Faster Block Tree Construction

Dominik Köppl
Department of Computer Science, University of Muenster, Germany

Florian Kurpicz
Karlsruhe Institute of Technology, Germany

Daniel Meyer
Karlsruhe Institute of Technology, Germany

Abstract

The block tree [Belazzougui et al. J. Comput. Syst. Sci. ’21] is a compressed text index that can answer access (extract a character at a position), rank (number of occurrences of a specified character in a prefix of the text), and select (size of smallest prefix such that a specified character has a specified rank) queries. It requires $O(z \log(n/z))$ words of space, where $z$ is the number of Lempel-Ziv factors of the text. For some highly repetitive inputs, a block tree can require as little as 0.015 bits per character of the text. Small values of $z$ make the block tree a space-efficient alternative to the wavelet tree, which is another index for these three types of queries. While wavelet trees can be constructed fast in practice, up so far compressed versions of the wavelet tree only leverage statistical compression, meaning that they are blind to spaced repetitions.

To make block trees usable in practice, a first step is to find ways in constructing them efficiently. We address this problem by presenting a practically efficient construction algorithm for block trees, which is up to an order of magnitude faster than previous implementations. Additionally, we parallelize our implementation, making it the first block tree construction implementation that works in parallel in shared memory.

2012 ACM Subject Classification Theory of computation → Data compression; Theory of computation → Pattern matching

Keywords and phrases compressed data structure, block tree, Lempel-Ziv compression, longest previous factor array, rank and select

Digital Object Identifier 10.4230/LIPIcs.ESA.2023.87

Supplementary Material Software (Source code): /github/pasta-toolbox/block_tree
Software (Comparison with competitors and raw data): /github/pasta-toolbox/block_tree_experiments

Funding This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No. 882500).

Dominik Köppl: Supported by JSPS KAKENHI Grant Numbers JP21K17701 and JP23H04378
Introduction

We experience an every-increasing amount of textual data produced in various domains. Examples include the exponentially increasing capability to sequence genetic data thanks to technical advances [64], code repositories such as GitHub, or natural text collections such as the English Wikipedia, which grows by around 2 million pages each year (currently there are over 58 million pages)\(^1\). Since there is no expectation that the production of such texts will decelerate, it seems that we start to drown in this sheer amount of data. Nevertheless, for the addressed examples, there is hope in that the produced textual data is usually highly repetitive: When sequencing two human individuals, we can expect to find that they share more than 99.9% of genetic data. In other domains such as code repositories or natural text collections, version control systems are used to track all versions of a document or source code to make it possible to revert changes or compare different versions. Since new versions often introduce only small changes, collections of all versions of the same document are often highly repetitive.

When stored or transmitted, texts are oftentimes compressed to save disk space or bandwidth, respectively. The most popular techniques for lossless text compression are based on the Lempel-Ziv 77 (LZ77) factorization [67]. Given \(z\) is the number of factors of the LZ77 factorization of a given text, we can represent the text in \(O(z)\) words of space. In many use cases, it does not suffice to only store or transmit textual data: the data also has to be processed. A naive way would be to decompress the data before processing it, which is, however, prohibitive for massive datasets. To avoid unnecessary decompression, we can use compressed text indices, which allow us to answer queries efficiently without decompression, while also guaranteeing us (asymptotically) the same space as the compressed text.

The block tree [6] is such a compressed text index that requires \(O(z \log(n/z))\) words of space for a text \(T\) of length \(n\) with \(z\) LZ77 factors. By default it can answer access queries. However, it can be augmented with additional information to also answer rank and select queries. The queries are defined as follows.

- **access**\((T, i)\) returns the character at position \(i\), i.e., \(T[i]\) for \(i \in [0, n)\),
- **rank**\(_\alpha\)(\(T, i\)) returns the number of occurrences of the character \(\alpha\) in the \(i\)-th prefix of the text, i.e., \(\text{rank}_\alpha(T, i) = |\{j \leq i : T[j] = \alpha\}|\) for \(\alpha \in \Sigma\) and \(i \in [0, n)\), and
- **select**\(_\alpha\)(\(T, i\)) returns the position of the first character \(\alpha\) that has rank \(i\), i.e., \(\text{select}_\alpha(T, i) = \min\{j : \text{rank}_\alpha(T, j) = i\}\) for \(\alpha \in \Sigma\) and \(i \leq \text{rank}_\alpha(T, n - 1)\).

One of the most popular data structures answering all three types of queries is the wavelet tree [35]. It is used in, among others, compressed full text indices based on the BWT [23, 31, 52] or on a grammar [15, 16], lossless data compression [22, 37, 42], and computational geometry [14]. For more related work, see Section 3. Using the block tree, all these queries can be answered in \(O(\log(n/z))\) time, with different space-time trade-offs available, see Section 4.

Our Contribution. In this paper, we present a block tree construction algorithm that leverages properties of the longest previous factor array, which is a common tool for computing the LZ77 factorization. We analyze our algorithm and show that it has the same asymptotic time complexity as previously presented construction algorithms. However, in our experimental evaluation, we observe that the implementation of our proposed algorithm is up to an order of magnitude faster than previous implementations. Finally, we show that our construction algorithm can also be parallelized.

2 Preliminaries

Let $T = T[0]T[1] \ldots T[n-1]$ be a text of length $n$ over an alphabet $\Sigma = [0, \sigma)$. The substring $T[i..j] = T[i] \ldots T[j]$ is called prefix if $i = 0$ and suffix if $j = n - 1$.

The Lempel-Ziv 77 (LZ77) factorization [67] parses the text into $z$ factors $f_0, \ldots, f_{z-1} \in \Sigma^*$ such that $T = f_0 \ldots f_{z-1}$. For all $i \in [0, z)$, $f_i$ is either a single character not occurring in $f_0, \ldots, f_{i-1}$ or the longest substring occurring at least twice in $f_0, \ldots, f_i$. The LZ77 factorization can be computed in linear time and space (see Ref. [2] for a survey).

The longest previous factor array $\text{LPF}$ stores at its $i$-th entry the length $\ell$ of the longest substring $T[i..i+\ell]$ having a previous occurrence in the text [17], i.e., $\text{LPF}[i] = \max\{\ell : T[i..i+\ell] = T[j..j+\ell] \text{ for } j < i\}$ for $i \in [0, n)$. In particular, if $i$ is the starting position of an LZ77 factor $f$, then $\text{LPF}[i] = |f|$, and thus we can compute the LZ77 factorization in linear time by scanning the $\text{LPF}$ array, which can be constructed in linear time [17]. Later on, we also need the position of the occurrence of a longest previous factor, which we store in the previous occurrence array $\text{PrevOcc}$. The previous occurrence array is also called quasi suffix array [27]. Here, for all $i \in [0, n)$ we have $T[i..i+\text{LPF}[i]] = T[\text{PrevOcc}[i]..\text{PrevOcc}[i]+\text{LPF}[i]]$ if $\text{LPF}[i] > 0$. We write $\text{PrevOcc}[i] = -1$ if $\text{LPF}[i] = 0$, i.e., when $T[i]$ is the leftmost occurrence of a single character in $T$.

3 Related Work

In this section, we give related work for compressed data structures answering our three types of queries (access, rank, and select) and work on block trees.

Access, Rank, and Select Data Structures

Answering queries such as access, rank, select are profound problems that have been well addressed in literature. Starting with the case for binary alphabets, there are plenty of results for indexing compressed [10,31,55,56,61] and uncompressed [34,44,48,54,58,65,66] bit vectors. A recent compressed approach involves the linear approximation of the distributions of parts of the ranks in the bit vector [10]. Despite that block trees also work on general alphabets, a block tree variant over the gapped compressed integer array of the ranks of the bit vector can be used to answer rank and select queries [24].

For larger alphabets, we are aware of statistically compressed data structures, where space is expressed in relation to the $k$-th order of empirical entropy $H_k$ with $k = o(\log \log n)$. Most prominent is the Huffman-shaped wavelet tree [35] using $nH_0(T) + o(nH_0(T) + 1) + O(\sigma \log n)$ bits, and solving all three queries in $O(\log \sigma)$ time. This time could be reduced to $O(1 + \log \sigma / \log \log n)$ with multiary wavelet trees [25], and by a later work [33], the space got reduced to $nH_0(T) + o(n \log n)$ bits. In practice, (Huffman-shaped) wavelet trees are also well-engineered in practice [13,18,19,20,21,26,28,46].

For faster queries on large alphabets, Golynski et al. [32] gave a data structure taking $n \log \sigma + o(n \log \sigma)$ bits that answers all three types of queries in $O(\log \log \sigma)$ time. The space got improved by Barbay et al. [4] to $nH_0(T) + o(nH_0(T) + 1)$ bits. Allowing slightly worse time complexities, Barbay et al. [5] achieved $nH_k(T) + o(n \log \sigma)$ bits, answering all queries in $o((\log \log \sigma)^{1+\epsilon})$ time, for any fixed constant $\epsilon > 0$. These time bounds were improved by Grossi et al. [36] to $O(\log \log \sigma)$ for rank and select, and constant time for access. Finally, Belazzougui and Navarro [9] presented a data structure achieving $nH_k(T) + o(n \log \sigma)$ bits of space, while answering rank in $O(\log \log \sigma)$ time. It further can answer access and select
in $O(1)$ and any time in $\omega(1)$, respectively, or the other way around. There are also several results on lower bounds for data structures answering the three queries we address here.

Another line of research is to augment grammar compression with an index to support our queries. Here, Belazzougui et al. [7] and Pereira et al. [57] presented grammar indices answering rank and select queries in $O(\log n)$ time. Their indices use $O(g \sigma \log n)$ bits of space when built on a grammar of size $g$, where the latter reference requires that the grammar is balanced. This requirement can be dropped in the light that a grammar can be made balanced in linear time [29].

**Block Trees**

Finally, we focus on block trees. Block trees have been proposed by Belazzougui et al. [8], who proposed a Monte Carlo construction algorithm using Karp-Rabin fingerprints in the external memory model. In the journal version [6], the authors provided two construction algorithms which we analyze in Section 4.1. Navarro [51, Section 4.2] recently surveyed block trees, who addresses also most of the references below for applications and variations.

A first application is pattern matching, where Navarro [50] uses block trees for locating pattern in the text. His index uses $O(z \log(n/z))$ words of space and finds all $occ$ occurrences of a pattern of length $m$ in $O(m^2 \log n + occ \log^\epsilon n)$ time for any constant $\epsilon > 0$. Brisaboa et al. [11] presented an extension of block trees to a two-dimensional data structure simulating $k^2$-trees. Recently, Cáceres and Navarro [12] applied block trees for the compression of the suffix tree topology, the suffix array, and its inverse.

Despite the fact that the space of block trees is related to the size of the LZ77 factorization, the space can, if we vary the definition of a block tree, be made related to the size $\gamma$ of a string attractor [39] or the substring complexity $\delta$ [62].

For the former (string attractor size $\gamma$), we recall that a string attractor is a set of positions of the text such that each substring has an occurrence in the text that contains a position of the string attractor. Kempa and Prezza [40, Theorem 5.3] gave a variant of block trees whose blocks cover substrings of consecutive string attractor positions and thus partition $T$ irregularly. Their variant uses $O(\gamma \log(n/\gamma)) = O(z \log(n/z))$ space and extracts a length-$\ell$ substring in $O(\log(n/\gamma) \cdot (1 + \ell)/\log_\sigma n)$ time. For indexing, Prezza [59] (for rank and select queries) and Navarro and Prezza [53] (for pattern matching) could revert the property that blocks on the same level have equal length while retaining the space size.

For the latter (substring complexity $\delta$), let $\delta := \max\{d_k/k : k \in [1,n]\}$ denote the substring complexity of $T$, where $d_k$ is the number of distinct length-$k$ substrings of $T$. Then there is a block tree variant that can be represented in $O(\delta \tau \log_\tau n)$ space supporting queries in $O(\log \frac{n}{\delta})$ time [41].

4 Block Trees

Let $T$ be a text of length $n$ over an alphabet of size $\sigma$, whose LZ77 factorization consists of $z$ factors. The block tree [6] is a compressed index requiring $O(z \log(n/z))$ words of space. It supports access, rank, and select queries in $O(\log(n/z))$ time. In the following, we describe a block tree with two integer parameters $s$ and $\tau$ that are greater than 1, which specify the out-degree of the root and all other internal nodes, respectively. For simplicity, we assume that $n = s \cdot \tau^h$ for some integer $h$. Now, the block tree is a tree of height $h = 1 + \log_\sigma \frac{s \log n}{\tau \sigma}$ with parameters $\tau$ and $s$ such that the root has $s$ children and every internal node is a leaf or has $\tau$ children.
Figure 1 The block tree (left) and its pruned version (right) for the text $T = \text{AAAABBAAABABBAA}$ with $\tau = 2$ and $s = 4$. Dashed arrows indicate pointers to the leftmost occurrence of the block the arrow starts at. The red line indicates the offset stored in addition to the pointer. Note that only characters in leaves are stored explicitly. For simplicity, our leaves contain only a single character. In the pruned block tree, we can replace an additional AA block. Note that we cannot prune the second AA block, as another block points into it.

Each node $u$ represents a substring of $T$ called block $B^u$. The root represents the whole text $T$ and has $s$ children representing $s$ consecutive blocks of length $n/s$. We refer to all blocks with the same depth as a block tree level. Two blocks are consecutive if they are in the same block tree level and if they are consecutive in $T$. Let $B_i \cdot B_{i+1}$ be the concatenated substring of two consecutive blocks. We mark the blocks $i$ and $i + 1$, if $B_i \cdot B_{i+1}$ is the leftmost occurrence of that substring in $T$. All non-root nodes that are not in the last level represent either marked or unmarked blocks.

All marked blocks $B^v$ are internal nodes with $\tau$ children. These children represent consecutive blocks of length $|B^v|/\tau$ whose concatenation is $B^v$. Unmarked blocks $B^u$, on the other hand, are leaves that only 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 the blocks.

The block tree requires $O(s + \tau \log \tau n \log n)$ words of space. Choosing $\tau$ as constant yields the minimum space requirement of the block tree mentioned above. Choosing $s = z$ results in block trees of size $O(\tau \log \tau n \log n)$ words. Different values for $\tau$ can introduce other space-time trade-offs, as described by Belazzougui et al. [6].

4.1 Construction

Belazzougui et al. [6] give two construction algorithms, which we briefly review. Their first algorithm requires $O(n)$ words of working space, where the idea is to use an Aho-Corasick automaton [1] that can identify all consecutive pairs of blocks $B_0 \cdot B_1, B_1 \cdot B_2, \ldots, B_{s-2} \cdot B_{s-1}$ on the first level. This automaton is then used to identify the first occurrences of all pairs and to mark them accordingly. To set the leftwards pointers into unmarked blocks, the automaton is replaced by a new automaton that recognizes all unmarked blocks is created. The text is traversed using this automaton. Whenever an unmarked block is found for the first time, a pointer (and offset) is stored. For then on, the second automaton is no
Faster Block Tree Construction

longer of use and can be removed. Subsequently, the algorithm continues with the next level, considering only the unmarked blocks from the previous level.

Their second algorithm uses $O(s + z\tau)$ words of working space and runs in $O(n)$ expected time. Here, the general idea is to replace the Aho-Corasick automaton with Karp-Rabin fingerprints [38], i.e., storing Karp-Rabin fingerprints of all consecutive pairs of blocks $B_i \cdot B_{i+1}$ in a hash table. Since there are at most $3\tau$ blocks per level, this approach requires only $O(s + z\tau)$ words of working space. Both algorithms work in linear time if $s = \Theta(z)$.

The block tree as described here only supports access queries. For rank and select support, additional information has to be stored for each marked block. In the case of rank queries, the occurrence of all characters in the text up to the beginning of the block is necessary. For more details, we refer to the original block tree paper [6].

Pruning. When we construct block trees with one of the two aforementioned construction algorithms, we meet the asymptotic space bounds—but the block tree may contain more blocks than necessary, see Figure 1. Remember that we mark the first occurrence of each pair of consecutive blocks $B_i \cdot B_{i+1}$ to guarantee that any block $B^u$ to their right can point to them. However, there may be no rightwards block pointing to either (or both) $B_i, B_{i+1}$. In this case we would replace one of the blocks (or both) with leftward pointers, if one (or both) of them occurs previously. Since we do not modify this part of the algorithm, we refer to Belazzougui et al.’s [6, Section 6.1] description of the pruning step for more details.

5 Block Tree Construction using the LPF Array

We now describe our new block tree construction algorithm based on the LPF array. First, in Section 5.1, we mark blocks using only the LPF array. Then, in Section 5.2, we find the leftmost occurrences of unmarked blocks, before, in Section 5.3, we combine all these ideas to our new algorithm.

5.1 Marking Blocks

The block tree is closely related to the LZ77 factorization of the text. Consecutive blocks $B_{i-1}, B_i, \text{ and } B_{i+1}$ are only marked if an LZ77 factor starts in $B_i$, i.e., if they contain the leftmost occurrence of some substring. Similarly, LPF values witness the shortest substring starting at each text position that is a leftmost occurrence. Hence, we can make use of the LPF array to mark blocks. Let $s(B^u)$ denote the starting text position of the substring represented by $B^u$.

Lemma 2. Given three consecutive blocks $B_{i-1}, B_i, \text{ and } B_{i+1}$ of length $\ell$. We mark $B_i$ if $\text{LPF}[s(B_{i-1})] < 2 \cdot \ell$ or $\text{LPF}[s(B_i)] < 2 \cdot \ell$ is true.

Proof. By the definition of the LPF array, a substring $T[i..i + \ell]$ has a preceding occurrence in the text if $\text{LPF}[i] \geq \ell$. We only leave $B_i$ unmarked if both $\text{LPF}[s(B_{i-1})]$ and $\text{LPF}[s(B_i)]$ are at least $2 \cdot |B_i|$ because this means that there is a previous occurrence of both $B_{i-1} \cdot B_i, \text{ and } B_i \cdot B_{i+1}$. Otherwise, if there is no previous occurrence of one of the two pairs, we have to mark $B_i$.

---

2 The description is based on and has text overlaps with Daniel Meyer’s Master’s thesis [49].
To determine whether the last block is marked, only its preceding block has to be considered. Otherwise, it is the same argument as used in Lemma 2. Since each level of the block tree contains \(O(\tau r)\) blocks, we get the following result.

**Lemma 3.** Given the LPF array, we can mark all blocks of a level in the block tree in \(O(\tau r)\) time.

### 5.2 Identifying Leftmost Occurrences

Now, for each unmarked block, we have to identify the leftmost substring in the text that is equal to that block, as we need to add pointers (and offsets) from the unmarked blocks to these occurrences. Note that in all levels but the first one, the index of the block \(B_u\) does not automatically translate to the block’s starting position \(\mathbf{a}(B_u)\), as we do not know how many blocks have been unmarked to its parent’s left in the previous level. Therefore, we need to store additional information regarding a block’s starting position for each block.

#### 5.2.1 Leftmost Occurrences as Text Positions

In this section, we describe how to identify the text position of the previous occurrence. Afterwards, in Section 5.2.2, we show how to identify the block that contains this text position on the current level. While the LPF array is sufficient to mark blocks, it does not contain information necessary to find the leftmost occurrences of blocks that we require for the leftward pointers. We now give a naive approach to compute the text positions in Section 5.2.1.1. Then, in Section 5.2.1.2, we improve the naive approach by using dynamic programming to obtain better asymptotic running times. The general idea in both cases is to follow the leftmost occurrences of previous occurrences for all blocks on a level.

**Lemma 4.** Let \(i \in [0, n)\) be a text position, \(j = \text{PrevOcc}[i]\), and \(k = \text{PrevOcc}[j]\). If \(0 < \text{LPF}[i] \leq \text{LPF}[j]\), then \(T[i..i + \text{LPF}[i]] = T[j..j + \text{LPF}[i]] = T[k..k + \text{LPF}[i]]\).

**Proof.** \(\text{LPF}[i] = \max\{k: T[i..i + k] = T[j..j + k]\) for \(j < i\}\) and \(\text{PrevOcc}[i]\) gives us the position \(j\), where this longest factor occurs. Since \(0 < \text{LPF}[i]\), we have \(T[i..i + \text{LPF}[i]] = T[j..j + \text{LPF}[i]]\) by definition. We also know that for text position \(j\), there exists a previous factor of length at least \(\text{LPF}[i]\) at position \(k\). Hence \(T[i..i + \text{LPF}[i]] = T[k..k + \text{LPF}[i]]\). \(\triangleleft\)

The same holds not only for length-LPF\([i]\) substrings, but for general length-\(\ell\) substrings with \(\ell \leq \text{LPF}[i]\), which is useful when processing a block tree level where blocks have all the same length \(\ell\).

**Observation 5.** Let \(i \in [0, n)\) be a text position and \(j = \text{PrevOcc}[i]\). If \(0 < \ell \leq \text{LPF}[i]\), then \(T[i..i + \ell] = T[j..j + \ell]\).

#### 5.2.1.1 Naive Approach

Using these properties, we can describe a naive algorithm to find the leftmost occurrence of a given unmarked block \(B_u\) = \(T[i..i + \ell]\). Here, we simply follow the \(\text{PrevOcc}\) entries until the length of the longest previous factor of the previous occurrence is smaller than \(\ell\). This leads us to the first occurrence of length \(\ell\). Unfortunately, this naive approach requires \(O(n)\) time for each block. For example, in an all-\(a\) text \(aaa\ldots a\), for text position \(i\) we would follow the longest previous occurrences \(i - 1, i - 2, \ldots, 0\) in case that \(\text{PrevOcc}(i) = i - 1\) for all \(i > 0\). Therefore, using the naive approach, we do not achieve the asymptotic running time of the original block tree construction algorithms by Belazzougui et al. [6]. However, in practice, this approach works very well, as we observed in our experimental evaluation in Section 6.
to compute \( j \) at \( \text{FirstOcc} \). We can now use this property to identify the leftmost occurrence for each block level-by-level in \( \text{FirstOcc} \), where we store \( \text{FirstOcc}_{\ell} \) for the current level and update it for each following level. Recall that by definition, each pair of marked blocks contains the leftmost occurrence of a block in the text. We use of dynamic programming to find the leftmost occurrences of a block in the text. We start with the definitions of \( \text{LPF}_\ell \) and \( \text{FirstOcc}_{\ell} \).

**Definition 6 (\( \ell \)-factor and \( \text{FirstOcc}_\ell \)).** For \( \ell > 0 \), we denote \( T[i..i+\ell] \) as \( \ell \)-factor, \( \text{FirstOcc}_{\ell}[i] \) stores \( \text{PrevOcc}[i] \), if no previous occurrence of \( \ell \)-factor exists (remember that \( \text{PrevOcc}[i] = -1 \) if \( T[i] \) is the leftmost occurrences of a character, i.e., \( \text{LPF}[i] = 0 \)) and the leftmost occurrence of \( \ell \)-factor, otherwise.

We can compute \( \text{FirstOcc}_\ell \) using dynamic programming by iterating over \( \text{PrevOcc} \) from left to right. To start with, we set \( \text{FirstOcc}_\ell[0] = -1 \). Suppose that we have processed \( \text{FirstOcc}_{\ell}[0..i-1] \) and are at text position \( i \in [0, n - \ell] \). For \( j := \text{PrevOcc}[i] \), we consider two cases:

**Case 1** \( \text{LPF}[i] \geq \ell \) and \( \text{LPF}[j] \geq \ell \): From \( \text{LPF}[j] \geq \ell \) follows that \( \ell \)-factor has a previous occurrence. Combined with Observation 5 and \( \text{LPF}[i] \geq \ell \), we can conclude that the previous occurrence of \( \ell \)-factor, is also an occurrence of \( \ell \)-factor. As we already calculated \( \text{FirstOcc}_{\ell}[j] \), we can set \( \text{FirstOcc}_{\ell}[i] = \text{FirstOcc}_{\ell}[j] \).

**Case 2** \( \text{LPF}[i] < \ell \) or \( \text{LPF}[j] < \ell \). We set \( \text{FirstOcc}_{\ell}[i] = j \) since either \( T[i..i+\ell] \) or \( T[j..j+\ell] \) is the leftmost occurrence of \( \ell \)-factor.

See Table 1 for an example. Note that we still need the \( \text{LPF} \) array to correctly interpret any \( \text{FirstOcc}_{\ell}[i] \). For \( \text{LPF}[i] \geq \ell \), we know that \( \ell \)-factor has an earlier occurrence at \( j \) but no occurrence further left. This dynamic programming approach requires \( O(n) \) time to compute \( \text{FirstOcc}_{\ell} \). Such time is unfeasible if we need to calculate \( \text{FirstOcc}_{\ell} \) for each level of the block tree.

However, it is possible to compute \( \text{FirstOcc}_{\ell_0} \) using \( \text{FirstOcc}_{\ell_1} \) for \( \ell_1 \geq \ell_0 \), as every occurrence of an \( \ell_1 \)-factor contains an \( \ell_0 \)-factor as a prefix. There might be an occurrence of the \( \ell_0 \)-factor further left, but we store information about that in \( \text{FirstOcc}_{\ell_1} \). Remember that we also store pointers to a previous occurrence of the longest previous factor if that factor is shorter than \( \ell_1 \).

We can now use this property to identify the leftmost occurrence for each block level-by-level in \( \text{FirstOcc}_{\ell} \), where we store \( \text{FirstOcc}_{\ell} \) for the current level and update it for each following level. Recall that by definition, each pair of marked blocks contains the leftmost occurrence of a block in the text. We make use of dynamic programming to find the leftmost occurrences of a block in the text. We start with the definitions of \( \ell \)-factors and \( \text{FirstOcc}_{\ell} \).
occurrence of at least one substring of \( T \). Therefore, each leftmost occurrence of any substring with length at most the current block level length \( \ell \) is contained in a marked block. All blocks in the current level (except for the first level) are children of marked blocks in the level before. Hence, the leftmost occurrence of each substring of a length equal to the current block length is contained in a block in the current level. We still have to update all text positions contained in the previous block tree level. This is necessary as we need to consider cases where the values in the \( \text{LPF} \) array are smaller than the last level’s block size \( \ell_1 \) but greater than the next level’s block size \( \ell_0 \), i.e., the positions \( i \in [0, n) \) where \( \ell_0 \leq \text{LPF}[i] < \ell_1 \).

In this case \( \text{FirstOcc}_{\ell_1}[i] \) points to a previous occurrence of the longest previous factor of \( i \). This occurrence can be in an unmarked block as it is not necessarily the first occurrence of said longest previous factor. Hence, we also have to update \( \text{FirstOcc} \) for text positions \( k \in [0, n) \) that fall into an unmarked block in the previous level. We do so from left to right. Suppose we have updated \( \text{FirstOcc}_{\ell_0}[0..k-1] \) and are processing \( \text{FirstOcc}_{\ell_0}[k] \). Let \( p = \text{FirstOcc}_{\ell_0}[k] \). We will update \( \text{FirstOcc}_{\ell_0}[k] \) if one of the two conditions is met.

**Condition 1** \( \text{LPF}[k] \geq \ell_0 \) and \( \text{LPF}[p] \geq \ell_0 \): We know that \( p \) and \( k \) share the same \( \ell_0 \)-factor (Observation 5). Therefore, the first occurrence of \( \ell_0 \)-factor \( p \), is also the first occurrence of \( \ell_0 \)-factor \( k \), and we can set \( \text{FirstOcc}_{\ell_0}[k] = \text{FirstOcc}_{\ell_0}[p] \).

**Condition 2** \( 0 < \text{LPF}[k] \leq \text{LPF}[p] \): If condition 2 is met but condition 1 is not, we still know that \( \text{FirstOcc}_{\ell_0}[k] = \text{FirstOcc}_{\ell_0}[p] \), as there are just two cases:

**Case 2.1** \( \text{LPF}[k] < \ell_0 \) and \( \text{LPF}[p] \geq \ell_0 \): Due to condition 2 and Lemma 4 and Observation 5 we know that said longest previous factor is a prefix of \( \ell_0 \)-factor \( p \). Due to \( \text{LPF}[p] \geq \ell_0 \), \( \text{FirstOcc}_{\ell_0}[p] \) points to the leftmost occurrence of \( \ell_0 \)-factor \( p \).

**Case 2.2** \( \text{LPF}[k] < \ell_0 \) and \( \text{LPF}[p] < \ell_0 \): Due to condition 2 and Lemma 4 we know that said longest previous factor has a previous occurrence at \( \text{FirstOcc}_{\ell_0}[p] \).

If neither condition is met, i.e., \( \text{LPF}[p] < \text{LPF}[k] \) and \( \text{LPF}[p] < \ell_0 \), \( k \) is either the leftmost occurrence for all factors of size at least \( \ell_0 \) or \( \text{FirstOcc}_{\ell_0}[k] \) points to a first occurrence of a substring smaller than \( \ell_0 \) and hence points into a marked block.

**Lemma 7.** Updting \( \text{FirstOcc} \), for all levels during the block tree construction requires \( O(n(1 + \log \tau \frac{z}{\tau})) \) time in total.

**Proof.** Initializing \( \text{FirstOcc} \), takes \( O(n) \) time. Every level but the first has at most \( 3\tau \tau \) blocks, and the size of blocks is decreasing by a factor \( \tau \) for each further level. This reduces the number of string positions still contained inside of blocks geometrically with each level. The total sum of all block lengths is \( O(n(1 + \log \tau \frac{z}{\tau})) \) [6, Section 6.1]. Since we only have to update \( \text{FirstOcc} \), for positions in unmarked blocks in the previous level, we obtain the required total time for all levels.

### 5.2.2 Leftmost Occurrences as Blocks

After updating \( \text{FirstOcc} \), to store the leftmost occurrence for each text position in the current block tree level \( B_0, B_1, \ldots \), we still have to find the marked blocks covering the leftmost occurrence of each unmarked block \( B^u \).

We can map the occurrences in the text to blocks in three parts. First, we store for each unmarked block \( B^u \) a pair \( \langle \text{FirstOcc}_{\ell_0}[s(B^u)], i \rangle \) containing its leftmost occurrence and its index in our block level in a set \( U \). Second, we sort the set by each pair’s first element using a radix sort, i.e., we sort our unmarked blocks by their leftmost occurrences in the text. Third, we sequentially scan our block tree level and our sorted set \( U \) simultaneously in the following fashion. Let \( \langle \text{occ}_j, j \rangle \) be the currently considered element in \( U \) and \( B^u \) be the
current block of our block tree level. If \( \ell \)-factor \( \text{occ}_j \) starts inside \( B_i \), we set a pointer from \( B_j \) to \( B_i \) with offset \( \text{occ}_j - s(B_i) \) and continue with the next element in \( U \). Otherwise, we continue with the next block in our block tree level.

▶ **Lemma 8.** For block trees with \( z\tau = O(n) \), finding the blocks containing the leftmost occurrences of unmarked blocks can be done in \( O(z\tau) \) time and \( O(z\tau) \) words of space.

**Proof.** The first and third step require \( O(z\tau) \) time, as we only scan over blocks in the current block tree level. Sorting \( U \) with radix sort requires \( O(z\tau) \) time and \( O(z\tau) \) words of space. ▶

Alternatively, we can also identify the mapping between text positions and blocks by traversing the already built block tree. This mimics an access query on the block tree that stops as soon as it reaches the current level. Such a query requires \( O(\log \tau n \log \sigma s \log n) \) time, which is linear in the height of the tree. Overall, computing the mapping using this approach requires \( O(\tau \log \tau n \log n \sigma) \) time.

5.3 New Block Tree Construction Algorithm

Now, we can put all these building blocks together to form a practically efficient block tree construction algorithm. First, we calculate the LPF array and \( \text{PrevOcc} \) array. We then initialize \( \text{FirstOcc}^\ast \) for \( \ell = n/s \) and construct each block tree level top-down as described in this section, i.e., we identify all marked blocks and then compute all pointers (and offsets) for the unmarked blocks. Finally, we update \( \text{FirstOcc} \), and repeat this process until we reach the deepest level, where the blocks are stored explicitly. While this algorithm does not introduce better asymptotic running times, it practically outpaces all other available construction implementations, as we will empirically evaluate in Section 6. Overall, combining all previous steps, we obtain the following result.

▶ **Theorem 9.** Given a string \( T \) of length \( n \) over an alphabet of size \( \sigma \), two integers \( s \) and \( \tau \) greater than 1, we can compute the block tree in \( O(n(1 + \log \tau z s)) \) time using \( O(n) \) words of space.

Pruning. The algorithm described in this section constructs the same block tree structure as the algorithms described by Belazzougui et al. [6], see Section 4.1. Thus, their pruning algorithm works without any changes of the block tree resulting from this construction.

6 Experimental Evaluation

We conducted our experiments on a server equipped with an AMD EPYC Rome 7702P (64 cores (128 hyperthreads), frequencies up to 3.35 GHz, and 256 MiB L3 cache) and 1024 GiB DDR4 ECC RAM. The server runs Ubuntu 20.04.2 LTS. We compiled all code with GCC 12.1 using the flags -03 and -march=native. For the evaluation of our parallel code written in OpenMP, we compiled the code with the additional flag -fopenmp.

We compare our block tree construction algorithms [43,45] with the (to our best knowledge) only other block tree implementation by Belazzougui et al. [6]. Their implementation uses Karp-Rabin fingerprints (Section 4) with parameter \( s = 1 \). While \( s = 1 \) is not a feasible choice in the formal definition of block trees (see Section 4), in practice, all levels without any unmarked blocks are removed during the pruning phase, making this configuration possible.

---

3 See https://github.com/elarielcl/BlockTrees, last accessed 2023-07-04
Table 2: Input names, number of characters $n$, alphabet size $\sigma$, number of LZ77 factors $z$, measure of compressibility $\frac{z \log n}{n \log \sigma}$, and the compression achieved with p7zip (v. 16.02) expressed by the ratio of the compressed output size divided by the input size.

<table>
<thead>
<tr>
<th>Input</th>
<th>$n$</th>
<th>$\sigma$</th>
<th>$z$</th>
<th>$\frac{z \log n}{n \log \sigma}$</th>
<th>p7zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>repetitive</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cere</td>
<td>461 286 644</td>
<td>5</td>
<td>1 700 630</td>
<td>0.044</td>
<td>5.35%</td>
</tr>
<tr>
<td>coreutils</td>
<td>205 281 778</td>
<td>236</td>
<td>793 915</td>
<td>0.013</td>
<td>11.75%</td>
</tr>
<tr>
<td>einstein.en</td>
<td>467 626 544</td>
<td>139</td>
<td>89 467</td>
<td>0.0007</td>
<td>0.10%</td>
</tr>
<tr>
<td>Escherichia_coli</td>
<td>112 689 515</td>
<td>15</td>
<td>2 078 512</td>
<td>0.121</td>
<td>7.76%</td>
</tr>
<tr>
<td>influenza</td>
<td>154 808 555</td>
<td>15</td>
<td>7 692 86</td>
<td>0.033</td>
<td>1.69%</td>
</tr>
<tr>
<td>kernel</td>
<td>257 961 616</td>
<td>160</td>
<td>1 446 468</td>
<td>0.021</td>
<td>2.53%</td>
</tr>
<tr>
<td>para</td>
<td>429 265 758</td>
<td>5</td>
<td>2 332 657</td>
<td>0.064</td>
<td>6.05%</td>
</tr>
<tr>
<td>world_leaders</td>
<td>46 968 181</td>
<td>89</td>
<td>175 740</td>
<td>0.014</td>
<td>1.39%</td>
</tr>
<tr>
<td>non-repetitive</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dblp.xml</td>
<td>296 135 874</td>
<td>97</td>
<td>9 576 081</td>
<td>0.138</td>
<td>12.74%</td>
</tr>
<tr>
<td>dna</td>
<td>403 927 746</td>
<td>16</td>
<td>25 628 189</td>
<td>0.453</td>
<td>22.79%</td>
</tr>
<tr>
<td>english</td>
<td>1 610 612 736</td>
<td>239</td>
<td>97 047 354</td>
<td>0.233</td>
<td>26.11%</td>
</tr>
<tr>
<td>pitches</td>
<td>55 832 855</td>
<td>133</td>
<td>5 994 276</td>
<td>0.391</td>
<td>25.89%</td>
</tr>
<tr>
<td>proteins</td>
<td>1 184 051 855</td>
<td>27</td>
<td>80 408 252</td>
<td>0.430</td>
<td>31.30%</td>
</tr>
<tr>
<td>sources</td>
<td>210 866 607</td>
<td>230</td>
<td>11 598 459</td>
<td>0.194</td>
<td>15.84%</td>
</tr>
</tbody>
</table>

There are two different variants of our block tree construction algorithm: LPF$^\text{DP}$ and LPF$_s$. The former uses the dynamic programming approach to identify the text position of the previous occurrence while the latter uses the naive approach, see Section 5.2. Since we need the LPF array for our construction algorithms, we can compute $z$ and also choose both $s = z$ and $s = 1$. To show that our improvements are not only based on engineering, we also include FP$_1$ which uses fingerprints instead of the LPF array while the rest of our code remains nearly unchanged, i.e., it is a reimplementation of the original algorithm.

For our implementation, we make use of libsais$^4$, the fastest suffix array construction algorithm implementation available to compute the suffix and longest common prefix arrays that we use for the LPF array construction. We also use the int_vector from the Succinct Data Structure Library [30] and the pasta::bit_vector [44] internally in the block tree.

We do not include fast wavelet tree construction algorithms [18,30] in our plots as preliminary experiments show that they can be constructed at least an order of magnitude faster than block trees. For a detailed comparison of query speed of block trees and wavelet trees, we refer to the block tree article by Belazzougui et al. [6]. They show that block trees require around the same space but can answer queries an order of magnitude faster.

We conducted our sequential experiments with all combinations of $\tau \in \{2, 4, 8, 16\}$ and maximum leaf size $b = \{2, 4, 8, 16\}$, i.e., the threshold on the number of characters stored explicitly as a leaf. The timing starts when the input is loaded in main memory and stops as soon as the block tree has been constructed. All reported values are the average of three runs. We used the repetitive text corpus from the Pizza&Chili corpus$^5$, which was also used in the original block tree article [6]. Additionally, we used the non-repetitive Pizza&Chili corpus$^6$. See Table 2 for details.

---

4 See https://github.com/IlyaGrebnov/libsais, last accessed 2023-07-04.
Table 3 Most space-efficient ($\tau_{\text{space}}$ and $b_{\text{space}}$) and fastest configurations ($\tau_{\text{time}}$ and $b_{\text{time}}$) of LPF$_1$ on the repetitive inputs (without rank and select support).

<table>
<thead>
<tr>
<th>Input</th>
<th>$\tau_{\text{space}}$</th>
<th>$b_{\text{space}}$</th>
<th>$\tau_{\text{time}}$</th>
<th>$b_{\text{time}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>cere</td>
<td>16</td>
<td>16</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>coreutils</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>einstein.en.txt</td>
<td>4</td>
<td>16</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Escherichia_Coli</td>
<td>4</td>
<td>16</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>influenza</td>
<td>4</td>
<td>16</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>kernel</td>
<td>8</td>
<td>16</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>para</td>
<td>4</td>
<td>16</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>world_leaders</td>
<td>4</td>
<td>16</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

6.1 Sequential Block Tree Construction

In this section, we present the results of our experimental evaluation of block tree construction algorithms. For the evaluation, we used both, repetitive and non-repetitive inputs.

Repetitive Inputs. In Figures 2 and 3, we show the construction throughput (processed input in MiB per second) on the $y$-axis and the space requirements of the final block tree (without and with additional rank and select support) on the $x$-axis. Plotting the throughput helps normalizing the running times for different input sizes. Furthermore, it highlights that the construction time of the block tree without rank and select support does not depend on the compressibility of the input.

Surprisingly, on most inputs, smaller block trees are not that much slower to construct than larger block trees. Our fastest construction algorithm for the smallest block trees is always LPF$_1$. Hence, we now only compare this algorithm with the original implementation. This also means that following the previous occurrence in the text naively is (for small block trees with fewer marked nodes) cheaper than explicitly computing the results. Furthermore, choosing $s = z$ provides no real benefit, as it does not result in a faster construction. For most inputs, the most space-efficient configuration of LPF$_1$ uses $\tau = 4$ and $b = 16$ and the fastest configuration of LPF$_1$ uses $\tau = 2$ and $b = 2$, see Table 3. Note that no two different configurations result in the same space requirements, even though, they can be very close.

When constructing only the block tree without rank and select support, LPF$_1$ is between 6.48 and 11.52 times faster than the original implementation (average: 9.51, median: 9.86). Computing the additional data for the rank and select support is the same for our and the original implementation. Thus, here LPF$_1$ is only between 3.61 and 11.23 times faster (average: 6.75, median: 6.24), despite the fact that the times for LPF$_1$ include also the LPF array construction. We further want to mention that the LPF array construction also introduces higher memory requirements during the construction than the fingerprint-based approaches. Hence, there is a working-space-time trade-off for the construction.

Non-Repetitive Inputs. We now give additional experimental results on the non-repetitive inputs. We only use 32 MiB prefixes of the texts, as the block tree construction algorithm by Belazzougui et al. [6] requires more than 1 TiB of working space for larger non-repetitive inputs. This is due to the order in which their algorithm constructs the block tree. Instead of first compressing all data internally, many operations and auxiliary data is computed on the uncompressed data. The results of these experiments are depicted in Figure 5.
Figure 2 Block tree construction without rank and select support, showing throughput (processed input in MiB per second) and space requirements of the final block tree (bits per character of the input) on repetitive inputs. The range of each x-axis depends on the compressibility of the input. Data points for each algorithm show different configurations of $\tau$ and $b$, see Table 3 for more details.

Overall, the results are similar to the results for the repetitive inputs: LPF$_1$ is the fastest construction algorithm most of the time. However, on some inputs, the dynamic programming LPF$_{DP}$ is faster. This is due to the long chains of previous occurrences that have been marked. Since the texts are non-repetitive, there are fewer marked blocks overall, resulting in bigger block trees. Overall, for non-repetitive inputs, computing larger block trees is slightly faster than computing very space-efficient block trees. This becomes very apparent for block trees with rank and select support.

### 6.2 Parallel Block Tree Construction

While the total running time of our algorithm is fast compared to our competitor, on average 71.33% of the running time is spent for the LPF array construction. Fortunately, libsais supports parallel computation. In addition, we use the LZ77 factorization algorithm by...
Shun and Zhao [63] that requires $O(n)$ work and $O(\log^2 n)$ time.\footnote{See https://github.com/zfy0701/Parallel-LZ77, last accessed 2023-07-04.} We also parallelized the construction of the rank and select support with a straight-forward implementation since the computed values are independent for each character.

We only achieve a speedup using up to 32 cores. This is most likely due to the fact that only eight memory controllers are available, which have to be shared by 16 groups of 4 cores. As soon as we use more than 32 cores, multiple groups have to share a controller. With 32 cores, we achieve a speedup of up to 6.64 (4.71 on average). This comes very close to the speedups of the parallel \texttt{libsais}, which achieves a speedup of at most 6 (5.2 on average). The additional speedup can be explained by the speedup thanks to the parallel construction of the rank and select support.
Figure 4 Parallel (strong scaling) block tree construction using the configuration $\tau = 8$ and $b = 16$. Construction times of a block trees includes parallel LPF array construction time.

7 Conclusion and Future Work

The LPF array allows us to construct the block tree up to an order of magnitude faster than using Karp-Rabin fingerprints. All tested algorithms produce the same block trees (when using the same parameters). A simple parallelization of our algorithm results in a speedup of up to 6.64 using 32 cores. However, the scalability of the current state of the algorithm is mostly limited by the LPF array computation. Here, it might be interesting to investigate a parallelization of the construction algorithm based on Karp-Rabin fingerprints using concurrent hash tables [47]. In general, better scalability is of great interest, as otherwise, construction speed similar to wavelet trees seems hard to achieve.

In the light that the LPF array can be represented in $2n + o(n)$ bits [3] with algorithms computing this representation in compact [3] or compressed space [60], future work includes engineering a more memory-efficient LPF array construction. Further improvements in construction time can be obtained by introducing stricter rules for the marking of nodes in the block tree rendering the pruning phase unnecessary. Finally, we want to compress the block tree recursively by using block trees internally.
Figure 5 Block tree construction throughput and space requirements per character of the input on 32 MiB prefixes of the non-repetitive inputs.
References


87:20 Faster Block Tree Construction


