD* in the first level as an unmarked block. In the second level the leftmost occurrence of the seventh

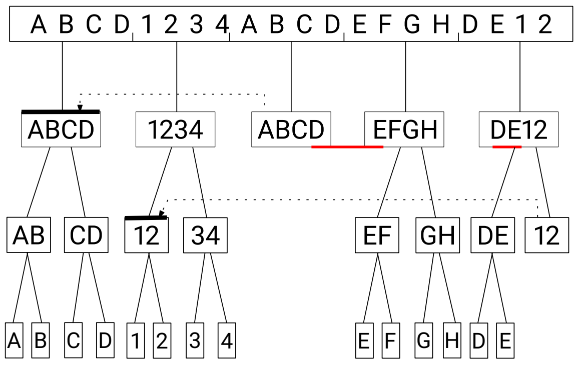


Figure 4.2: Block tree of *ABCD1234ABCDEFDEDE12* constructed with the greedy heuristic. Red underlined blocks highlight a leftmost occurrences, that is contained an already replaced block.

block *DE* is found in the first level pair $B_2 \cdot B_3$ but B_2 has no children in the second level and hence we cannot consider *DE* as unmarked.

5 Implementation

In this chapter, we provide a quick overview for our implementation used in the experimental evaluation. To allow comparisons between our implementation and the implementation provided by Belazzougui et al. [7] our implementation also has two parameters. τ determining the number of children for each unmarked block and b, setting the block size where we stop storing leftwards pointer and just store the content as text. First, we explain how we represent the block tree in memory. Next, we show how to traverse the block tree for rank, select and access queries. Then, we describe the depth first searches we use to prune and add the additional fields required for rank/select queries. Afterwards, we describe our block tree construction process. We also discuss the block tree variants we implemented. Finally, we introduce a simple parallelization in sharded memory.

5.1 Data Structure for Block Trees

In their public implementation[1], Belazzougui et al. [7] first constructed the block tree as an object-oriented tree structure and later compressed it into a data structure consisting of the following:

- A bit vector with rank support representing each block tree level. The i-th bit is set if the i-th block in a block tree level is marked, and not set for unmarked blocks. $rank_0$-queries are used to find the leftmost occurrence for unmarked blocks, and $rank_1$-queries to traverse the block tree.
- The pointers towards the leftmost occurrences for unmarked blocks are stored in an array.
- After reducing the alphabet, the last-level blocks are stored in an array.
- To answer rank and select queries for each unique symbol a in S additional information is required. For all top-level blocks, Belazzougui et al. [7] create an array storing the rank of a just before the block starts. For all other blocks B the number of a-s in B's parent before B starts is stored in level-wise arrays. In addition, for unmarked blocks B^u Belazzougui et al. [7], store $B^u.rank_a(d)$ (see Section 3.3.1) in level-wise arrays.

All arrays of integers use the minimum number of bits needed to store their maximum value. We adapt this data structure in the following way. For unmarked blocks B^u with a leftmost occurrence starting in B_i, we store two separate values, i and the starting position of the leftmost occurrence in B_i, instead of encoding them into one value. Initial experiments indicated that reading two values is faster than reading one and performing

[1]https://github.com/elarielcl/BlockTrees

an expensive decoding step, including an integer division instruction. To support rank/select queries, we store for each block B the number of occurrences for each character in the parent of B up to the end of B.

Traversing

To answer queries, it is necessary to traverse the block tree top to bottom, as well as find the leftwards pointers for unmarked blocks. Let bv_i be the bit vector representing the current block tree level i, p a marked block in i and q an unmarked block in i.
To traverse into the children of p, we can perform a $bv_i.rank_1(p)$ query on bv_i and determine the number of marked blocks before p. Each of this marked blocks creates τ children in the next level. Therefore, there are $\tau \cdot bv_i.rank_1(p)$ blocks in level $i+1$ before the children of p. To load the leftwards pointer for the unmarked block q, we can perform a $bv_i.rank_0(p)$ query and determine the number of unmarked blocks before q. Afterwards, we simply load the $bv_i.rank_0(p)$-th pointer and offset stored for i.

To add the information required for rank/select queries, we perform a post order depth first search for each unique character a. Here, we count the occurrences of a in each string leaf. For marked blocks B^m the amount of occurrences of a is the sum of the occurrences of a for each child of B^m. As we traverse the block tree in post order unmarked blocks B^u always point to blocks where we already determined the occurrences of a. Therefore, we can calculate $B^u.rank_a(d)$, $B_j.rank_a(g)$ and $B_{j+1}.rank_a(g)$ with B^u pointing to $B_j \cdot B_{j+1}$ and B^u starting at offset $g+1$. We do this by traversing downwards from B_j and B_{j+1} again and adding up the number of a up to index g.

Although we do not build on top of a general succinct tree data structure (e.g., LOUDS [26] or Balanced Parentheses [39]), we can still traverse through the block tree in the required (reverse) post order. Recall that the block tree is a balanced tree where the root has s children and all other internal nodes have τ children. To visit all nodes in post order, we can iterate over the first level bit vector bv_0 and visit each node. If $bv_0[p] = 1$ indicates a marked block, we can determine the index of each child in bv_1 with the $rank_1$ query method outlined above and visit them recursively. When $bv_0[p] = 0$ indicates an unmarked block, we can follow the leftwards pointer to an already visited block and calculate all required information.

For the reverse post order depth first search required for the pruning step, we simply iterate in reverse over the top level. We also visit all children in reverse order.

Queries

We provide query implementations similar to Belazzougui et al. [7]. Therefore, access queries take $\mathcal{O}(\log_\tau(n/b))$ time. For $rank_a$ queries, we can answer all traversing steps in constant time but store no rank/select information for leaves containing explicit text, hence we have to count occurrence of a in such leaves. This results in $rank_a$ taking $\mathcal{O}(\log_\tau(n/b) + b)$ time. Like Belazzougui et al. [7] we implement select queries by performing a binary search on the rank values of the top level, and performing sequential search on the children. Therefore, select takes $\mathcal{O}(\log s + \tau \log_\tau(n/b) + b)$ time.

5.2 Sequential Implementation

We construct the `LPF`- and `PrevOcc`-array based on Algorithm 4 by Crochemore and Ilie [13]. The `libsais`[2] library by Ilya Grebnov was used to construct the required suffix array and longest common prefix array. Suffix array construction is based on the induced sorting algorithm proposed by Nong et al. [43] and implements further optimizations [51] and a sharded memory parallelization [53]. Longest common prefix array construction is based on the ϕ-algorithm by Kärkkäinen et al. [30].

Different from Belazzougui et al. [7], we directly create the block tree data structure in the compact data structure described in Section 5.1 and do not require an additional compressing step. This is crucial as for less repetitive strings the temporary uncompressed block tree becomes too large to fit in memory, even if the final compressed block tree is small (See Section 6.1). We use the bit vector implementation[3] by Kurpicz [35] (class `pasta::bit_vector` with rank support `pasta::RankSelect`) and the compressible integer vectors [4] by Gog et al. [22] (class `sdsl::int_vector`). Our block tree implementation is publicly available at https://github.com/uqdwq/block_tree.

5.2.1 Block Tree Variants

For our experimental evaluation, we implemented the following block tree variants:

- *LPF s = z DP*: We set s to z, the number of `LZ77`-phrases, and perform the block tree construction outlined in Algorithm 12 but use binary search instead of sorting to map between text positions and blocks.
- *LPF s = z*: Similar to *LPF s = z DP* but instead of using the dynamic programming approach outlined in Section 4.2, we update `PrevOcc` one time at start to precalculate all universal chaining steps and then use the naive approach (Algorithm 8) to find the leftmost occurrences.
- *LPF s = 1*: Like *LPF s = z*, but similar to the implementation by Belazzougui et al. [7] we set s to 1.
- *LPF pruned s = 1*: *LPF s = 1*, but we apply the pruning step outlined in Section 3.3.2.
- *LPF pruned s = z*: *LPF s = z*, but we apply the pruning step outlined in Section 3.3.2.
- *LPF pruned s = z DP*: *LPF s = z DP*, but we apply the pruning step outlined in Section 3.3.2.
- *LPF heuristic s = z*: The greedy heuristic outlined in Section 4.5 with s set to z.
- *FP s = 1*: Reimplementation of the Rabin-Karp fingerprint based approach proposed by Belazzougui et al. [7] with s set to 1.
- *FP s = z*: Reimplementation of the Rabin-Karp fingerprint based approach proposed by Belazzougui et al. [7] with s set to z.

[2]https://github.com/IlyaGrebnov/libsais
[3]https://github.com/pasta-toolbox/bit_vector
[4]https://github.com/simongog/sdsl-lite

- *FP pruned s = 1*: *FP s = 1*, but we apply the pruning step outlined in Section 3.3.2.
- *FP pruned s = z*: *FP s = z*, but we apply the pruning step outlined in Section 3.3.2.

For all variants with s set to z we calculate z from the LPF array (See Algorithm 1). In addition, we cut all top levels not containing an unmarked blocked. For strings with length $n \neq s \cdot \tau^h$ for some integer h, we extend the string as described in the definition (Section 3.3), but omit blocks that are made up only by dummy symbols, as they will never be accessed.

5.3 Naive Parallelization

Initial experiments (see Figure 6.20) indicated that a significant portion of the time spent constructing the block tree is dedicated to construct the LPF array and add the necessary fields to support rank and select queries. Only a small amount of time is needed to actually construct the block tree itself. This suggests that parallelizing the LPF array construction as well as adding the fields required for rank/select queries can already speed up the block tree construction significantly.

Therefore, we provided a simple sharded memory parallelization for both steps. We parallelize LPF array construction based on the $\mathcal{O}(n)$ work and $\mathcal{O}(\log^2 n)$ time LZ77 factorization algorithm[5] by Shun and Zhao [49]. Here we use the sharded memory parallel suffix array and longest common prefix array construction provided by `libsais`. To provide rank/select support, we use a naive parallelization approach where we run a depth first search for each unique character in parallel. This approach has a limited effect on inputs with a small alphabet (e.g., DNA texts with up to 15 unique characters) or on machines with a high number of cores.

[5]Their implementation is available at https://github.com/zfy0701/Parallel-LZ77

6 Experiments

We experimentally compare the performance of our different block construction methods with the known implementation by Belazzougui et al. [7], considering construction times, block tree sizes and the answer times for rank, select and access queries. Finally, we ran an initial experiment evaluation a naive parallelization for our block tree construction attempt.

6.1 Setup

We conducted our experiments on an AMD EPYC Rome 7702P (64 cores/128 threads with frequencies up to 3.35GHz and 256 MiB L3 Cache) and 1024 GiB DDR4 ECC RAM running Ubuntu 20.04.2 LTS. We compiled the code with GCC 12.1 with the -O3 and -march=native flags enabled. As the Code evaluated in Section 6.2.1 is sequential, only one core was used at a time. For the evaluation of the simple parallelization, the programs were compiled with GCC 12.1 with the -O3, -march=native and -fopenmp flags enabled.

During our evaluation, we collect the following data for each variant and parameter configuration (τ, b): Block tree construction time (with/without rank/select support), block tree size (with/without rank/select support) as well as the average answer time for access, rank and select queries. To achieve this, we generate a $1.000.000$ random text position. For access queries, we report the average answer time on these text positions. For rank and select queries, we translate these text positions into a meaningful query based on the relative frequency for each character. We first calculate the cumulative histogram C for S. Afterward, we choose the character c associated with the bucket that contains each random text position i. For rank queries, we use i and c as parameters. For select queries, we use $i - C[c-1]$ and c to create a valid select query. We calculate the block tree size by adding up the size of all underlying data structures required for the respective query. The reported times are the average of three runs (each with new queries, but we test the same queries on all variants during each run).

Belazzougui et al. [7] evaluated their block tree implementation[1] on the Repetitive Corpus of the *Pizza&Chili* platform[2]. We reproduce the experiment on eight real texts from the Repetitive Corpus and additionally evaluate block trees on four texts from the standard (and less repetitive) Corpus of the *Pizza&Chili* platform[3]. Table 6.1 lists the texts with basic statistics, including values measuring the Lempel-Ziv compressibility. A theoretical value, the Lempel-Ziv compression factor $\frac{z \log n}{n \log \sigma}$ [7] and a practical measurement by measuring the compressibility using **p7zip**[4].

[1]https://github.com/elarielcl/BlockTrees
[2]http://pizzachili.dcc.uchile.cl/repcorpus
[3]http://pizzachili.dcc.uchile.cl/texts.html
[4]https://p7zip.sourceforge.net/

Collection	n	σ	z	$\frac{z \log n}{n \log \sigma}$	p7zip
cere	461286644	5	1700630	0.044	5.35%
para	429265758	5	2332657	0.064	6.05%
einstein.en.txt	467626544	139	89467	0.0007	0.10%
kernel	257961616	160	1446468	0.021	2.53%
coreutils	205281778	236	793915	0.013	11.75%
influenza	154808555	15	769286	0.033	1.69%
escherichia coli	112689515	15	2078512	0.121	7.76%
world leaders	46968181	89	175740	0.014	1.39%
sources	210866607	230	11598459	0.194	15.84%
pitches	55832855	133	5994276	0.391	25.89%
proteins	1184051855	27	80408252	0.430	31.30%
dna	403927746	16	25628189	0.453	22.79%
english	1610612736	239	97047354	0.233	26.11%
xml	296135874	97	9576081	0.138	12.74%

Table 6.1: The sequences we use, with their size n symbols (we use one byte per symbol), their alphabet size σ, the number of Lempel-Ziv factors z (calculated with Algorithm 1), a measure of compressibility $\frac{z \log n}{n \log \sigma}$ and the compression achieved with p7zip (Version 16.02). Note that we do not use the whole english text, but only a prefix.

We present graphs comparing the effects of different approaches and configurations. To keep the graphs clear, we split the approaches in different categories and compare by category:

1. Block trees, faithful to the theoretical proposal, therefore not including the pruning step. This includes the following variants from Section 5.2.1:
   LPF s = z, *LPF s = z DP*, *LPF s = 1*, *FP s = 1* and *FP s = z*.

2. Pruned block trees. This includes the following variants from Section 5.2.1:
   LPF pruned s = z, *LPF pruned s = z DP*, *LPF pruned s = 1*, *FP s = 1*, *FP s = z*, *LPF heuristic s = z* and implementation by Belazzougui et al. [7].

We include the implementation by Belazzougui et al. [7] as a baseline into the graphs of the first category, but we only discuss the performance in relation to our pruned block tree implementations.

During our initial experiments, we noticed that the implementation by Belazzougui et al. [7] uses large amount of memory when constructing the block tree for the less repetitive sequences from the standard *Pizza&Chili* Corpus. We believe this is due to their approach of first constructing the block tree as an object-oriented pointer-based graph and later parsing into a compact data structure. This becomes especially apparent, when we add rank support to the data structure for large alphabets. Here, even on a machine with 1 TiB of main memory, it is only possible to construct the block tree for prefixes of the *english* text smaller than 200 MiB and constructing the block tree for the 100 MiB prefix took nearly 4 hours. Therefore, we also report the results for the 32 MiB prefixes of the standard *Pizza&Chili* Corpus to allow for some comparisons. Furthermore, the implementation by Belazzougui et al. [7] does not terminate successfully for the two larger texts *protein* and *english* (> 1 GiB) in the standard *Pizza&Chili* Corpus even when we do not add rank support. As constructing the block tree for the *english* text with rank support takes also at least 3 hours for a single run for all of our configurations and block tree variants, we decided to remove the *english* text dataset from our benchmark dataset as proper benchmarks would exceed our allocated time budget and only provide data on the 32 MiB prefix.

6.2 Sequential Evaluation

The results of our experiment are presented in 2D grids, split by text input. Each point on the graph represents the results of a possible configuration (variant, (τ, b)) for construction size, construction time, and average query time for a given text input. The x-axis of the graph represents construction size in bits per symbol input text, while the y-axis represents construction time in seconds on a logarithmic scale or average query time in nanoseconds on a linear scale. All graphs in a figure share the same y-axis, but we use different scales on the x-axis due to the larger variance in block tree size. Note that, our implementation produces same sized block trees for different parameter pair (τ, b) (e.g, a block trees with $\tau = 16$ and $\tau = 8$ are identically in structure if the first level size is smaller than $8b$). Here, the figures show the average over similar data points instead of 3 different data points. Furthermore, we consider the repetitive and standard corpus separately. When we set $s = z$ for some configuration, the reported construction time includes the time to calculate LPF and count the number of LZ77-phrases. As the implementation by Belazzougui et al. [7] creates multiple compact block tree representations during the final step, we only report time spent calculating the initial block tree presentation, performing the pruning step and if enabled the time spent adding rank support.

For the texts included in the repetitive *Pizza&Chili* Corpus, we run benchmarks for the following configurations $\tau \in \{2, 4, 8, 16\}$ and the max length of last level leaves $b \in \{2, 4, 8, 16\}$. For the texts included in the standard *Pizza&Chili* Corpus, we ran our expriments with $\tau \in \{2, 4, 8\}$ and the max length of last level leaves $b \in \{2, 4, 8\}$.

6.2.1 Construction Time and Tree Size

Block Trees Without Pruning. We begin our evaluation by comparing our Rabin-Karp fingerprint based approaches to the LPF based approaches without applying the pruning step. Figure 6.1 shows the relationship between construction time and final block tree size for the block tree faithful to the theoretical proposal on repetitive texts without adding rank support. Comparing our implementations for LPF and Rabin-Karp fingerprint based approaches, the approaches based on the LPF array are clearly in the Pareto-front in terms of construction time, but the speed-up increases further for smaller block tree configurations.

For repetitive texts, *LPF s = 1* constructs the smallest configurations 7.3–11.5 times faster than *FP s = 1*, overall the speed-up varies between 1.9–11.5 depending on configuration and input text. The average speed-up is 5.53. For $s = z$ the speed-up for the smallest block trees varies between 3.2–5, while in general the speed-up varies between 1.8–5 with an average speed-up of 3.04. On average, configurations for $s = z$ are 2.2% larger than their $s = 1$ counterpart, but the construction time speeds up on average by 81.2% for *FP* $s = z$, but we cannot determine a significant speed-up for *LPF* $s = z$.

Note that, the block tree size decreases for smaller values of τ, s and b, but the height of the block tree increases, which leads to more symbols being processed during construction. This indicates that LPF based approaches may have smaller constant factors. Furthermore, it becomes clear when we consider that spending extra time to calculate z to increase s leads to an 81.2% faster on average construction for *FP* $s = z$, while it does not decrease construction time for *LPF* $s = z$. We believe this is due to LPF construction dominating the construction time for repetitive texts without rank support for *LPF* $s = z$ and *LPF* $s = 1$ (see Figure 6.20). Recall that block trees allow for time-space-tradeoffs, where we can increase the size and therefore decrease the query time. Therefore, it is not necessarily wanted to construct the smallest block tree.

Figure 6.2 shows the relationship between construction time and final block tree size for the block tree faithful to the theoretical proposal on repetitive texts adding rank support. Here an interesting observation is that not for all texts the smallest configuration is the configuration with the longest construction time. We believe this is due to the fact that we implemented a less efficient rank query, where we forgo constructing a binary rank data structure for all last level leaves, and instead just scan over the leaves and count the number of c for a $rank_c$ query. Therefore, having fewer levels in the block tree seems to be beneficial for sequences with a large alphabet, as it not only decreases query times but also decreases block tree size, when rank support is enabled. Note that, we observe similar behavior for the implementation from Belazzougui et al. [7] as they implemented a similar rank query. The last level leaf sizes b used in our experiments are rather small and are focused on reducing the size of the block tree structure without considering the rank support.

With enabled rank support on repetitive texts, we observe that *LPF* $s = 1$ constructs block trees 3.75 times faster on average in comparison to *FP* $s = 1$, depending on configuration and input text. *LPF* $s = z$ is 2.23 times faster in comparison to *FP* $s = z$. We observe overall a smaller speed-up for all LPF-based variants over their fingerprint-based counterpart, as adding the rank support should take the same time for both. When considering the size differences between $s = z$ and $s = 1$ variants, we can see in Figure 6.2, $s = z$ variants have significantly larger block trees compared to $s = 1$. With otherwise the same parameters, the block trees with $s = z$ are 43.9% larger on average.

Figures 6.3 and 6.4 show the relationship between construction time and block tree size for the less repetitive text in the standard *Pizza&Chilli* Corpus. Overall, we see similar behavior when not adding rank support. *LPF* $s{=}1$ is 8.8 times faster on average than *FP* $s{=}1$ (between 3.6–15.3 times faster), while *LPF* $s{=}z$ is 3.5 times faster on average than *FP* $s{=}z$ (between 2.1-5.9 times faster). Further, indicating that the LPF-based takes less time per processed symbol as the speed-up increases when compared to more repetitive texts. When adding rank support, this decreases to 3.54 and 1.59 times on average, respectively.

Pruned Block Trees Figures 6.5 and 6.6 show the relation between construction time and block tree size on repetitive text. Figures 6.6 and 6.7 show the relation between construction time and block tree size on the less repetitive text in the standard *Pizza&Chilli*

Corpus. Notice that, for both, *LPF pruned s=z* and *LPF pruned s=1* construct the block tree faster than their *FP pruned s=z* and *FP pruned s=1* counterparts. This is particularly evident for $s = 1$ on the less repetitive texts in Figures 6.6 and 6.7. Hence, we believe that not updating `FirstOcc`* after every level and instead using the naive scan outlined in Algorithm 8 might also be efficient, but we cannot provide any runtime guarantees at the time. During our following evaluation, we will therefore only compare *LPF pruned s=z* and *LPF pruned s=1* to other approaches.

The heuristic *LPF heuristic s = z* did not speed up the construction times, when compared to *LPF pruned s = z*. Overall, for repetitive texts the construction time for *LPF pruned s = z* was actually 1% faster on average, while the block trees were of similar size and only slightly larger for the heuristic. When including adding rank support, construction with *LPF pruned s = z* was 2% faster on average. For the less repetitive texts, *LPF heuristic s = z* was slightly faster (4% on average) without rank support, but there is no speed-up when adding rank support. Note that, for both approaches, the `LPF` construction time and the σ depth first searches calculate the samples used for rank and select queries dominate the overall run and therefore the runtime effects of this approximation. Therefore, we conclude that our heuristic did not show any practical value.

Finally, we compare *LPF pruned s=z* and *LPF pruned s=1* against the fingerprint based approaches *FP pruned s=z* and *FP pruned s=1* and the implementation by Belazzougui et al. [7]. We first compare *LPF pruned s=1* to the implementation by Belazzougui et al. [7], as it has the s set to 1 and therefore should create structural similar block trees. On the repetitive texts in Figures 6.5 and 6.6 we observe that our approach to construct block tree is 6.99 times faster on average (between 2.66–14.75 times based on input text and configuration), while for the smallest block tree construction on each text, it is 11.88 faster on average (between 8.21–14.75 times). When taking rank and select support into consideration, construction is 7.13 times faster on average. Again, speed-up increases for smaller block tree configurations. Note that, our implementations create larger block trees for texts with a small alphabet size. This is due to a missing space optimization in our implementations, where we do not reduce the alphabet before storing the last level leaves explicitly. The issue becomes more apparent for configurations that store large parts of the string explicitly (e.g., when using a larger value for b). On average, our pruned block trees for $s = 1$ are 14% larger than the block trees by Belazzougui et al. [7] on the repetitive texts. With rank support enabled, this reduces to 11% as we store similar extra data.

As for the less repetitive texts in Figures 6.6 and 6.7, the implementation by Belazzougui et al. [7] ran out of memory (see Section 6.1) for larger texts or when adding rank support, therefore we can only compare it with our implementation for the available data in Figure 6.7. Here, *LPF pruned s=1* constructs the block trees 14 times faster on average (between 8.19–23 times faster) and 19 times faster on average for the smallest configurations of each text. Figures 6.9 and 6.10 show the results for the 32 MiB prefixes of the standard *Pizza&Chilli* Corpus, where *LPF pruned s=1* is still 11.66 times faster on average, showing that the issue also implementation by Belazzougui et al. [7] effects runtime. When also considering the time needed to add rank support (Figure 6.10) *LPF pruned s=1* is 9.65 times faster on average. But for most texts all block tree configurations are significantly larger than the input text (1Byte per symbol), the only exception are texts with a small alphabet size (e.g., dna32MB and proteins32Mb). If we consider the block tree size while only supporting access operations, all configurations tested, take less space than the original text.

When using the same configurations, *FP pruned s=1* is 1.24 times faster on average compared to the implementation by Belazzougui et al. [7] on repetitive texts and 1.81 times faster on less repetitive texts. Therefore, in a last comparison, we analyze the construction time differences between and *FP pruned s=z* and *FP pruned s=1*, showing similar behavior to their not pruned counterparts. Without rank support, *FP pruned s=z* is 1.81 times faster on average than *FP pruned s=1* on repetitive texts and 2.52 times faster on less repetitive texts.

Overall, the pruned block trees without rank support and $s = z$ are 5% larger on average for repetitive texts than their $s = 1$ counterpart and 4% larger on average for less repetitive texts. When also considering the extra space required for rank support, setting $s = z$ creates 38% and 60% larger block trees on average. This can be explained by considering that block trees with $s = z$ (i) store additional larger rank samples for the first level, (ii) have more blocks and therefore store more samples.

We would also like to point out that our LPF based implementations have a large memory footprint, as we require two integer values per symbol input text during the block tree steps (and up to $6n$ integer cells during LPF construction). This is significantly more than the hash tables for fingerprint based approaches require, but predeterminable.

6.2.2 Query Times

We report the query times for our block tree variations *LPF heuristic s = z*, *LPF s = z*, *LPF pruned s = z*, *LPF s = 1*, *LPF pruned s = 1* and the implementation by Belazzougui et al. [7]. Note that, all block tree variations but the heuristic *LPF heuristic s = z* construct the same block tree for parameters s, τ, b. Hence, we will simplify the evaluation by only considering the variations mentioned above.

Access. Figures 6.11, 6.12 and 6.13 show the average access time in relation to the block tree size. Generally, we observe a clear time-space-tradeoff, where smaller configurations report longer access times, for all variants. When comparing the implementation by Belazzougui et al. [7] to *LPF pruned s = 1*, we notice that for texts with a small alphabet (e.g., para and cere in Figure 6.11) their implementation constructs smaller block trees.

Analyzing this issue, we found that our implementations do not reduce the alphabet, when storing the last level leaves, and use a flat 1Byte per symbol instead. Comparing the access time, the implementation by Belazzougui et al. [7] is, on the texts in Figure 6.11, on average 5.1% faster. Overall, the average access queries are between 0.83 and 1.37 times faster based on the input text. As we implement the same query algorithm, one explanation for this difference in performance could be the internal bit vector implementation used. Although bit vectors can answer rank and select queries in constant time, there are time-space-tradeoffs in different implementations. Kurpicz [35] compared several bit vector implementations, showing that the implementation we use has a smaller memory footprint of just 3.51% extra space while the implementation, *sdsl_v* contained in the SDSL [22], used by Belazzougui et al. [7] uses 25% extra space, but answers rank queries on small bit vectors significantly faster [35]. We believe this is a valid tradeoff, as bit vectors and their rank data structure make up only a small part of the block tree. For completeness, we measure that the competitor answer on average 27% faster access queries on the standard Corpus in Figure 6.12.

Furthermore, *LPF pruned s = z* performs access queries on average 44.2% faster than *LPF pruned s = 1* on repetitive texts, while the block tree is on average 5% larger. Highlighting furthermore the impact of reducing block tree height on query times. For the standard Corpus in Figure 6.12 we report access queries being on average 2.17 times faster, while the block tree is on average 4% larger.

Finally, the heuristic *LPF heuristic s = z* performs access queries in roughly the same time as the canonical *LPF pruned s = z*, while being only slightly larger. This is expected as both variations have the same height and also roughly the bit vector lengths on all levels.

Rank and Select. Figures 6.14, 6.15 and 6.16 show the average rank time in relation to the block tree size. Note that, other than in the previous paragraph for access queries, we can observe the space-time-tradeoffs only for highly repetitive texts or repetitive texts with a small alphabet (e.g., *einstein, para and cere* in Figure 6.14). We believe this is due to the inefficient rank implementation, and counting the cs in a longer string leaf is faster than traversing extra block tree levels and then count the cs in a shorter string leaf.

When comparing rank query times for *LPF pruned s=1* and the implementation of Belazzougui et al. [7], there is no speed-up on average for the repetitive texts, but *LPF pruned s=1* answers 26% faster on average for the 32 MiB prefixes of less repetitive texts. Note that *LPF pruned s=z* answer rank queries faster than *LPF pruned s=1* as fewer block tree levels reduces the necessary work. Although this comes at the cost of significantly larger block trees. For repetitive texts in Figure 6.14, *LPF pruned s=z* is 35% faster on average, but require 24% more space on average. For the less repetitive texts in Figure 6.15, *LPF pruned s=z* answers 2.1 faster on average, but require, 46% more space on average.

Figures 6.17, 6.18 and 6.19 show the average select time in relation to the block tree size. Similar to rank queries, we can observe the space-time-tradeoffs only for highly repetitive texts like or repetitive texts with a small alphabet (e.g., *einstein, para and cere* in Figure 6.17). We believe this is due to the inefficient select implementation, and counting the cs in a longer string leaf is faster than traversing extra block tree levels and then count the cs in a shorter string leaf. When comparing select query times for *LPF pruned s=1* and the implementation of Belazzougui et al. [7], similar to rank queries there is no speed-up on average for the repetitive texts, but *LPF pruned s=1* answers 10% faster on average for the 32 MiB prefixes of less repetitive texts. Block trees with $s = z$ perform select queries significantly slower than their $s = 1$ counterparts. This is due to the inefficient select query performing a binary search on the first level, which takes longer for larger s. *LPF pruned s=1* answer select queries 20% faster than *LPF pruned s=1* on repetitive texts, while in the block tree with $s = z$ are 24% larger on average. For the less repetitive texts in Figure 6.15, *LPF pruned s=z* answers 54% faster on average and *LPF pruned s=1* requires 46% more space on average.

Overall, we come to the conclusion, that our query implementation answers roughly similar times as the implementation of Belazzougui et al. [7]. Furthermore, the additional space-time-tradeoff we implemented was beneficial for access queries, but leads to uninspiring results on rank and select queries. Next step, is to implement rank and select queries faithful to their theoretical proposal. As these improvements require additional data structures, and we already create large block trees for less repetitive texts, block trees might be impractical for less repetitive texts. A next step could be to compare the block trees to well-researched rank/select/access data structures like the wavelet tree [23] and wavelet matrix [12]. These can be compressed using entropy encoders [37]. Initial experiments indicated that wavelet trees can be constructed at least an order of magnitude faster than our block trees implementations.

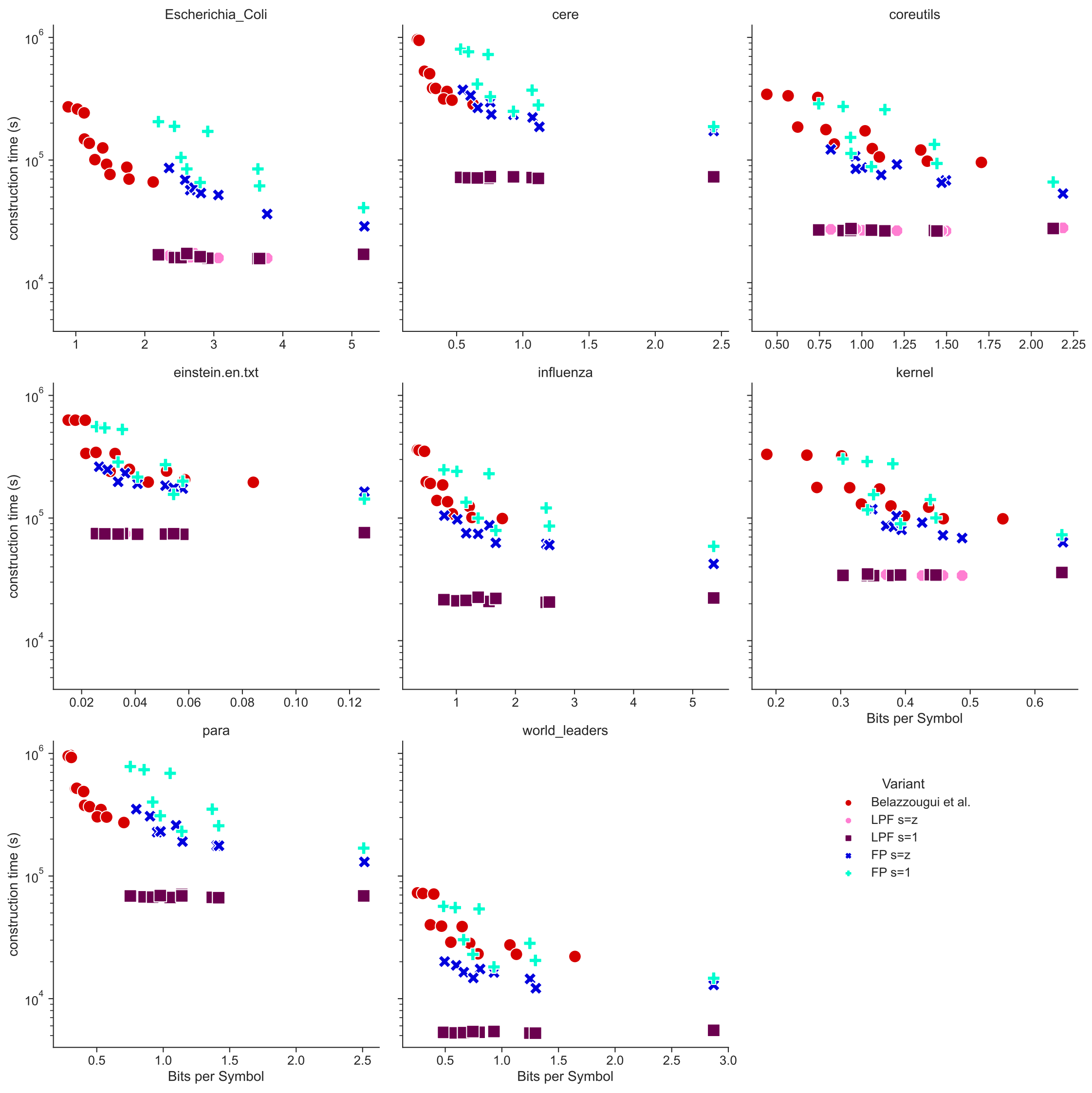


Figure 6.1: Block tree construction time in relation to the block tree size without pruning on repetitive texts. The x-axis shows the block tree size in bits used per symbol input text, while the y-axis shows the block tree construction time in seconds on a log-scale. Note that each graph has an individual scale for the x-axis.

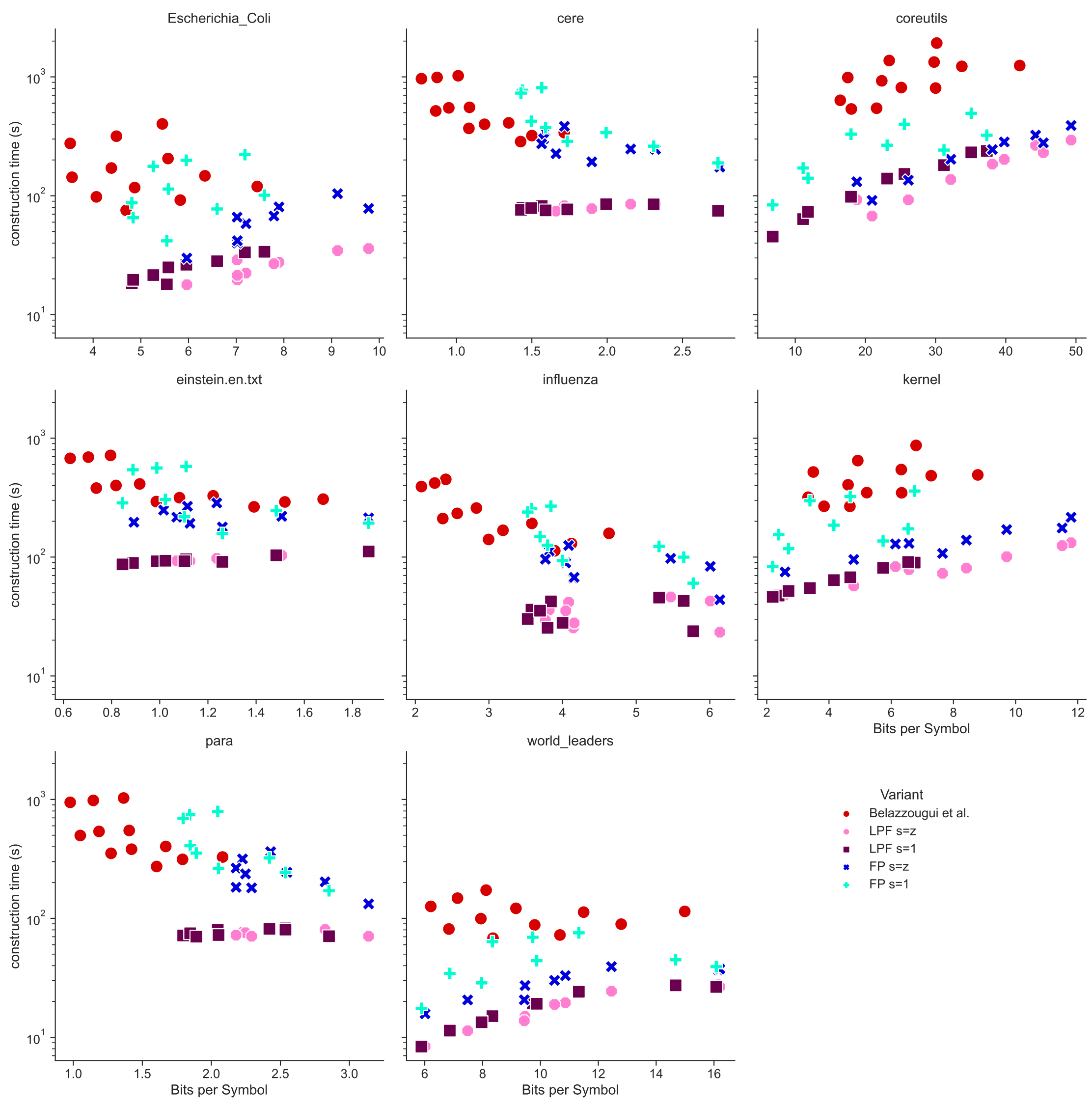


Figure 6.2: Block tree construction time (with rank/select support) in relation to the block tree size without pruning on repetitive texts.

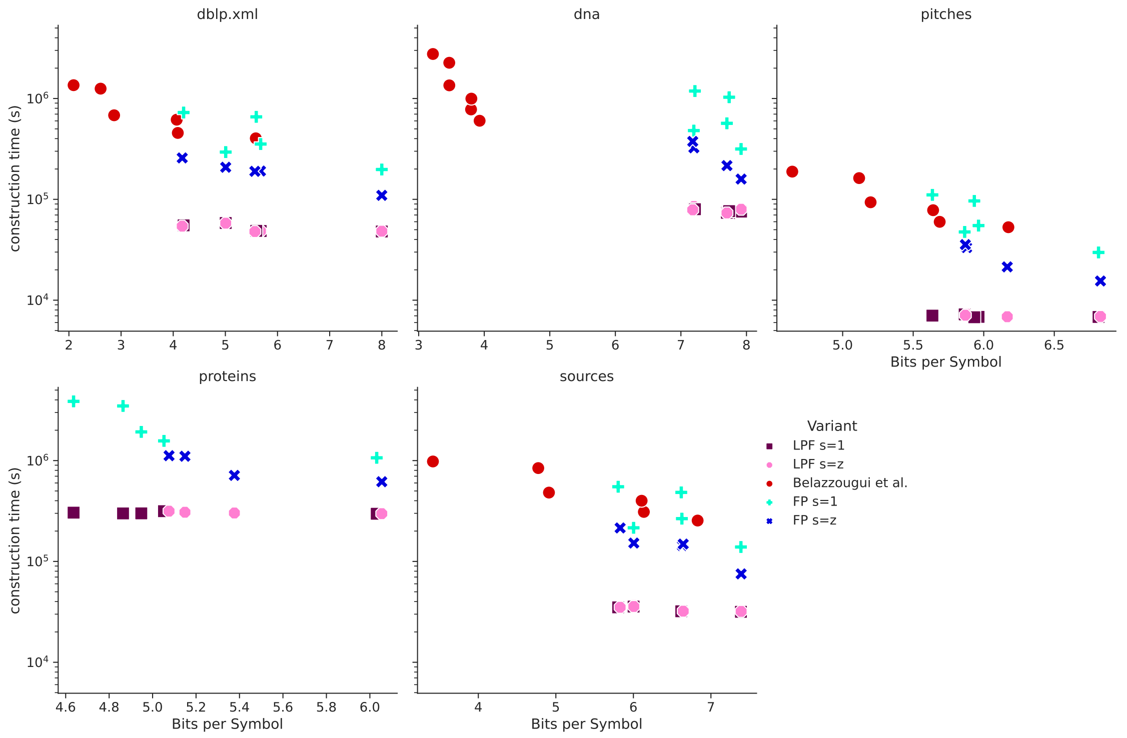


Figure 6.3: Block tree construction time in relation to the block tree size without pruning on less repetitive texts.

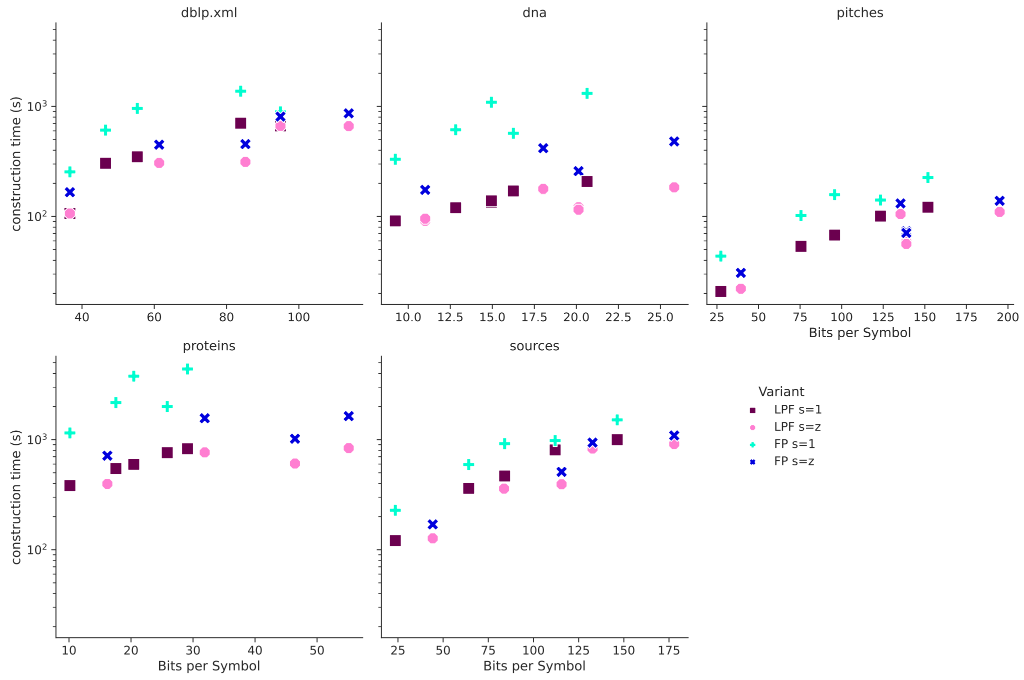


Figure 6.4: Block tree construction time (with rank/select support) in relation to the block tree size without pruning on less repetitive texts.

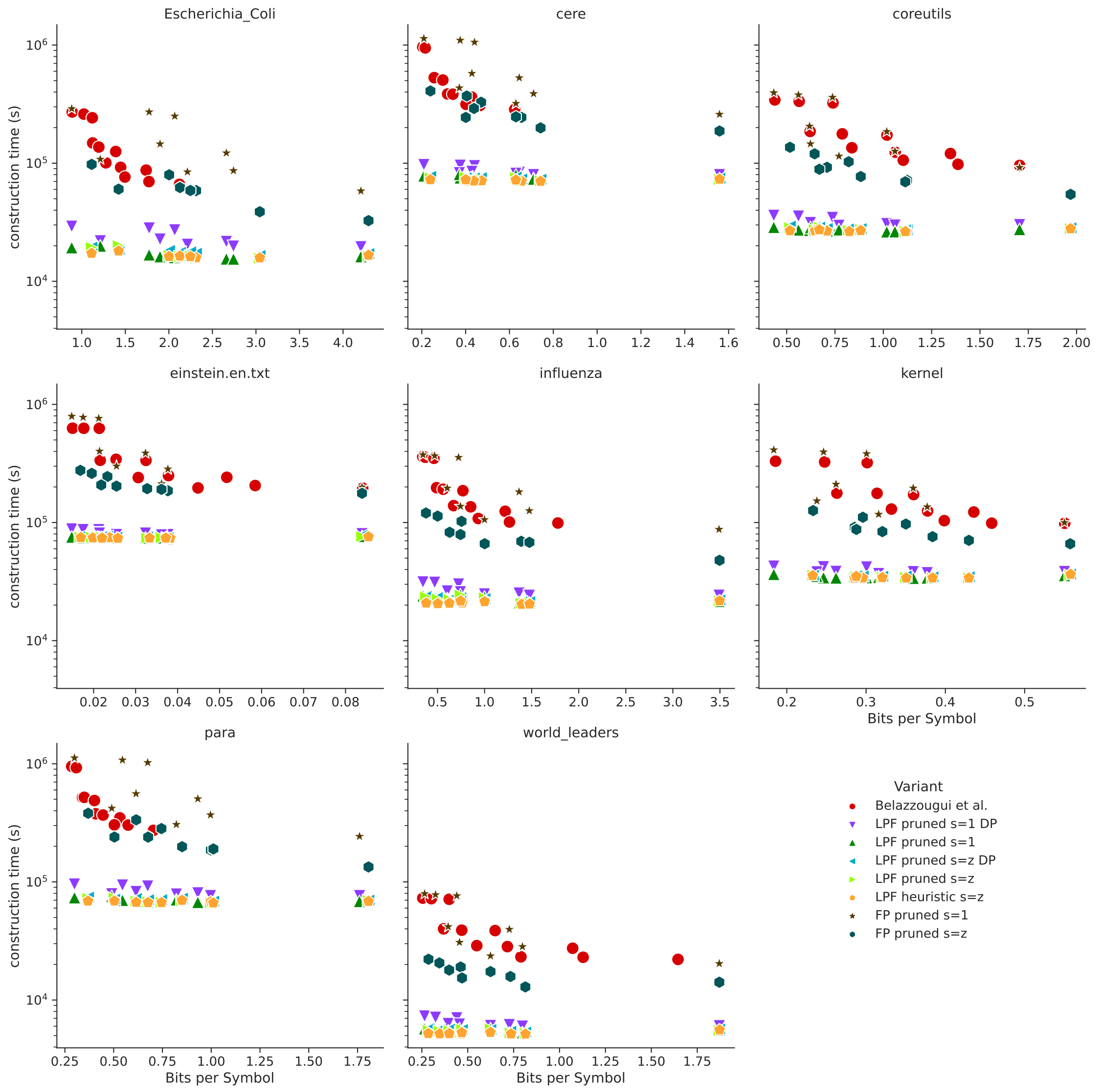


Figure 6.5: Block tree construction time in relation to the block tree size, with pruning enabled on repetitive texts.

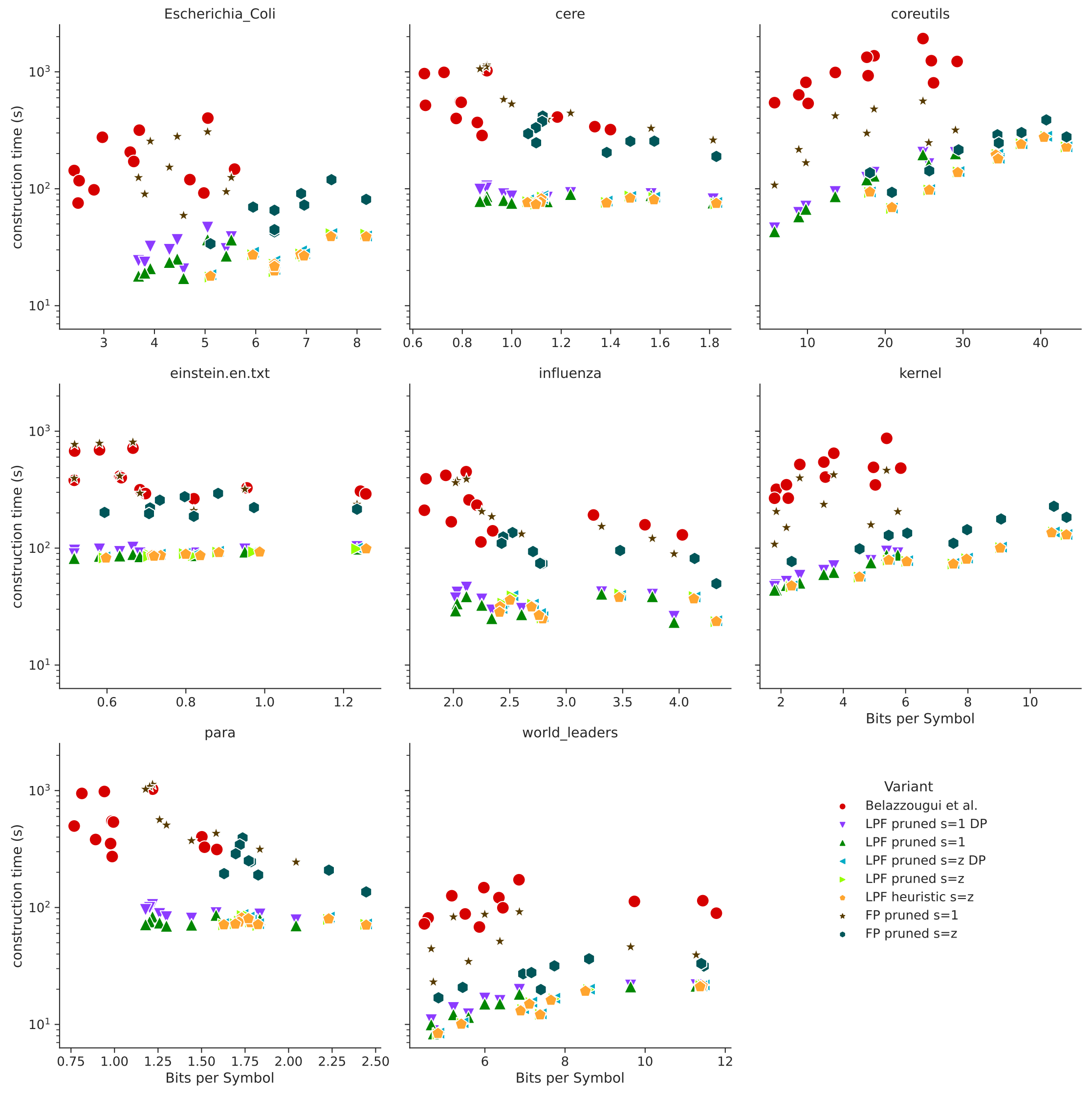


Figure 6.6: Block tree construction time (with rank/select support) in relation to the block tree size, with pruning enabled on repetitive texts.

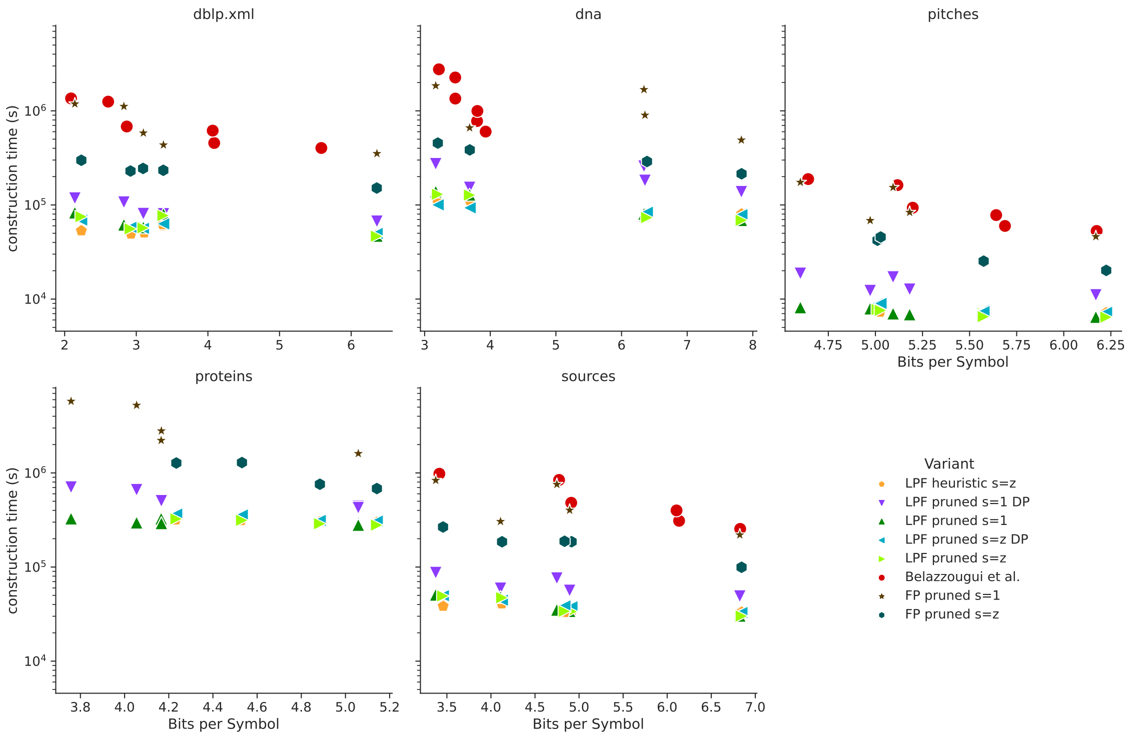


Figure 6.7: Block tree construction time in relation to the block tree size, with pruning enabled on less repetitive texts.

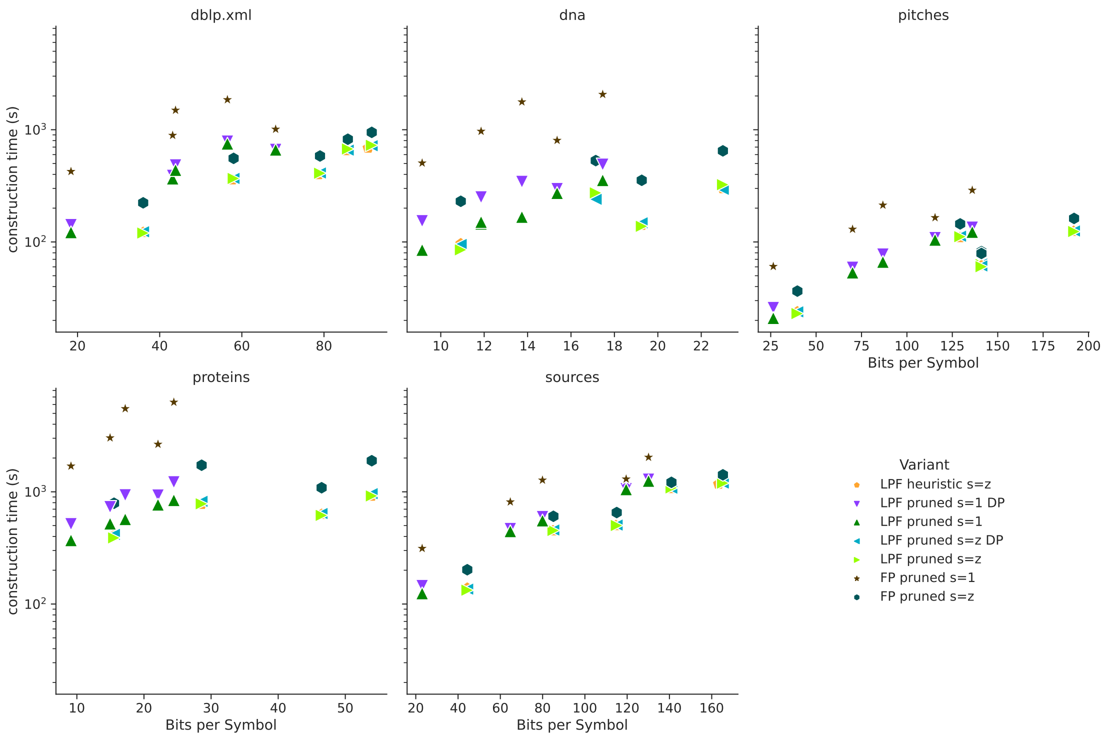


Figure 6.8: Block tree construction time in relation to the block tree size, with pruning enabled on less repetitive texts.

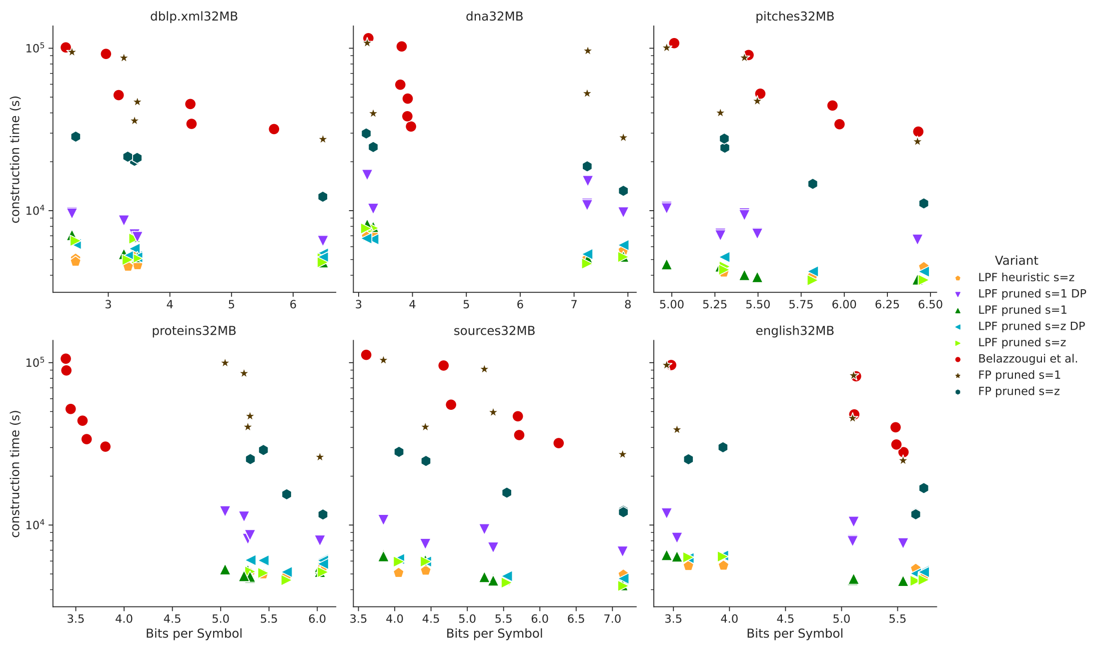


Figure 6.9: Block tree construction time in relation to the block tree size, with pruning enabled on the 32 MiB prefixes of less repetitive texts from the standard *Pizza&Chilli* Corpus.

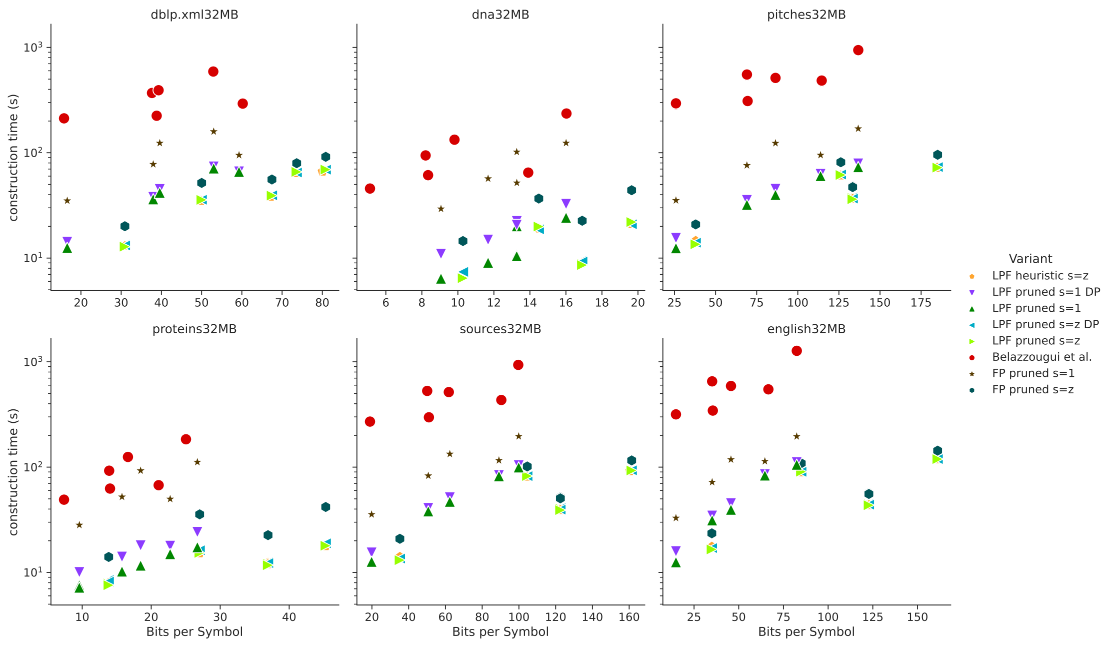


Figure 6.10: Block tree construction time (with rank/select support) in relation to the block tree size, with pruning enabled on the 32 MiB prefixes of less repetitive texts from the standard *Pizza&Chilli* Corpus.

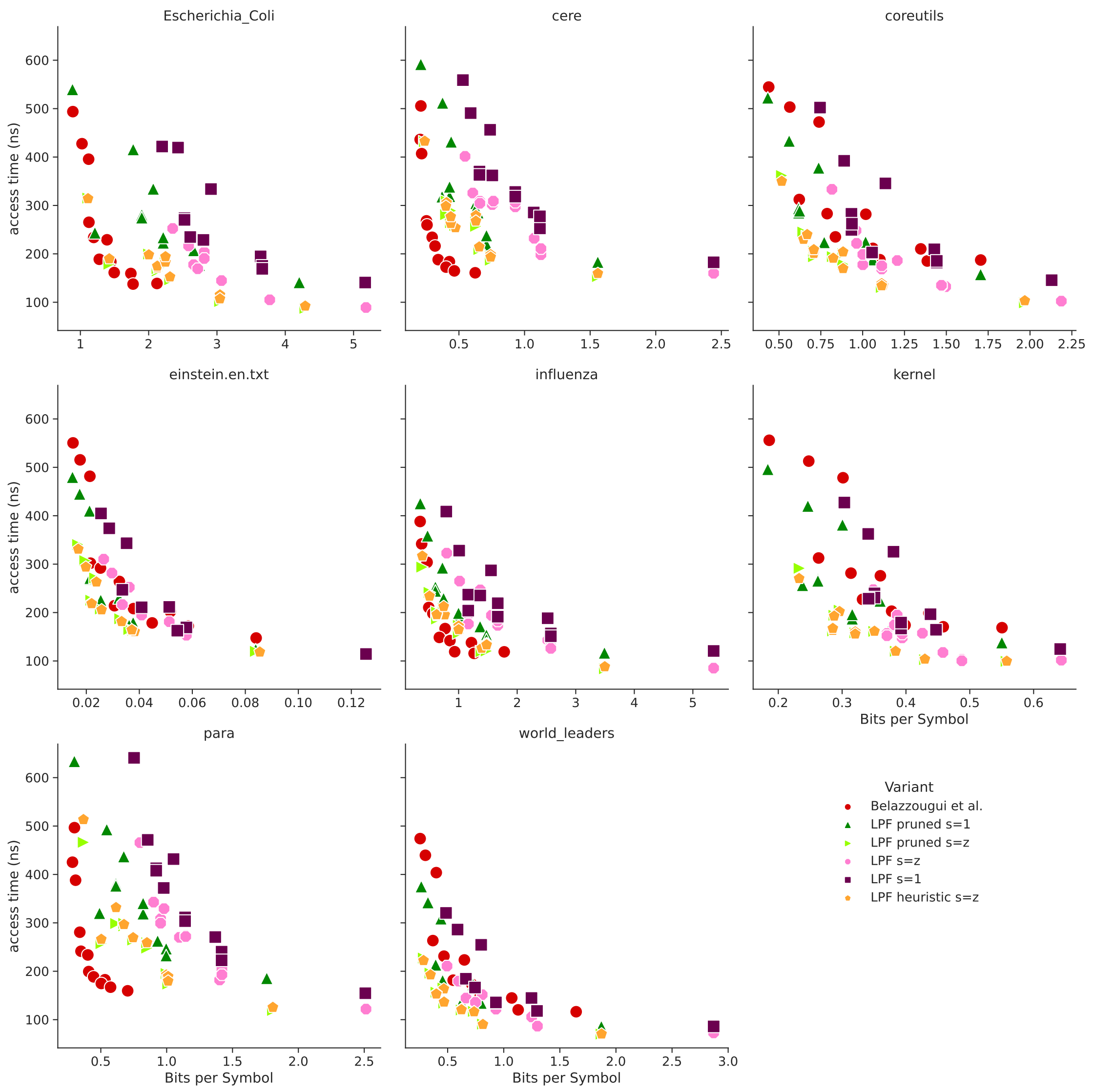


Figure 6.11: Average access time in relation to the block tree size for repetitive texts. The x-axis shows the block tree size in bits used per symbol input text, while the y-axis shows the average access time in nanoseconds on a linear scale. Note that each graph has an individual scale for the x-axis.

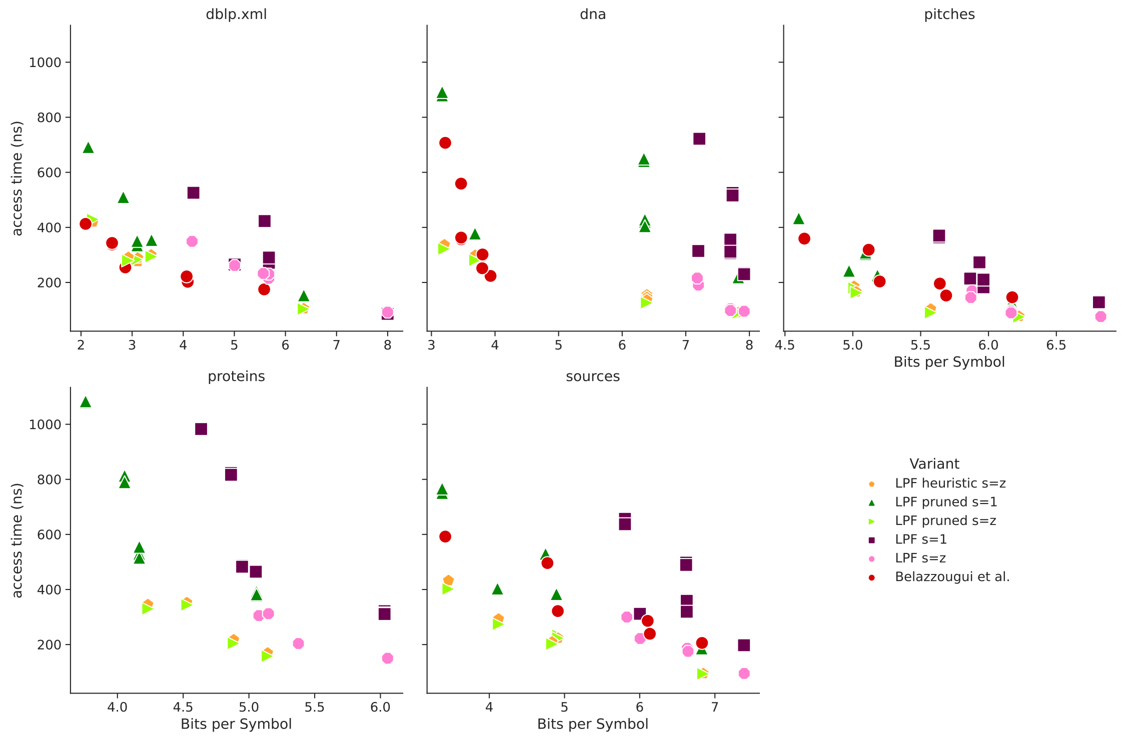


Figure 6.12: Average access time in relation to the block tree size for the less repetitive standard *Pizza&Chilli* Corpus.

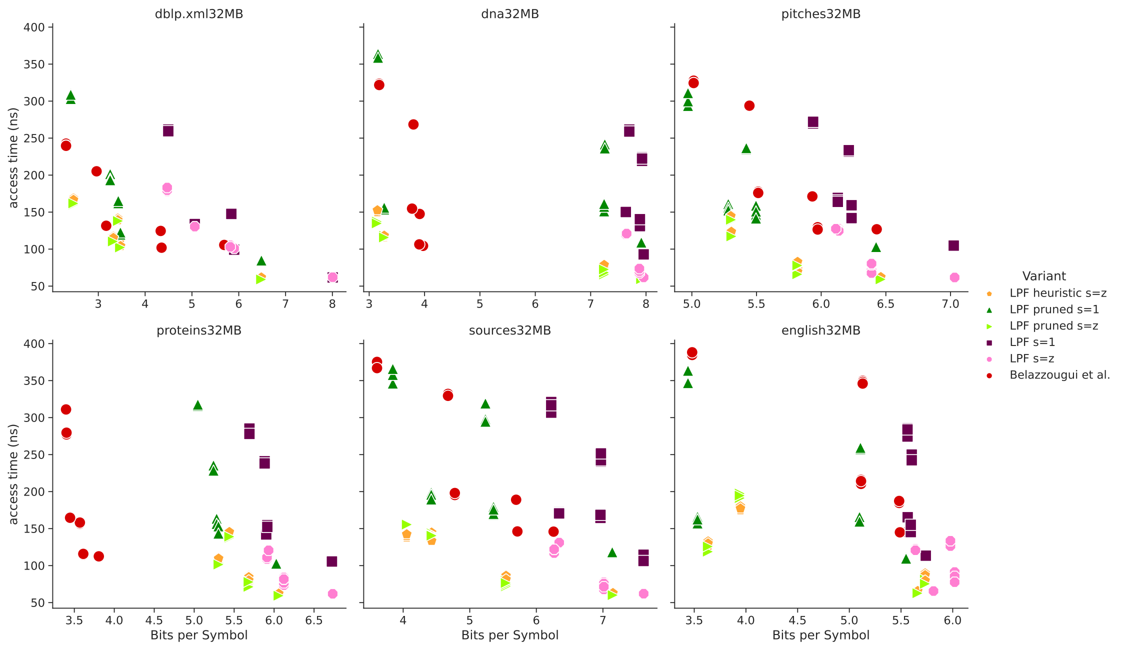


Figure 6.13: Average access time in relation to the block tree size for the 32 MiB prefixes for texts in standard *Pizza&Chili* Corpus.

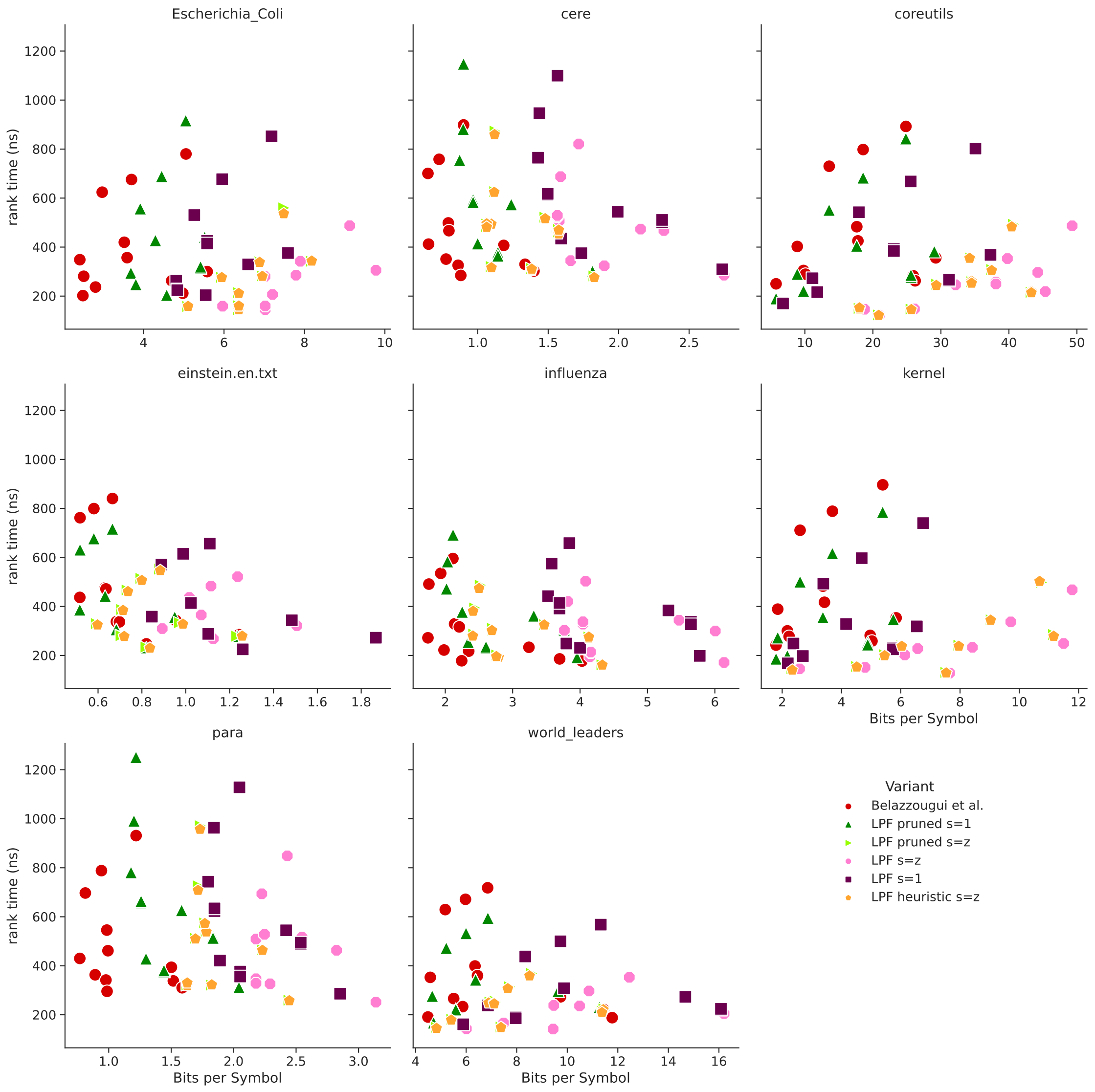


Figure 6.14: Average rank time in relation to the block tree size for repetitive texts. The x-axis shows the block tree size in bits used per symbol input text, while the y-axis shows the average rank time in nanoseconds on a linear scale. Note that each graph has an individual scale for the x-axis.

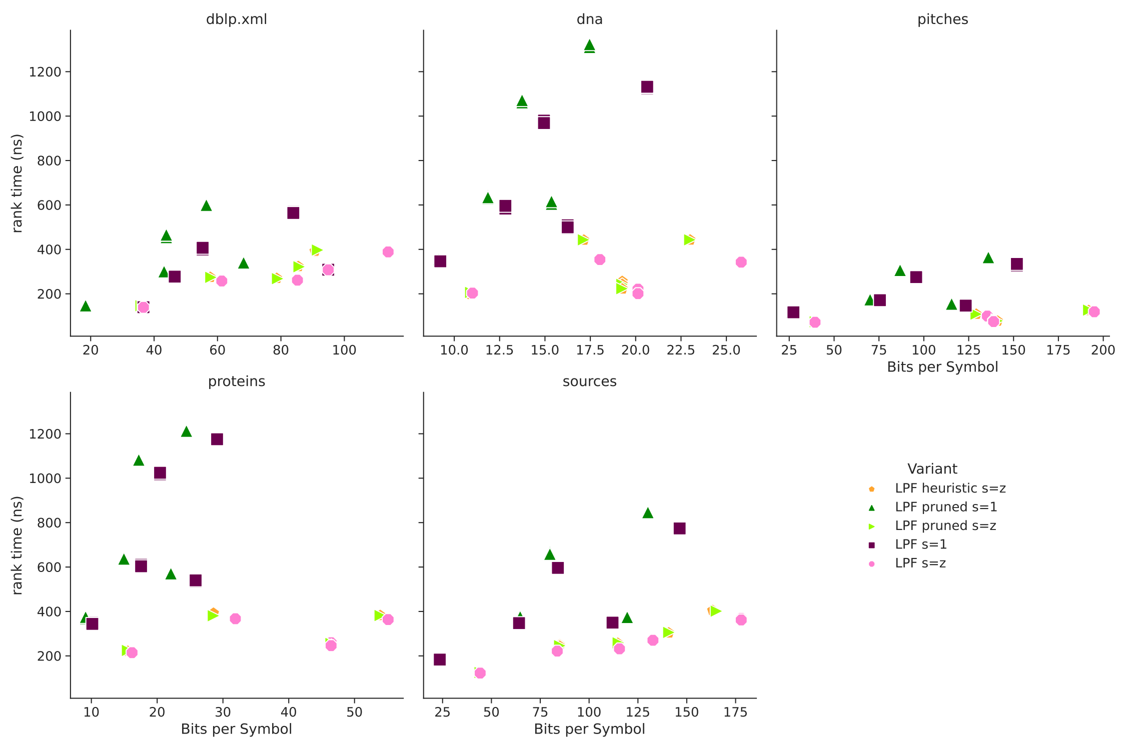


Figure 6.15: Average rank time in relation to the block tree size for texts in the standard *Pizza&Chili* Corpus.

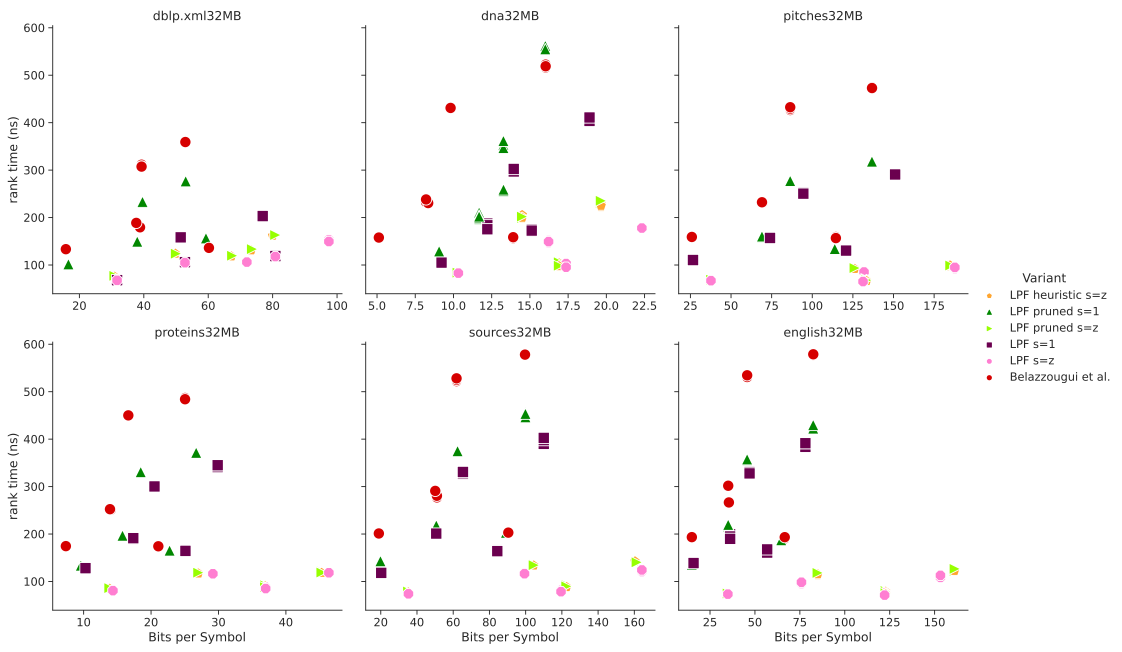


Figure 6.16: Average rank time in relation to the block tree size for the 32 MiB prefixes for texts in standard *Pizza&Chili* Corpus.

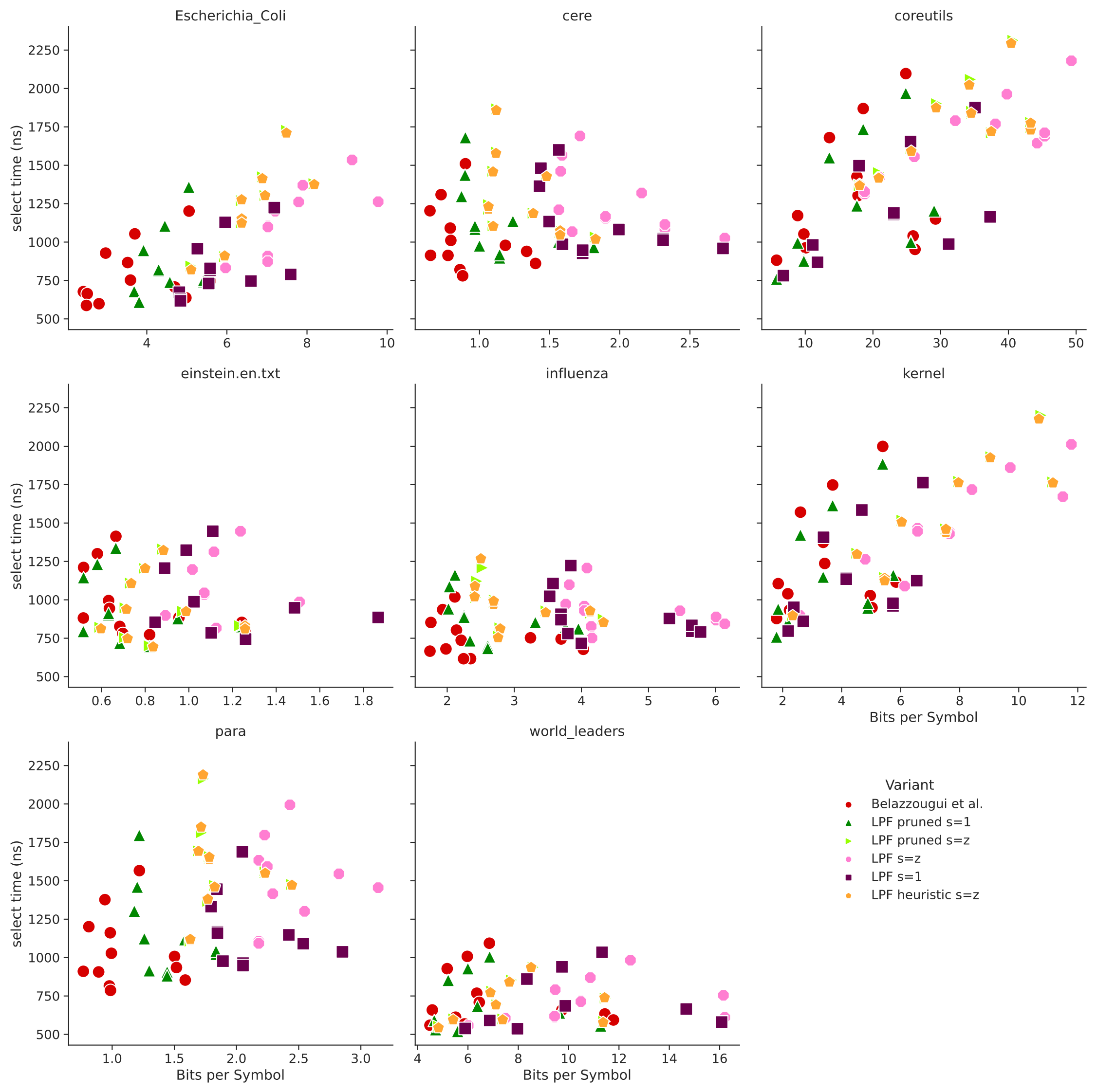


Figure 6.17: Average select time in relation to the block tree size for repetitive texts. The x-axis shows the block tree size in bits used per symbol input text, while the y-axis shows the average select time in nanoseconds on a linear scale. Note that each graph has an individual scale for the x-axis.

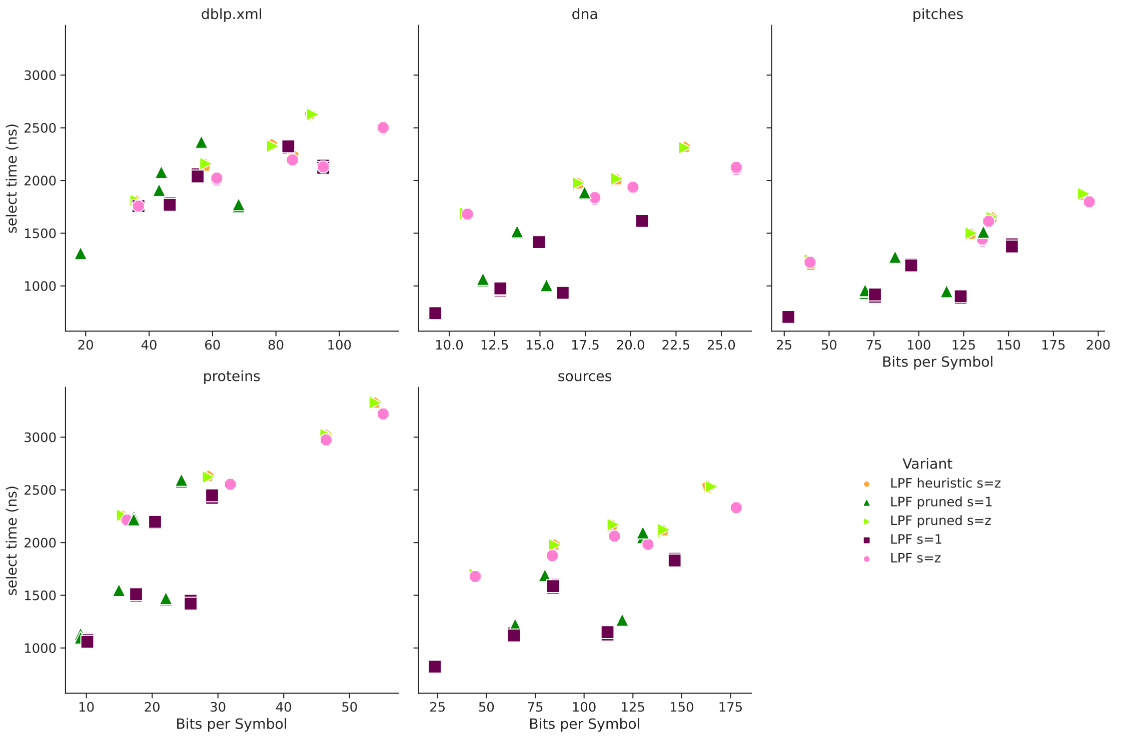


Figure 6.18: Average select time in relation to the block tree size for texts in the standard *Pizza&Chili* Corpus.

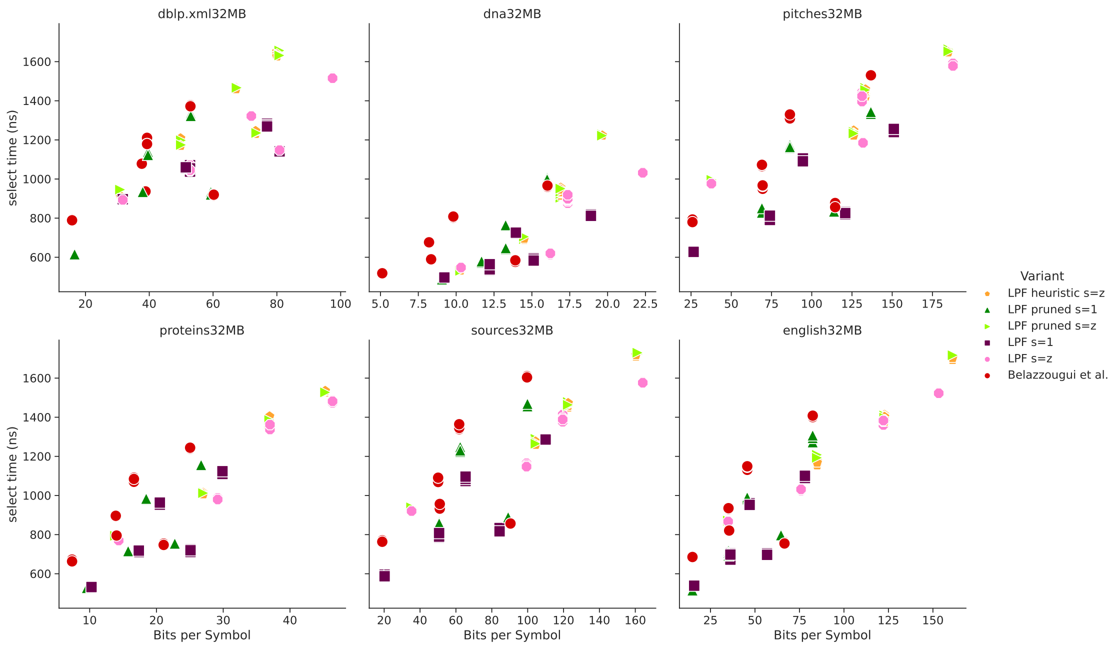


Figure 6.19: Average select time in relation to the block tree size for the 32 MiB prefixes for texts in standard *Pizza&Chili* Corpus.

6.3 Parallel Evaluation

We experimentally compare the performance of the naive sharded memory parallelization outlined in Section 5.3 with our sequential implementation. Due to unavailable hardware and an impending deadline, we can only provide the results for a single run over the highly repetitive *Pizza&Chilli* Corpus. Recall that our parallelization does not parallelize the actual block tree construction step, but instead relies on parallel LPF-array construction and running the σ depth first searches required for rank support. We believe these results are still useful, as they not only indicate the viability of block tree construction parallelization, but also that advances in sequential LPF-array construction could heavily impact the block tree construction time. For the actual block tree construction, we use *LPF pruned s=z* and the run benches for the following parameter configurations (τ, b): $(2,4), (4,4), (8,4)$. Setting b to 4 produced the smallest block trees when not considering rank support. We performed the experiment on the same 64-core machine (128 threads with simultaneous multithreading enabled) and 1024 TiB RAM described in Section 6.1.

The time to construct the LPF-array parallel *LPF*, the block tree without rank support *BT par* and the block tree with rank support *BT par rs* as a function of the number of processors for all texts in the repetitive *Pizza&Chilli* Corpus are shown in Figure 6.20. In addition, we plot performance of the sequential implementation tested in Section 6.2 as baselines *BT seq* and *BT seq rs*. We plot the average over all three configurations tested for each text and number of processors. Parallelizing *LPF* achieves good speed-up between 2 and 8 cores, but scales no further for higher clock counts. Naively parallelizing the σ depth first searches provides significant speed-up on texts with a large alphabet (e.g., *coreutils* with $\sigma = 236$) but shows barely any speed up on texts with small alphabets (e.g., *para, cere* with $\sigma = 5$). *LPF* achieves, on eight cores, a 3.6–4.3 times speed-up with respect to one core and on 64 cores (with simultaneous multithreading) a 4.4–4.9 times speed-up with respect to one core. Note that these results are worse than the results reported by Shun and Zhao [49], but both use different underlying suffix array implementations and algorithms. *BT par* achieves, on eight cores, a 2.2–3.7 times speed-up with respect to one core and on 64 cores (with simultaneous multithreading) a 2.4–4.75 times speed-up with respect to one core. Although, our current implementation does not scale, it shows that our current block tree construction time without rank support is heavily dominated by the LPF construction time. Therefore, to improve the sequential construction time in praxis, we suggest optimizing the current LPF construction method first. Finally, *BT par rs* achieves on eight cores, a 3.1–5.5 times speed-up with respect to one core and on 64 cores (with simultaneous multithreading) a 3.6–14.4 times speed-up with respect to one core. Showing that for this naive parallelization, the speed-up is heavily dependent on alphabet size.

In conclusion, this small initial experiment gives a few pointers towards next steps: (i) for our current approaches the construction is dominated by constructing the LPF-array and for larger alphabets adding the rank samples (for repetitive texts and running *BT par* on single core, constructing the LPF-array takes up $77\% - 94\%$ of construction time) and therefore optimizing these steps should take priority, (ii) as our current naive approach does not scale, the next step could be to integrate adding rank support to block tree into the block tree construction step itself. This would increase the work done during the actual construction step and therefore increases the potential speed-up of a parallelization attempt for the block tree construction. Although this come with the potential issue, that

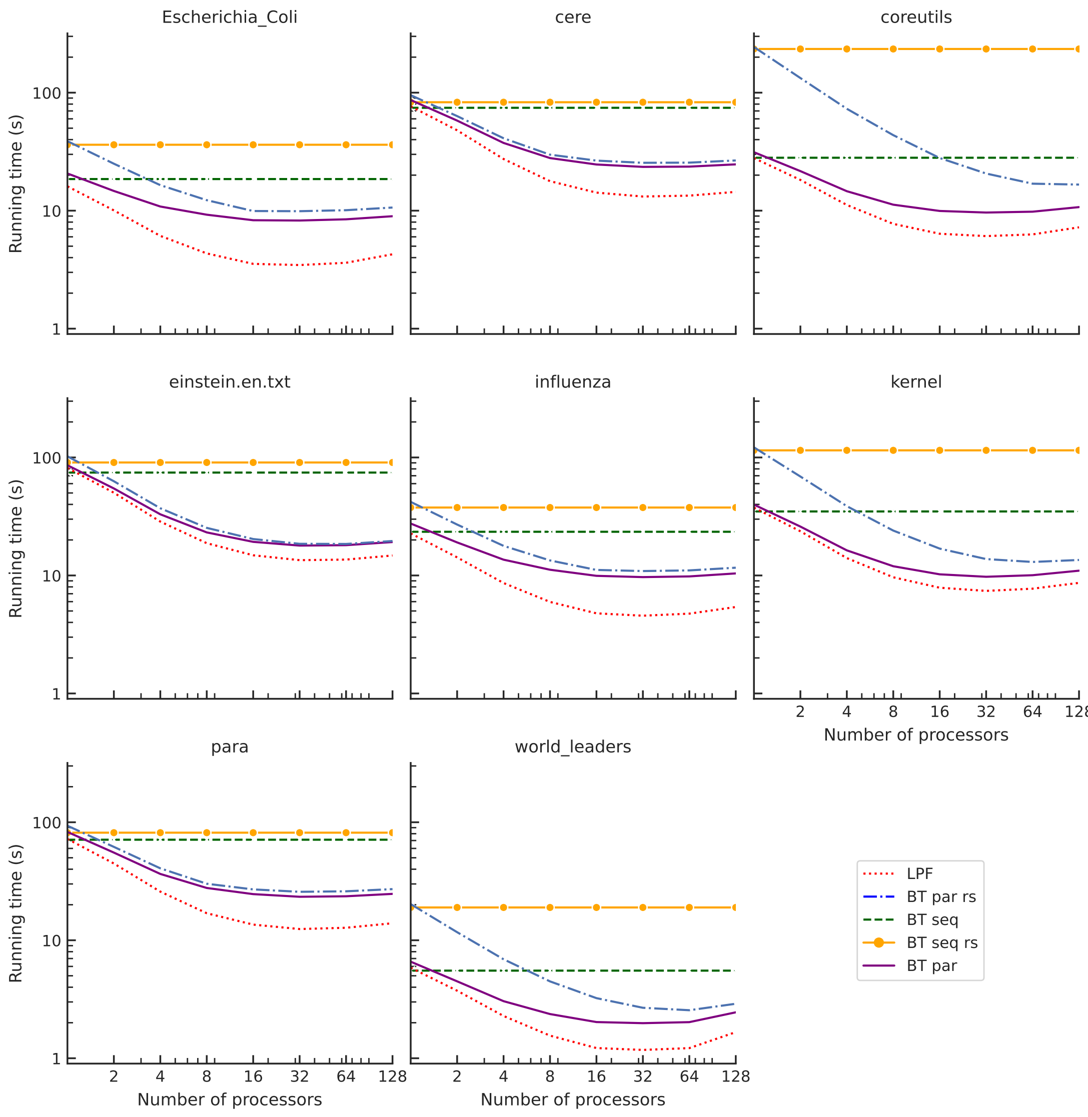


Figure 6.20: Log-log plots of constructing times on a 64-core machine (with simultaneous multithreading enabled).

we have to deal with larger temporary rank samples like the implementation by Belaz-zougui et al. [7].

7 Conclusion

Motivated by slow construction times in practice for the current implementation of block trees [7], we present a novel block tree construction algorithm, that relies on the longest previous factor array and previous occurrence array to identify leaves and find their leftmost occurrences. After constructing the longest previous factor array once, it can answer whether a block in a given level is a leaf in constant time without having to update its content regularly. This offers a significant benefit over the previous approaches, where it was necessary to rely on string matching techniques [1, 31] for every level individually. The previous occurrence array helps to find the leftmost occurrences of leaf blocks, but has to be updated after each level. On the other hand, our approach comes with a factor $\mathcal{O}(\log n)$ space penalty. Compared to the previous implementation, our implementation constructed block trees between 2.66 and 14.75 times faster on a highly repetitive text data set and between 8.19 and 23 times faster on a standard corpus. However, it should be noted that our implementation creates larger block trees for the same input parameters and due to a missing space optimization. We believe this optimization can be executed quickly and therefore should not affect run-time too much.

Our work introduces a series of improvements over the previous implementations of block tree. We implemented the block tree faithful to its theoretical proposal, while performing all construction steps, including the pruning step and the depth first searches necessary to add rank and select support on the compact bit vector based representation proposed by [7]. This eliminates cases, where for less repetitive texts with larger alphabets the temporary, uncompressed block tree became extremely large and would not fit into memory even if the final block tree is significantly smaller. We also implemented a greedy heuristic for the block tree, where we would replace a block with a leftwards pointer when ever possible, that behaved very similar in practice to our canonical block tree implementation and offered no construction time speed-up, while creating only slightly larger block trees.

Finally, we ran an initial experiment to parallelize block tree construction by using a known algorithm to parallelize the LPF-array construction in sharded memory and also added a naive parallelization to the σ depth first searches, used to add rank support, by running them in parallel. This parallelization provides a modest speed-up of 3.1–5.5 times with eight cores in respect to one core used, but only scales further for texts with a large alphabet (speed-up of 3.6–14.4 times with 64 cores in respect to one core used).

7.1 Future Work

Our block tree construction approaches based on the LPF-array decrease the practical construction time significantly when compared to the previous approaches, but come with a worse asymptotic space complexity of $\mathcal{O}(n \log n)$. LPF-arrays can be encoded in $2n+o(n)$

bits while supporting access to any position in $\mathcal{O}(\log n/\log\log n)$ time [44], but to the best of our knowledge, there is no linear time algorithm that constructs the encoded LPF-array in $\mathcal{O}(n)$ working space. Prezza and Rosone [44] present an online algorithm that constructs the encoded LPF-array in $\mathcal{O}(n\log^2 n)$ time and $n\mathcal{H}_k + o(n\log\sigma) + \mathcal{O}(n) + \sigma\log n + o(\sigma\log n)$ bits of working space. Furthermore, we require another integer array to store the updated first occurrences. Not updating these positions every level, but instead performing the naive scan outlined in Algorithm 8 showed no practical performance impact and even performed better for small values of s. Hence, it might be possible to amortize the number of "chains" taken during the construction process and present an algorithm, that does not update the first occurrence every level and still has the same runtime guarantees.

Our current approaches follow the general top-down construction method proposed by Belazzougui et al. [7] (Algorithm 5). We believe it might be possible to efficiently construct blocks trees bottom-up using the LPF-array. The first leaf in a block tree is always a last level leaf, as at least the first symbol of S has no previous occurrence in S. Thus, the next $\tau - 1$ sibling leaves are also last level leaves. Therefore, we can simply store them and jump to the next text position i after the last sibling, where we check if a block starting at i matches the condition to be a second to last level leaf. If this is the case, we ascend up the tree and repeat the check for the next higher level and so on. When we finally reach a level, where a block starting at i is a marked leaf, we back track and calculate the leftwards pointer. We move on to its next sibling and check if it is a leaf on the same level or if it is necessary to descend into a lower level. Note that for every descending step on the same text position, we took a previous ascending step that skips $\tau - 1$ potential leaves. Therefore, we believe that the approach would be efficient. Open questions are: How can we efficiently determine the leftwards pointers? How do we store the tree structure above the leaves? How can we determine the leftmost occurrences for marked blocks, which are required for the pruning step?

We suspect that the LPF-array might allow for another potential improvement. While marking blocks, we currently only mimic the conservative condition proposed by Belazzougui et al. [7], where initially the only unmarked blocks B_i are blocks, where both $B_{i+1} \cdot B_i$ and $B_i \cdot B_{i+1}$ have an earlier occurrence in S. Thus, requiring the pruning step as a post-processing space optimization. As the LPF-array stores the longest previous factor for all text positions, not just the starting positions of potential blocks. Therefore, it might be possible to formulate a stricter condition with range minima queries on the LPF-array, where we consider earlier occurrence for all substring of length $|B_i|$ in $B_{i-1} \cdot B_i \cdot B_{i+1}$ at once. Ideally, this makes the whole pruning step obsolete and is compatible with the previously mentioned bottom-up approach.

Bibliography

[1] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, jun 1975. ISSN 0001-0782. doi:10.1145/360825.360855.

[2] 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. ISSN 0306-4573. doi:https://doi.org/10.1016/j.ipm.2011.01.008. Large-Scale and Distributed Systems for Information Retrieval.

[3] 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 *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '12, page 255–264, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450314725. doi:10.1145/2348283.2348320.

[4] Jérémy Barbay and Gonzalo Navarro. On compressing permutations and adaptive sorting. *Theoretical Computer Science*, 513:109–123, 2013. ISSN 0304-3975. doi:https://doi.org/10.1016/j.tcs.2013.10.019.

[5] Jérémy Barbay, Johannes Fischer, and Gonzalo Navarro. Lrm-trees: Compressed indices, adaptive sorting, and compressed permutations. *Theoretical Computer Science*, 459:26–41, 2012. ISSN 0304-3975. doi:https://doi.org/10.1016/j.tcs.2012.08.010.

[6] Djamal Belazzougui and Gonzalo Navarro. Optimal lower and upper bounds for representing sequences. *ACM Trans. Algorithms*, 11(4), apr 2015. ISSN 1549-6325. doi:10.1145/2629339.

[7] Djamal Belazzougui, Manuel Cáceres, Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Gonzalo Navarro, Alberto Ordóñez, Simon J. Puglisi, and Yasuo Tabei. Block trees. *Journal of Computer and System Sciences*, 117:1–22, 2021. ISSN 0022-0000. doi:https://doi.org/10.1016/j.jcss.2020.11.002.

[8] O. Berkman, B. Schieber, and U. Vishkin. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *Journal of Algorithms*, 14 (3):344–370, 1993. ISSN 0196-6774. doi:https://doi.org/10.1006/jagm.1993.1018.

[9] M. Charikar, E. Lehman, Ding Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005. doi:10.1109/TIT.2005.850116.

[10] David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '96, page 383–391, USA, 1996. Society for Industrial and Applied Mathematics. ISBN 0898713668.

[11] David Richard Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, CAN, 1998. UMI Order No. GAXNQ-21335.

[12] Francisco Claude, Gonzalo Navarro, and Alberto Ordóñez. The wavelet matrix: An efficient wavelet tree for large alphabets. *Information Systems*, 47:15–32, 2015. ISSN 0306-4379. doi:https://doi.org/10.1016/j.is.2014.06.002.

[13] Maxime Crochemore and Lucian Ilie. Computing longest previous factor in linear time and applications. *Information Processing Letters*, 106:75–80, 04 2008. doi:10.1016/j.ipl.2007.10.006.

[14] Maxime Crochemore, Lucian Ilie, Costas S. Iliopoulos, Marcin Kubica, Wojciech Rytter, and Tomasz Waleń. Lpf computation revisited. In Jiří Fiala, Jan Kratochvíl, and Mirka Miller, editors, *Combinatorial Algorithms*, pages 158–169, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-10217-2.

[15] Robert M. Fano. *Transmission of information: A statistical theory of communications*. The M.I.T. Press, 1968.

[16] Martin Farach and S. Muthukrishnan. Optimal parallel dictionary matching and compression (extended abstract). In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, page 244–253, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897917170. doi:10.1145/215399.215451.

[17] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 390–398, 2000. doi:10.1109/SFCS.2000.892127.

[18] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 390–398, 2000. doi:10.1109/SFCS.2000.892127.

[19] Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011. doi:10.1137/090779759.

[20] František Franěk, Jan Holub, William F. Smyth, and Xiangdong Xiao. Computing quasi suffix arrays. *J. Autom. Lang. Comb.*, 8(4):593–606, jul 2003. ISSN 1430-189X.

[21] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully-functional suffix trees and optimal text searching in bwt-runs bounded space. *CoRR*, abs/1809.02792, 2018.

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

[23] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, page 841–850, USA, 2003. Society for Industrial and Applied Mathematics. ISBN 0898715385.

[24] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952. doi:10.1109/JRPROC.1952.273898.

[25] G. Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science*, pages 549–554, 1989. doi:10.1109/SFCS.1989.63533.

[26] Guy Joseph Jacobson. *Succinct Static Data Structures*. PhD thesis, Carnegie Mellon University, USA, 1988. AAI8918056.

[27] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison Wesley Longman Publishing Co., Inc., USA, 1992. ISBN 0201548569.

[28] Artur Jeż. A really simple approximation of smallest grammar. *Theoretical Computer Science*, 616:141–150, 2016. ISSN 0304-3975. doi:https://doi.org/10.1016/j.tcs.2015.12.032.

[29] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Automata, Languages and Programming*, pages 943–955, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-45061-0.

[30] Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. Permuted longest-common-prefix array. In Gregory Kucherov and Esko Ukkonen, editors, *Combinatorial Pattern Matching*, pages 181–192, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-02441-2.

[31] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987. doi:10.1147/rd.312.0249.

[32] J.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. doi:10.1109/18.841160.

[33] S.R. Kosaraju and G. Manzini. Compression of low entropy strings with lempel-ziv algorithms. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*, pages 107–121, 1997. doi:10.1109/SEQUEN.1997.666907.

[34] Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013. ISSN 0304-3975. doi:https://doi.org/10.1016/j.tcs.2012.02.006. Special Issue Combinatorial Pattern Matching 2011.

[35] Florian Kurpicz. Engineering compact data structures for rank and select queries on bit vectors. *CoRR*, abs/2206.01149, 2022. doi:10.48550/arXiv.2206.01149.

[36] A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976. doi:10.1109/TIT.1976.1055501.

[37] Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. *Nordic J. of Computing*, 12(1):40–66, mar 2005. ISSN 1236-6064.

[38] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. doi:10.1137/0222058.

[39] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001. doi:10.1137/S0097539799364092.

[40] Gonzalo Navarro. Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences. *ACM Comput. Surv.*, 46(4), mar 2014. ISSN 0360-0300. doi:10.1145/2535933.

[41] Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016. doi:10.1017/CBO9781316588284.

[42] Gonzalo Navarro. A self-index on block trees. In Gabriele Fici, Marinella Sciortino, and Rossano Venturini, editors, *String Processing and Information Retrieval*, pages 278–289, Cham, 2017. Springer International Publishing. ISBN 978-3-319-67428-5.

[43] Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Transactions on Computers*, 60:1471–1484, 2011.

[44] Nicola Prezza and Giovanna Rosone. Faster online computation of the succinct longest previous factor array. In Marcella Anselmo, Gianluca Della Vedova, Florin Manea, and Arno Pauly, editors, *Beyond the Horizon of Computability*, pages 339–352, Cham, 2020. Springer International Publishing. ISBN 978-3-030-51466-2.

[45] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding ik/i -ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):43, nov 2007. doi:10.1145/1290672.1290680.

[46] Michael Rodeh, Vaughan R. Pratt, and Shimon Even. Linear algorithm for data compression via string matching. *J. ACM*, 28(1):16–24, jan 1981. ISSN 0004-5411. doi:10.1145/322234.322237.

[47] Wojciech Rytter. Application of lempel–ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1):211–222, 2003. ISSN 0304-3975. doi:https://doi.org/10.1016/S0304-3975(02)00777-6.

[48] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948. doi:10.1002/j.1538-7305.1948.tb01338.x.

[49] Julian Shun and Fuyao Zhao. Practical parallel lempel-ziv factorization. In *2013 Data Compression Conference*, pages 123–132, 2013. doi:10.1109/DCC.2013.20.

[50] Zachary D. Stephens, Skylar Y. Lee, Faraz Faghri, Roy H. Campbell, Chengxiang Zhai, Miles J. Efron, Ravishankar Iyer, Michael C. Schatz, Saurabh Sinha, and Gene E. Robinson. Big data: Astronomical or genomical? *PLOS Biology*, 13(7): 1–11, 07 2015. doi:10.1371/journal.pbio.1002195.

[51] Nataliya Timoshevskaya and Wu-chun Feng. Sais-opt: On the characterization and optimization of the sa-is algorithm for suffix array construction. In *2014 IEEE 4th International Conference on Computational Advances in Bio and Medical Sciences (ICCABS)*, pages 1–6, 2014. doi:10.1109/ICCABS.2014.6863917.

[52] Dan Willard. Examining computational geometry, van emde boas trees, and hashing from the perspective of the fusion tree. *SIAM J. Comput.*, 29:1030–1049, 01 2000. doi:10.1137/S0097539797322425.

[53] Jing Yi Xie, Ge Nong, Bin Lao, and Wentao Xu. Scalable suffix sorting on a multicore machine. *IEEE Transactions on Computers*, 69(9):1364–1375, 2020. doi:10.1109/TC.2020.2972546.

[54] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977. doi:10.1109/TIT.1977.1055714.

List of Figures

List of Tables

List of Algorithms