Accelerated Partial Decoding in Wavelet Trees*

Gilad Baruch[a], Shmuel T. Klein[a], Dana Shapira[b]

[a] *Department of Computer Science, Bar Ilan University, Ramat Gan, Israel*
gilad.baruch@gmail.com, tomi@cs.biu.ac.il

[b] *Computer Science Department, Ariel University, Israel*
shapird@g.ariel.ac.il

Abstract

A Wavelet Tree is a compact data structure which is used in order to perform various well defined operations directly on the compressed form of a file. As *random access* is one of these operations, the underlying file is not needed anymore, and is often discarded because it can be restored, when necessary, by repeated accesses. This paper concentrates on cases in which partial decoding of a contiguous portion of the file, or even its full decoding, is still needed. We show how to accelerate the decoding relative to repeatedly performing random accesses on the consecutive indices. Experiments on partial and full decoding support the effectiveness of our approach, and present an improvement of about 50% of the run-time for full decoding, and about 30% or more for partial decoding of large enough ranges.

Keywords: Wavelet tree, direct access, range decoding

1. Introduction

The compressed pattern matching paradigm was defined by Amir and Benson [1] for directly searching for a pattern within a compressed file without decompressing it. The challenge was to be able to achieve an answer to a query in time proportional to the size of the compressed input file. In recent years the focus shifted to attaining a faster query response at the price of some more preprocessing. This led to the development of a new discipline known as *compact data structures* [21], suggesting a compressed

*This is an extended version of a paper that has been presented at the Prague Stringology Conference (PSC'16) in 2016, and appeared in its Proceedings, 63–70.

representation of the data involved, that still provides means for efficient query operations. The objective is to design data structures for supporting well defined specific operations known in advance, for example, random access, using just about the information theoretic lower bound amount of space, and retaining the ability for efficient queries in-place, and without any decompression.

The existence of algorithms enabling random access causes the compressed text itself to be redundant, so that it is not needed any more and may be discarded. In case further operations are desired, the original file, or its relevant parts, can be reconstructed using random access repeatedly.

One of the relevant data structures in this context is known as a Wavelet tree, and will be defined in more detail below. Given such a Wavelet tree, we suggest in this paper to enhance random access for a sequence of consecutive indices, possibly the entire file, via *range* decoding, unlike the acceleration of a single random access in [3]. The proposed method uses the dependency between the consecutive indices, exploiting the fact that the corresponding search paths in the Wavelet tree have some overlapping prefix, rather than repeatedly performing random access on each index independently from the others. Experiments on partial and full decoding support the effectiveness of our approach, and present an improvement of about 50% of the run-time for full decoding, and about 30% for partial decoding for large enough ranges.

There are obviously many scenarios in which the partial decoding of a large compressed file is needed, and we shall mention only one example. Searches in large full text retrieval systems are generally not performed by direct pattern matching, but are rather based on so-called *inverted files*, dictionaries and concordances that have been built in a pre-processing stage, see, e.g., [15, Section 8.1.1]. To answer a query, rather than scanning the text for the occurrences of the requested terms, the sorted lists of their locations are retrieved from the auxiliary files and are processed according to the Boolean query at hand. This results in a series of pointers $\ell_1, \ldots, \ell_r$ into the given text, and it is only at this stage that the compressed file is accessed, directly at ℓ_1, then at ℓ_2, etc. For each of these locations, the user is generally interested in seeing not just the requested terms of the query, but also some of their local contexts. These contexts are known as *snippets* or *KWICs* (KeyWord In Context) [19], and consist of about one or two lines of text. If according to this short excerpt, the user judges that the occurrence is relevant, a larger portion of the file may be decoded. In any case, for a single query, the compressed file may be accessed at many different locations, and the possibility of partial decoding is critical for this application.

Our paper is organized as follows. Section 2 introduces the relevant notions and discusses previous research dealing with random access to files encoded using variable length codes. Section 3 recalls the details of Wavelet trees and presents our proposed algorithm for accelerating partial decoding. Section 4 then empirically compares our suggested algorithm to the traditional one. Finally, Section 5 concludes.

2. Definitions and Previous Research

The binary tree corresponding to a prefix code C is defined as follows: we imagine that every edge pointing to a left child is labeled 0 and every edge pointing to a right child is labeled 1; each node v is associated with the bit string obtained by concatenating the labels on the edges on the path from the root to v; finally, the tree is defined as the binary tree for which the set of bit strings associated with its leaves is the code C.

Given a probability distribution $\{p_1, p_2, \ldots, p_\sigma\}$ for the elements of an alphabet Σ of size σ, Huffman's [12] algorithm generates an optimal prefix code, known as a Huffman code, in the sense that it assigns codewords of lengths $\{\ell_1, \ell_2, \ldots, \ell_\sigma\}$ to the characters of Σ such that the weighted average codeword length $\sum_{i=1}^{\sigma} p_i \ell_i$ is minimized. The corresponding tree is called a *Huffman tree*.

One of the common fundamental compact data structures is a *bitmap*, also known as a *bit-vector*, which is a finite size array of bits for supporting operations rank and select, as well as random access. More formally, given a bitmap B of size n and a bit $b \in \{0, 1\}$,

$\mathsf{rank}_b(B, i)$ returns the *number* of occurrences of b in B up to and including position i;

$\mathsf{select}_b(B, i)$ returns the *position* of the ith occurrence of b in B; and

$\mathsf{access}(B, i)$ returns the bit $B[i]$ for any $1 \leq i \leq n$.

Jacobson [13] showed that rank, on a bitmap of length n, can be computed in constant time using $n + O(\frac{n \log \log n}{\log n}) = n + o(n)$ bits. Other efficient implementations for rank and select are due to Raman et al. [24], Okanohara and Sadakane [23], Barbay et al. [2] and Navarro and Providel [22], to list only a few.

Bit-vectors are generalized to compact data structures that support the same set of operations, but consider a finite alphabet $\Sigma = \{a_1, \ldots, a_\sigma\}$ of size σ rather than a binary one.

A well known succinct data structure suggested to cope with the three operations rank, select and random access performed on a file T, is the *Wavelet tree* (WT), defined by Grossi et al. [10]. A balanced Wavelet tree for a text file T of size n over an alphabet Σ, is a full binary tree (all leaves are on the lowest or on the two lowest levels), whose leaves are labeled by the elements of Σ, and whose internal nodes store bitmaps. The alphabet of symbols $\{a_1, \ldots, a_\sigma\}$ is divided into two sets $\{a_1, \ldots, a_{\sigma/2}\}$ and $\{a_{\sigma/2+1}, \ldots, a_\sigma\}$. The root of the Wavelet tree holds the bitmap B so that $B[i] = 0$ if $T[i] \in \{a_1, \ldots, a_{\sigma/2}\}$ and $B[i] = 1$, otherwise. The bitmap B of size n, one bit for each character of the text T, is then used to partition the text into to subsequences $T_1 = \{T[i] \mid B[i] = 0\}_{i=1}^{n}$ and $T_2 = \{T[i] \mid B[i] = 1\}_{i=1}^{n}$. The process is repeated recursively on the subsequences T_1 and T_2 of T corresponding to the left and right subtrees of the root. Using constant time rank and select on the bitmaps yields balanced Wavelet trees constructed in $O(n \log \sigma)$ time and require $n \log \sigma(1 + O(1))$ bits.

A balanced Wavelet tree is induced by a fixed length code of size $\approx \log \sigma$ and can be generalized to any prefix free code. The Wavelet tree is then associated with the corresponding compressed file $\mathcal{E}(T)$. The root of the Wavelet tree holds the bitmap obtained by concatenating the *first* bit of each of the sequence of codewords in the order they appear in $\mathcal{E}(T)$. The left and right children of the root hold, respectively, the bitmaps obtained by concatenating, again in the given order, the *second* bit of each of the codewords starting with 0, respectively with 1. This process is repeated similarly with the grand-children of the root that hold the bitmap obtained by concatenating the *third* bit of the sequence of codewords, and so on. An example of a Wavelet tree induced by a non-balanced Huffman tree, is given in the following section.

Wavelet trees require space of $nH_h + O(\frac{n \log \log n}{\log_\sigma n})$ bits, for all $h \geq 0$, where H_h denotes the hth-order empirical entropy of the text, which is at most $\log \sigma$; processing time is just $O(m \log \sigma + \text{polylog}(n))$ for searching any pattern sequence of length m.

Multiary Wavelet trees replace the bitmaps by sequences over sublogarithmic sized alphabets in order to reduce the $O(\log \sigma)$ height of binary Wavelet trees, and obtain the same space as the binary ones, but their times are reduced by an $O(\log \log n)$ factor. If the alphabet Σ is small enough, say $\sigma = O(\text{polylog}(n))$, the tree height is a constant and so are the query times.

Klein and Shapira [16] applied a pruning strategy to WTs based on Fibonacci Codes, so that in addition to supporting improved rank, select and random access to the corresponding Fibonacci encoded file, the size of

the Fibonacci based WT is reduced. However, for any finite probability distribution, the compression by a prefix of the Fibonacci code will always be inferior to what can be achieved by a Huffman code. In a following research [3], we therefore suggested a different method based on pruning a Huffman shaped Wavelet tree according to the underlying *skeleton* Huffman tree [14]. The resulting smaller WT is especially designed to support faster random access for a single index and saves memory storage, at the price of less effective rank and select operations, when compared to the original Huffman shaped WTs. The general idea is to apply some cut-off strategy on the internal nodes of the WTs, so that the overhead of the additional storage, used by the data structures for processing the stored bitmaps, is reduced. Moreover, the average path lengths corresponding to the codewords is also decreased, and so is also the average time spent for traversing the paths from the root to the desired leaf, which is the basic processing component used to evaluate random access.

If the text is encoded by using some standard fixed length code, such as ASCII, random access to the ith codeword is straightforward for any i. However, fixed length codes are wasteful from the storage point of view, and have therefore been replaced in many applications by variable length codes. This may improve the compression performance, but at the price of losing the simple random access, because the beginning position of the ith codeword is the sum of the lengths of all the preceding ones.

A possible solution to allow random access is to divide the encoded file into blocks of size b codewords, and to use an auxiliary vector to indicate the beginning of each block. The time complexity of random access depends on the size b, as we can begin from the sampled bit address of the $\frac{i}{b}$th block to retrieve the ith codeword. This method, known as *sampling*, thus suggests a processing time vs. memory storage tradeoff, since direct access requires decoding $i - \lfloor \frac{i}{b} \rfloor b$ codewords, i.e., less than b.

Ferragina and Venturini [6] replace every block of a fixed number ℓ of symbols by a single codeword of a Huffman code built according to the frequencies of occurrence of the blocks. Their idea is to represent T as a sequence of $\lceil \frac{n}{\ell} \rceil$ macro-symbols over the macro-alphabet Σ^ℓ, where ℓ is chosen as $\left\lceil \frac{\log_\sigma n}{2} \right\rceil$. To guarantee constant time direct access to the encoding of the blocks, they use a two level storage scheme for the starting positions: absolute ones every $\Theta(\log n)$ contiguous blocks, and relative ones for the rest. Their representation uses $O\left(\frac{n \log\log n}{\log_\sigma n}\right)$ bits.

Teuhola [25] extends Moffat and Stuiver's work [20] on *Interpolative coding*, so that direct access, as well as finding the position in which the prefix

sum exceeds some threshold, is achieved in $O(\log n)$ time. They consider the successive gaps in the sequence as basic elements, and build a complete binary tree of pairwise sums with the elements as leaves.

Variable Byte (VByte) codes [28] are byte-aligned codewords designed for speeding up decoding. The highest bit of every byte composing the codeword is used as a flag-bit to distinguish between the byte that starts the codeword and the remaining bytes. The highest bit of each codeword is 0 in the byte holding the most significant bits and 1 in the others. Thus, the 0 bits acts as a comma between a sequence of codewords.

Brisaboa et al. [4] apply an n-ary Wavelet tree on VBytes instead of a binary one as follows. The root of the Wavelet tree contains the first *byte*, rater than the first bit, of all the codewords, in the same order as they appear in the original text. The second level nodes then store the second byte of the corresponding codewords, and so on. The reordering of the compressed text bits becomes an implicit index representation of the text, which is empirically shown in [4] to be better than explicit main memory inverted indexes built on the same collection of words, when little extra space in addition to the compressed text is available.

The idea of VByte codes is generalized to any block of a fixed size of b bits rather than 8 used in the original definition of VByte codes. In the worst case, these codes lose one bit per b bits plus b bits for an almost empty leading block. The rank data structures are integrated into these extended Vbyte codes to form *directly accessible codes* (DACs) [5]. DACs can be regarded as a reorganization of the bits of the extended Vbyte, plus extra space for the rank structures, that enables direct access to it. First, all the least significant blocks of all codewords are concatenated, then the second least significant blocks of all codewords having at least two blocks, and so on. Then the rank data structure is applied on the flag bits for attaining $\frac{\log(M)}{b}$ direct access processing time, where M is the maximum integer to be encoded. The authors suggest trade-offs of space and access time by different constructions of DACs using different values of b for each level, possibly introducing more levels, and thus slower access time.

Külekci [18] suggested the usage of Wavelet trees for *Elias* and *Rice* variable length codes. The method is based on handling separately the unary and binary parts of the codeword in different strings so that random access is supported in constant time. As an alternative, applying WTs over the lengths of the unary section of each Elias or Rice codeword is proposed, while storing their binary section, allowing direct access in time $\log r$, where r is the number of distinct unary lengths in the file.

3. Accelerating partial decoding for Wavelet Trees

3.1. Wavelet tree details

As mentioned above, the nodes of the WT are annotated by bitmaps. These bitmaps can be stored as a single bit stream by concatenating them in order of any predetermined top-down tree traversal, such as depth-first or breadth-first. No delimiters between the individual bitmaps are required, since we can restore the tree topology along with the bitmaps lengths at each node, once the size n of the text is given in the header of the file.

For the sake of keeping this paper self-contained, we repeat here in Figure 1 the example Wavelet tree of [3] induced by the Huffman tree for the example text $T =$ `A--HUFFMAN--WAVELET--TREE--MATTERS`. We assume the alphabet to be $\Sigma =$`{-,E,A,T,F,M, R,H,L,N,S,U,V,W}`, and its elements appear in the sample text T with frequencies $8, 5, 4, 4, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1$, respectively. The Huffman encoded file is a binary string, the beginning of which is:

$$\begin{array}{rcccccccccc} \mathcal{E}(T) & = & \underline{0}11 & \underline{0}0 & \underline{0}0 & \underline{1}\overline{1}001 & \underline{1}\overline{1}101 & \underline{1}\overline{0}10 & \underline{1}\overline{0}10 & \underline{1}\overline{0}11 & \cdots \\ & & \texttt{A} & \texttt{-} & \texttt{-} & \texttt{H} & \texttt{U} & \texttt{F} & \texttt{F} & \texttt{M} & \cdots \end{array}$$

in which spaces between the codewords have been added for clarity. The WT is the entire figure including the annotating bitmaps.

The bitmaps stored in the nodes of the WT are in fact a reordering of the bits of the encoded file. The bitmap stored in the root consists of 34 bits, one for each of the characters of T, and starts with $00011111\cdots$, corresponding to the underlined bits above. More specifically, the bitmap is the concatenation of the *first* bits of the 34 codewords in the encoding of T. These codewords are then partitioned into those starting with a 0-bit, in positions 1, 2, 3, 9, 11, 12, 14, 16, 18, etc, and those starting with a 1-bit, in the other positions. The root's left child then refers to the 17 codewords starting with a 0-bit. Collecting the *second* bits of these codewords in the order they appear in the sample text, results in the bitmap 10010011100110011, which is stored in the root's left child. Similarly, the *second* bits of the 17 codewords starting with 1 are concatenated to yield 11000111100100011, which is stored in the right child of the root; the first few bits of this string correspond to the overlined bits in the sample above. This process of splitting the set of codewords corresponding to some node into two sub-sets that are assigned to the node's children, and collecting the i-th bit of the codewords for nodes on level i, continues for all internal nodes.

The algorithm for extracting the i-th element of the text T by means of a WT rooted by v_{root} is given in Figure 2, using the function call extract(v_{root},i).

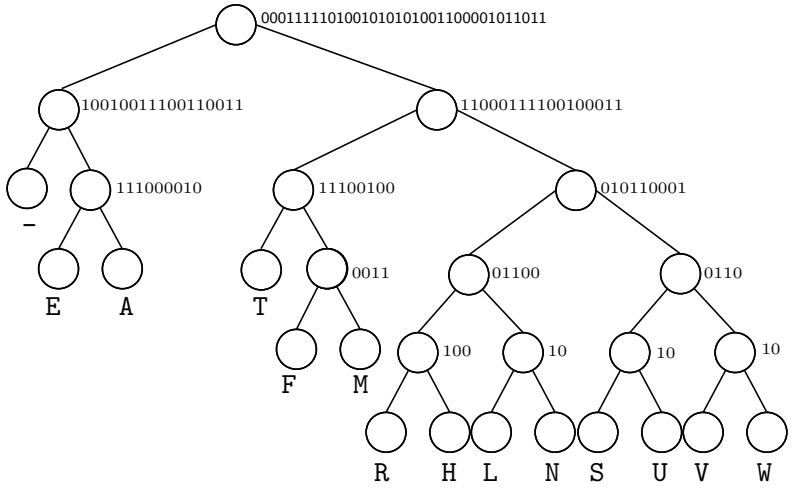

FIGURE 1: *The WT induced by the Huffman tree corresponding to the frequencies* {8,5,4,4,2,2,2,1,1,1,1,1,1,1} of {-,E,A,T,F,M,R,H,L,N,S,U,V,W}, respectively, assigned to the leaves, left to right.

B_v denotes the bitmap belonging to vertex v of the WT, and the dot $\cdot$ denotes concatenation. Computing the new index in the following bitmap is done by the rank operation in lines 2.1.2 and 2.2.2. The decoding of the codeword cw in line 3 using the decoding function $\mathcal{D}$ can be done by a preprocessed lookup table.

The straightforward decoding algorithm works on successive indices *independently*, starting each time at the root, and working its way down the WT until a leaf is reached, where the information for that index is extracted. The formal algorithm for partial decoding of a range of elements with indices between i and j is given in Figure 3, where the decoding is output to an array A.

3.2. The new partial decoding method

Unlike the traditional approach, the proposed algorithm takes advantage of the fact that partial decoding is applied on a strictly monotonic increasing series of indices. During runtime, partial calculations are stored so that the same computations are not done more than once. A similar idea is performed in the well known KMP algorithm [17] for pattern matching, in which the algorithm makes sure that it does not match any character more than once.

In spite of the fact that there exist constant time solutions for rank and select that require sublinear extra space, in many practical cases, simple

```
extract(v, i)
1    cw ⟵ ϵ
2    while v is not a leaf
2.1        if B_v[i] = 0 then
2.1.1          cw ⟵ cw · 0
2.1.2          i ⟵ rank_0(B_v, i)
2.1.3          v ⟵ left(v)
2.2        else
2.2.1          cw ⟵ cw · 1
2.2.2          i ⟵ rank_1(B_v, i)
2.2.3          v ⟵ right(v)
3    return D(cw)
```

FIGURE 2: *Extracting the i-th element of T from a WT rooted at v.*

```
range_decoding(i, j)
1    for k = i to j
1.1        A[k − i] ⟵ extract(root, k)
2    return A
```

FIGURE 3: *Traditional range decoding.*

solutions are better in terms of time and space [9]. Thus, in order to save space, the rank operation in lines 2.1.2 and 2.2.2 is not necessarily done in $O(1)$ time. In either case, it can even be done faster using the fact that the ranks of consecutive zeros or consecutive ones in a given bitvector differ only by one. More precisely, if for indices i and j, it holds that $B_v[i] = 0$ and $B_v[j] = 0$, but for each index k between i and j, $B_v[k] \neq 0$, then

$$\mathsf{rank}_0(B_v, j) = \mathsf{rank}_0(B_v, i) + 1.$$

For this reason the rank results are maintained for each internal node of the WT which has already been visited during the production of the solution of the current range decoding query.

Each time a node is visited for the first time, the rank queries in lines 2.1.2 and 2.2.2 of Figure 3 are fully computed for the corresponding bit using the rank/select data structures. The resulting value is then stored at

the node for future use. If, during the computation of $\mathsf{extract}(i)$, a node is reached that has already been visited, the stored value is extracted and incremented, rather than recalculated from scratch by the rank operation, as done for the first time. In the special case of full decoding, the rank results for all nodes are initialized by zero and none of them are obtained by means of the rank/select data structure.

Since $\mathsf{rank}_{1-b}(B,i) = i - \mathsf{rank}_b(B,i)$, only one of the two, say, $\mathsf{rank}_0(B,i)$ needs to be stored. We denote the stored value in node v by $rnk(v)$. At allocation time of a new node, its rnk value will be initialized by -1. The line $i \longleftarrow \mathsf{rank}_0(B_v,i)$ in Figure 2 is replaced by the top half of the code of Figure 4 indicated by 2.1.2, whereas the line $i \longleftarrow \mathsf{rank}_1(B_v,i)$ in Figure 2 is replaced by the bottom half of the code, indicated by 2.2.2.

```
2.1.2   if rnk(v) < 0   // first visit at v
            i ⟵ rank_0(B_v, i)
        else
            i ⟵ rnk(v) + 1
        rnk(v) ⟵ i

2.2.2   if rnk(v) < 0   // first visit at v
            rnk(v) ⟵ rank_0(B_v, i)
            i ⟵ i − rnk(v)
        else
            i ⟵ (i − rnk(v)) + 1
```

FIGURE 4: *Partial decoding acceleration.*

Note that the two parts are not completely symmetrical. The upper part, 2.1.2, corresponds to a 0-bit, so the assignment to $rnk(v)$ is excluded from the if-clause, as it has to be performed on any visit to the node v. The lower part, 2.2.2, corresponds to a 1-bit, thus the value of $rnk(v)$ is only set at the first visit to the node, since it does not change on recurring visits: $\mathsf{rank}_0(B_v,i) = \mathsf{rank}_0(B_v,i-1)$ when the ith bit is 1.

3.3. Analysis

The impact of the proposed amendment for partial decoding obviously depends on the subset of characters of the alphabet which appear in the given range. It is therefore not possible to assess the expected savings relative

to the traditional approach analytically, and we shall report on empirical experiments in the next section. Our theoretical analysis will be restricted to point to the extreme cases between which the performance savings will fluctuate.

A first observation is that the suggested improvement is mostly indeed such, since the number of required rank evaluations is in the worst case the same as for the traditional method without the $rnk(v)$ variable, and in all other cases, this number can only be reduced. The time spent on the additional if-statements in 2.1.2 and 2.2.2 will generally be compensated for by the savings in the number of the much more expensive rank operations.

Let $h(y)$ denote the codeword assigned by the given Huffman code to the character $y \in \Sigma$, and let $x_1, x_2, \ldots, x_t$ denote the t (not necessarily different) characters in the range to be decoded. The traditional decoding method, denoted as $\mathcal{T}$, traverses the Huffman tree shaped WT repeatedly from its root to the leaves corresponding to $h(x_1), h(x_2)$, etc. The total number of visited internal nodes is thus $\sum_{i=1}^{t} |h(x_i)|$, and there is one rank evaluation for each of these nodes. By contrast, in the new improved version, denoted $\mathcal{N}$, a standard rank evaluation is only performed on the first visit at a given node, so that the overall number of evaluations is the number of *different* visited internal nodes. For full decoding, even the initializing rank evaluation may be saved as mentioned above.

The largest savings by using $\mathcal{N}$ instead of $\mathcal{T}$ will thus be achieved for a range containing t copies of a single character z: there will be $t|h(z)|$ evaluations for $\mathcal{T}$ and only $|h(z)|$ for $\mathcal{N}$, an improvement by a factor of t.

The other extreme case is when all the characters in the range are different, and moreover, the paths in the WT leading from the root to the leaves corresponding to these t characters share as few common nodes as possible. These paths can not be disjoint: they all start at the root, and only at level $\lceil \log_2 t \rceil$ are there at least t nodes, so that above this level in the WT, some paths must share some common nodes. In this worst case from the point of view of the improvement of $\mathcal{N}$ over $\mathcal{T}$, the paths from level $\lceil \log_2 t \rceil$ to the leaves will be disjoint, so for this part, the numbers of rank evaluations will be the same for $\mathcal{N}$ and $\mathcal{T}$. For the upper parts of these paths, the number of evaluations for $\mathcal{T}$ will be $t(\lceil \log_2 t \rceil - 1)$, while for $\mathcal{N}$ it will be the number of nodes in these levels, which is roughly $\frac{t}{2} + \frac{t}{4} + \cdots + 2 + 1 = t - 1$. There are thus additive savings of at least $t \log t - 2t$ rank evaluations, even in the worst case.

4. Experimental Results

We considered six texts of different languages and alphabet sizes. *ftxt* is the French version of the European Union's JOC corpus, a collection of pairs of questions and answers on various topics used in the ARCADE evaluation project [26]; *ebib* is the Bible (King James version) in English, in which the text was stripped of all punctuation signs; *English* is the concatenation of English text files selected from etext02 to etext05 collections of the Gutenberg Project, from which the headers related to the project were deleted so as to leave just the real text; and *dblp* is an XML file providing bibliographic information on major Computer Science journals and proceedings, obtained from `dblp.uni-trier.de`; *Einstein* is the collection of all the versions of the Wikipedia page about Albert Einstein in English; and *sources* is formed by C/Java source codes obtained by concatenating all the .c, .h and .java files of the linux-2.6.11.6 distributions.

Table 1 presents some information on the data files involved. The second column presents the original file sizes in MB, and the third column gives the sizes of the alphabets, i.e., the number of encoded characters.

File	size (MB)	$\|\Sigma\|$
ftxt	7.6	132
ebib	3.5	53
English	200.0	225
dblp	200.0	96
Einstein	446.0	139
sources	200.0	230

TABLE 1: *Information about the used datasets*

Our implementation used the *Succinct Data Structure Library* [7], which is an open-source library implementing succinct data structures efficiently in C++. All experiments were conducted on a machine running 64 bit Linux Ubuntu with an Intel Core i7-6700 at 2.60GHz processor, 6144K L3 cache size of the CPU, and 32GB of main memory.

Our first experiment considers several variants of the Wavelet tree with different topology and different rank data structure implementations. As all variants produced basically the same results for full decoding, we present here the ones for the Huffman based Wavelet tree and rank implementation of Vigna [27]. Table 2 compares the processing times of full decoding, averaging 10 independent runs, of the traditional approach to that of our algorithm. The second and third columns give the processing times, in seconds, of the

traditional and the proposed algorithm, respectively, and the fourth column is the ratio of the latter to the former. As can be seen, our method is about 50% faster, and consistently achieves a significant processing time improvement relative to the traditional approach.

File	traditional	proposed	ratio
ftxt	0.80	0.39	0.49
ebib	0.31	0.14	0.45
English	20.41	10.94	0.54
dblp	22.52	8.92	0.40
Einstein	51.61	26.91	0.52
sources	22.94	11.05	0.48

TABLE 2: *Full decoding processing time comparison.*

Our next experiment compares the processing times for partial decoding. The range sizes were chosen as a series of increasing powers of 2 until the size of the entire file is reached. As, for the shorter ranges, the results are only slightly different, the times, in microseconds, are given explicitly in Table 3. For each of the test files and range sizes given in the title line, the partial decoding was run 100 times, with randomly chosen starting points. The displayed numbers are the averages over these runs. The same setup was used also for the results displayed in Figure 5 and Table 4.

File		4	8	16	32	64	128
ftxt	traditional	2.42	3.17	4.02	5.67	8.82	14.87
	proposed	2.72	3.2	3.91	5.23	8.34	10.87
ebib	traditional	2.22	2.79	3.67	4.58	7.38	12.30
	proposed	2.38	2.86	3.21	4.37	5.68	9.23
English	traditional	3.16	3.81	4.98	7.07	10.03	17.24
	proposed	2.91	3.90	4.75	6.28	8.11	12.69
dblp	traditional	3.29	4.09	5.42	7.39	12.32	19.99
	proposed	3.87	4.52	5.60	6.32	8.93	13.07
Einstein	traditional	4.78	4.13	5.83	8.02	11.86	21.01
	proposed	3.99	3.64	4.83	6.66	10.01	14.57
sources	traditional	3.07	4.03	6.52	7.41	12.09	20.15
	proposed	2.82	3.83	5.31	6.93	9.53	14.25

TABLE 3: *Partial decoding processing time comparison for short ranges.*

As can be seen, the traditional method is slightly faster only for the

very small ranges, typically up to the value of 16. In all cases, our method becomes faster for ranges beyond this threshold. This phenomenon can be clarified by the fact that on average, most of the nodes are visited for the first time in the case shorter ranges are considered. As a result, an extra if clause is performed as compared to the traditional approach. That is, the new approach is worthwhile when nodes are revisited, so that the time saved by eliminating rank operations will ultimately dominate the time consumed by this extra if clause.

Figure 5 displays the processing times for the *ebib* dataset on the larger ranges. The plots are given on a log-log scale, showing the processing time, in microseconds, as function of the range size, measured in number of characters. The results for the other test files were similar.

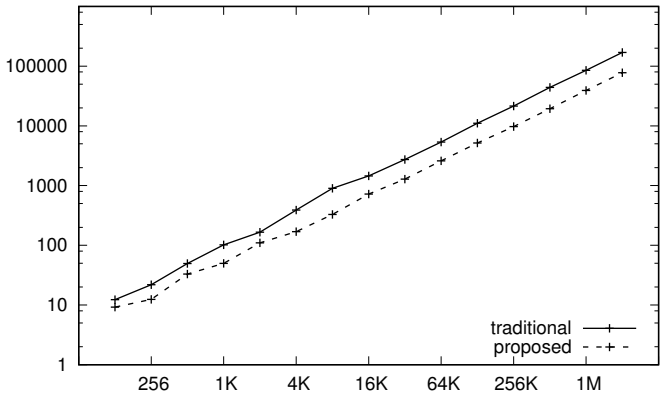

FIGURE 5: *Range decoding applied on* ebib *for longer ranges.*

The plots are almost straight lines, showing a linear dependency of the decoding time on the size of the range. The fact that the lines are roughly parallel implies an almost constant ratio of the speed of the proposed method to that of the traditional one. Though the lines seem to be close, recall that the display uses a log-scale — the actual ratio being about 0.45.

Our last experiment wishes to verify that the better performance of the proposed method does not depend neither on a specific implementation of Wavelet trees, nor on different implementations of rank and select. Table 4 considers a fixed length range of 512 bytes, and compares the processing times of both methods using different implementations of Wavelet trees and rank and select on the file *ebib*. Test results on the other files were similar. As to Wavelet trees, their shape can be that of a Huffman tree, or according to Hu Tucker's algorithm [11], in which the ordered set of codewords is assigned

to the symbols in alphabetic order, or simply a balanced tree. For rank and select, the first and second approaches of Vigna [27] are used, as well as Gog and Petri's method [8] called interleaving, and two implementations of Raman et al. [24], using different block sizes [22].

WT implementation	rank/select implementation	traditional method	proposed method	ratio
Huffman	Vigna1	51	30	0.59
	Vigna2	100	32	0.32
	interleaving	96	33	0.34
	RRR–15	341	200	0.59
	RRR–63	1358	625	0.46
Hu Tucker	Vigna1	50	29	0.59
	Vigna2	100	31	0.31
	interleaving	111	35	0.32
	RRR–15	390	151	0.39
	RRR–63	1736	758	0.44
Balanced	Vigna1	82	35	0.43
	Vigna2	141	38	0.27
	interleaving	192	57	0.30
	RRR–15	500	256	0.51
	RRR–63	1669	666	0.40

TABLE 4: *Comparison on different Wavelet trees and* rank/select *implementations.*

5. Conclusion

We have presented an enhanced range decoding especially designed for Wavelet trees, and gave empirical evidence that the running time performance of full and partial decoding is significantly improved as compared to the running time of the traditional approach. Our improvement can be implemented without any additional storage: the *rnk* values are generated and used only during run time and need only $O(\Sigma)$ bytes of RAM, which is independent of the size of the text.

References

[1] A. AMIR AND G. BENSON: *Efficient two-dimensional compressed matching*, in Proc. IEEE Data Compression Conference DCC–92, 1992, pp. 279–288.

[2] J. Barbay, T. Gagie, G. Navarro, and Y. Nekrich: *Alphabet partitioning for compressed rank/select and applications.* Algorithms and Computation, Lecture Notes in Computer Science, 6507 2010, pp. 315–326.

[3] G. Baruch, S. T. Klein, and D. Shapira: *A space efficient direct access data structure.* Journal of Discrete Algorithms, 43 2017, pp. 26–37.

[4] N. R. Brisaboa, A. Fariña, G. Ladra, and G. Navarro: *Reorganizing compressed text*, in Proc. of the 31th Annual Internetional ACM SIGIR Conference on Research and Developing in Information Retrieval (SIGIR), 2008, pp. 139–146.

[5] N. R. Brisaboa, S. Ladra, and G. Navarro: *DACs: Bringing direct access to variable length codes.* Information Processing and Management, 49(1) 2013, pp. 392–404.

[6] P. Ferragina and R. Venturini: *A simple storage scheme for strings achieving entropy bounds.* Theoretical Computer Science, 372 2007, pp. 115–121.

[7] S. Gog, T. Beller, A. Moffat, and M. Petri: *From theory to practice: plug and play with succinct data structures*, in International Symposium on Experimental Algorithms (SEA 2014), 2014, pp. 326–337.

[8] S. Gog and M. Petri: *Optimized succinct data structures for massive data*, in Software, Practice and Experience, vol. 44, 2014, pp. 1287–1314.

[9] R. González, S. Grabowski, V. Mäkinen, and G. Navarro: *Practical implementation of rank and select queries*, in Poster Proceedings of 4th Workshop on Efficient and Experimental Algorithms (WEA–05), 2005, pp. 27–38.

[10] R. Grossi, A. Gupta, and J. S. Vitter: *High-order entropy-compressed text indexes*, in Proceedings of the 14th Annual SIAM/ACM Symposium on Discrete Algorithms (SODA), 2003, pp. 841–850.

[11] T. C. Hu and A. C. Tucker: *Optimal computer search trees and variable-length alphabetical codes.* SIAM Journal on Applied Mathematics, 21(4) 1971, pp. 514–532.

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

[13] G. Jacobson: *Space efficient static trees and graphs*, in Proceedings of FOCS, 1989, pp. 549–554.

[14] S. T. Klein: *Skeleton trees for the efficient decoding of Huffman encoded texts*. Information Retrieval, 3(1) 2000, pp. 7–23.

[15] S. T. Klein: *Basic Concepts in Data Structures*, Cambridge University Press, Cambridge, UK, 2016.

[16] S. T. Klein and D. Shapira: *Random access to Fibonacci encoded files*. Discrete Applied Mathematics, 212 2016, pp. 115–128.

[17] D. E. Knuth, J. H. Morris, and V. Pratt: *Fast pattern matching in string*, in SIAM Journal on Computing, vol. 6, 1977, pp. 323–350.

[18] M. O. Külekci: *Enhanced variable-length codes: Improved compression with efficient random access*, in Proc. Data Compression Conference DCC–2014, Snowbird, Utah, 2014, pp. 362–371.

[19] H. P. Luhn: *Keyword-in-context index for technical literature*. American Documentation, 11(4) 1960, pp. 288–295.

[20] A. Moffat and L. Stuiver: *Binary interpolative coding for effective index compression*. Information Retrieval, 3(1) 2000, pp. 25–47.

[21] G. Navarro: *Compact Data Structures: A Practical Approach*, Cambridge University Press, Cambridge, UK, 2016.

[22] G. Navarro and E. Providel: *Fast, small, simple rank/select on bitmaps*. Experimental Algorithms, LNCS, 7276 2012, pp. 295–306.

[23] D. Okanohara and K. Sadakane: *Practical entropy-compressed rank/select dictionary*, in Proc. ALENEX, SIAM, 2007.

[24] R. Raman, V. Raman, and S. Rao Satti: *Succinct indexable dictionaries with applications to encoding k-ary trees and multisets*. Transactions on Algorithms (TALG), 2007, pp. 233–242.

[25] J. Teuhola: *Interpolative coding of integer sequences supporting log-time random access*. Information Processing and Management, 47(5) 2011, pp. 742–761.

[26] J. Véronis and P. Langlais: *Evaluation of parallel text alignment systems: The* arcade *project.* Parallel Text Processing, 2000, pp. 369–388.

[27] S. Vigna: *Broadword implementation of rank/select queries*, in Proc. of 7th Workshop on Experimental Algorithms (WEA), 2008, pp. 154–168.

[28] H. Williams and J. Zobel: *Compressing integers for fast file access.* The Computer Journal, 42(30) 1999, pp. 192–201.