A Space Efficient Direct Access Data Structure

Gilad Baruch*, Shmuel T. Klein*, and Dana Shapira†

*Computer Science Department
Bar Ilan University
Ramat Gan 52900, Israel

†Department of Computer Science
Ariel University
Ariel 40700, Israel

gilad.baruch@gmail.com, tomi@cs.biu.ac.il, shapird@ariel.ac.il

Abstract

Given a file T, we suggest a data structure based on pruning a Huffman shaped Wavelet tree (WT) according to the underlying skeleton Huffman tree that enables direct access to the i-th element of T. This pruned WT is especially designed to support faster random access and save memory storage, at the price of less effective rank and select operations, as compared to the original Huffman shaped WT. We give empirical evidence that when memory storage is of main concern, our suggested data structure outperforms other direct access techniques such as those due to Külekci, DACs and sampling, with a slowdown as compared to DACs and fixed length encoding.

1. Introduction

Research in Lossless Data Compression was originally concerned with finding a good balance between the competing efficiency criteria of compressibility of the input, processing time and additional auxiliary storage for the involved data structures. Working directly with compressed data is now a popular research topic, including not only classical text but also various useful data structures, and with a wide range of possible applications. A common operation used in text processing is *Random Access*, which enables direct access to any element of the encoded text. Efficient random access to an encoded file may lead to effective range decoding, in which only a portion of the file bounded by two indices has to be decompressed, and may also improve *parallel decoding*, as different threads will decode disjoint ranges of the file in parallel. If the text is encoded by using some standard fixed length codes (FLC), random access to the i^{th} codeword is straightforward for any i. However, FLC 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 some features. For example, with variable length codes (VLC), random access becomes more involved, because the beginning position of the ith codeword is the sum of the lengths of all the preceding ones. In this paper we are interested in data structures supporting random access in efficient time, while using a compact representation.

*This is an extended version of a paper that has been presented at the Prague Stringology Conference (PSC'15) in 2015, and appeared in its Proceedings, 67–77, and a paper that has been presented at the Data Compression Conference (DCC'16) in 2016, and appeared in its Proceedings 63-72.

A Wavelet tree (WT), suggested by Grossi et al. [13], is a data structure which reorders the bits of the compressed file into an alternative form, thereby enabling direct access, as well as other efficient operations. Wavelet trees can be defined for any prefix code, and the tree structure associated with this code is inherited by the WT. The internal nodes of the WT are annotated with bitmaps. The root of the WT holds the bitmap obtained by concatenating the *first* bit of each of the sequence of codewords in the order they appear in the compressed text. 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 on the next levels: the grand-children of the root hold the bitmaps obtained by concatenating the *third* bit of the sequence of codewords starting, respectively, with 00, 01, 10 or 11, if they exist at all, etc.

Various manipulations on the bitmaps of the WT are based on fast implementations of operations known as rank and select. These are defined for any bit vector B and bit $b \in \{0, 1\}$ as:

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

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

Efficient implementations for rank and select are due to Jacobson [14], Raman et al. [24], Okanohara and Sadakane [23], Barbay et al. [1] and Navarro and Providel [22], to list only a few. Wavelet trees can be seen as extensions of rank and select operations to a general alphabet.

In this paper we suggest a pruning method based on pruning a Huffman tree shaped WT according to the underlying *skeleton* Huffman tree [15]. This pruned WT is especially designed in order to support faster random access and save memory storage, at the price of less effective rank and select operations, as compared to the original Huffman shaped WTs. The general idea is to apply some pruning 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. We present experimental results comparing our method to the state of art, showing that skeleton tree based WTs are superior to Huffman shaped WTs, which suggests that if direct access based operations are done much more often than rank and select, the former trees may be a better choice. In addition, the space savings by our compact data structure outperforms other direct access based solutions used in practice, such as DACs and fixed length coding, while the penalty in processing time is reasonable.

Our paper is organized as follows. Section 2 discusses previous research dealing with random access to files encoded using variable length codes. Section 3 deals with random access to Huffman encoded files, using WTs especially adapted to Huffman compressed files. Section 4 improves the self-indexing data structure by pruning the WT using a skeleton Huffman tree. Section 5 evaluates the savings induced by the proposed data structure. Section 6 further improves the overhead storage by pruning the Wavelet tree even further by means of a

reduced skeleton tree. Section 7 then compares our suggested data structures to the state of art, and presents some experiments demonstrating that our data structures are competitive. Finally, Section 8 concludes.

2. Related Work

The choice of a code will be guided by the intended application and expected properties. Thus in many situations, FLC are used despite their storage inefficiency, because of the simplicity of processing encoded data. In other cases, although FLC are possible, it is preferable to use VLC for storage efficiency when storage is of main concern. This paper focuses on data structures supporting random access in efficient time, while using a compact representation of the underlying data.

Two famous VLC codes are due to P. Elias [6] who developed universal codes for encoding positive integers, known as Elias-γ and Elias-δ codes. To encode an integer $x \geqslant 1$ using Elias-γ, its standard binary representation without leading zeros, $B(x)$, is used. The length of $B(x)$ is $\lfloor \log x \rfloor + 1$ bits. $B(x)$ without its leading 1-bit is preceded by the unary encoding of its length (the unary code is composed of the codewords {1, 01, 001,...}). Thus, to represent a number x, Elias-γ uses $2\lfloor \log_2(x) \rfloor + 1$ bits. For example, the number 23 is encoded as 00001 0111 (the space, here and below, is inserted for clarity), since $B(23) = 10111$.

The Elias-γ code is used as a building block for the Elias-δ code. To represent an integer x, Elias-δ precedes $B(x)$ without its leading 1-bit by the Elias-γ encoding of its length, for a total of $\lfloor \log_2(x) \rfloor + 2\lfloor \log_2(\lfloor \log_2(x) \rfloor + 1 \rfloor + 1$ bits. For example, the number 23 is encoded as 00101 0111, as the Elias-γ encoding of 5, the number of bits in $B(23)$, is 00101. Elias-δ is asymptotically shorter than Elias-γ, however, for the first numbers, Elias-γ codewords are shorter, so for many distributions, especially when the first probabilities are significantly larger than the last ones, Elias-γ might give better compression performance.

Another VLC which serves as an alternative to Elias codes is based the on Fibonacci numbers as follows. The Fibonacci sequence is defined as

$$F(0) = 1, \quad F(1) = 2, \qquad \text{and} \qquad F(n) = F(n-1) + F(n-2) \quad \text{for} \quad n \geq 2.$$

Any integer B can be represented by a binary string of length r, $c_r c_{r-1} \cdots c_0$, such that $B = \sum_{i=0}^{r} c_i F(i)$. Constructing a unique representation for a given integer x is done by finding the largest Fibonacci number $F(r)$ smaller or equal to x, and then continuing recursively with $B - F(r)$. For example, $31 = 21 + 8 + 2$, so its binary Fibonacci representation would be:

21	13	8	5	3	2	1
1	0	1	0	0	1	0

As a result of this encoding procedure, there are never consecutive Fibonacci numbers in any of these sums, implying that in the corresponding binary representation, there are no adjacent 1s. This property can be exploited to devise an infinite code whose set of codewords

consists of the Fibonacci representations of the integers: to assure the code being UD, each codeword is prefixed by a single 1-bit, which acts like a comma and permits to identify the boundaries between the codewords. To facilitate the decoding, the codewords are then reversed to yield a prefix code: $\{11, 011, 0011, 1011, 00011, 10011, \ldots\}$.

The Fibonacci codeword for a integer n is about 44% longer than the minimal $\log_2 n$ bits needed for the standard binary representation using only the significant bits, but it is shorter than the $2\log_2 n$ bits needed for the corresponding Elias-γ code.

Rice coding depends on a parameter k. To encode a number x, $\frac{x-1}{2^k}$ is unary encoded, and followed by the k least significant bits of the binary representation of $x-1$. For example, if $k=2$, $x=9$ is encoded by 110 00, as the unary encoding of $\frac{8}{2}=2$ is 110, and the 2 lower bits of the binary representation of $x-1=8$ are 00. If most of the numbers are small, fairly good compression can be achieved. Rice coding is generally used to encode entropy in codecs for audio and video.

A possible solution to allow random access to VLCs 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 [7] replace every fixed length block of symbols by a codeword of a Huffman code built according to the frequency of occurrence of the blocks. Their idea is to represent T of size n as a sequence of $\lceil \frac{n}{\ell} \rceil$ macro-symbols over the macro-alphabet Σ^ℓ, where $\ell = \left\lceil \frac{\log_{|\Sigma|} n}{2} \right\rceil$. 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 [26] extends Moffat and Stuiver's work [20] on *Interpolative coding*, so that direct access and finding the position in which the prefix sum exceeds some threshold are both 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. Each internal node is the sum of its children, and the root has the total sum. In addition to the root, only the left children need to be encoded, because the right ones are obtained by subtraction. The root of the tree is encoded using some universal code. The encoding of other nodes is based on the knowledge that the node value is between 0 and the parent value. Thus, fixed-length binary coding, truncated to the code length of the parent value, can be used. The space is at most $n\log(1+\frac{s}{n}) + O(n)$ where s is the sum of the non-negative integers.

Brisaboa et al. [2] use a variant of a Wavelet tree on Byte-Codes. This induces an n-ary tree rather than a binary one, and the root of the Wavelet tree contains the first *byte*, rather 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 [2] to be better than explicit main memory inverted indexes built on the same collection of words, when little extra space on top of the compressed text is available.

In another work, Brisaboa et al. [3] introduced directly accessible codes (DACs) by integrating rank data structures into variable lengths codes. Their method is based on Vbyte coding [30], in which the codewords represent integers. The Vbyte code splits the $\lfloor \log x_i \rfloor + 1$ bits needed to represent an integer x_i in its standard binary form into blocks of b bits and prepends each block with a flag-bit as follows. The highest bit is 0 in the extended block holding the most significant bits of x_i. and 1 in the others. Thus, the 0 bits acts as a comma between codewords. For example, if $b = 3$, and $x_i = 25$, the standard binary representation of x_i, 11001, is split into two blocks, and after adding the flags to each block, the codeword is **0**011 **1**001. In the worst case, the Vbyte code loses one bit per b bits of x_i plus b bits for an almost empty leading block, which is worse than Elias-δ encoding. DACs can be regarded as a reorganization of the bits of 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 improve the space requirements by choosing different values of b for each level, possibly introducing more levels, and thus slower access time. They suggest different constructions of DACs taking into account trade offs of space and access time.

Külekci [19] 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, the usage of a WT 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, which is bounded by $\log \log M$.

Klein and Shapira [17] 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. The idea is based on the property of the Fibonacci code that all codewords, except the first one 11, terminate with the suffix 011. These suffixes are necessary to ensure the prefix property of the Fibonacci code, but some of the corresponding nodes in the Fibonacci Wavelet Tree are redundant. As the binary tree corresponding to the Fibonacci code is not complete, and we can eliminate all the nodes which are single children of their parents. The bitmaps corresponding to the remaining *internal* nodes of the pruned tree are the only information needed in order to achieve constant random access.

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. For small alphabets, like a typical distribution of English characters, the excess of Fibonacci versus Huffman encoding can be about 17% [8], and may be less, around 9%, on much larger alphabets [16]. It would therefore be interesting to apply a similar pruning approach to Huffman based

WTs and evaluate its compression and processing time savings, which motivated the current research.

Our research follows this trend of adapting the Wavelet tree to various coding schemes, but aims at improving both the time and space complexities, and we concentrate on Huffman codes. The expected improvement is based on exploiting specific features of these codes, which permit to reduce the size of the associated Wavelet trees. In the following sections, we bring some technical details on the necessary operations, as well as on skeleton trees, which will be useful for the understanding of the ideas below.

3. Skeleton shaped Wavelet trees

The binary tree T_C 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, T_C is defined as the binary tree for which the set of bit strings associated with its leaves is the code C.

A binary tree is called *canonical* if, when scanning its leaves from left to right, they appear in non-decreasing order of their depth. Labeling an edge pointing to the left or right child by 0 or 1, respectively, as mentioned above, this definition is equivalent to the property that when the codewords are sorted by the frequency of the symbols they encode, they are ordered lexicographically. To build a canonical tree, Huffman's algorithm is only used for generating the optimal lengths ℓ_i of the codewords, and the ith codeword then consists of the first ℓ_i bits immediately to the right of the "binary point" in the infinite binary expansion of $\sum_{j=1}^{i-1} 2^{-\ell_j}$, for $1 \leq i \leq n$ [10]. Canonical codes are used to save space and enhance processing time. Turpin and Moffat [27] use canonical codes to improve decoding in Huffman encoded texts, so that more than a single bit can be processed in one machine operation.

A *full* subtree is a subtree all of whose leaves are on the same level. *Pruning* canonical trees will refer here to the process of eliminating all the nodes which have an ancestor that is the root of a full subtree. The resulting pruned tree is called a *skeleton tree* [15], or sk-tree for short. More formally, an sk-tree is a canonical Huffman tree from which all full subtrees of depth $h \geq 1$ have been pruned. Thus, a path from the root to a leaf of a sk-tree may correspond to a prefix of several codewords of the original Huffman tree. The prefix is the shortest necessary in order to identify the length of the current codeword. A leaf, v, of the sk-tree contains the height, $h(v)$, of the subtree that has been pruned or $h(v) = 0$ for leaves that were also leaves in the canonical Huffman tree. Skeleton trees have been used to accelerate compressed pattern matching in [25].

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. Figure 1 depicts the canonical Huffman tree for the example text $T =$ `A--HUFFMAN--WAVELET--TREE--MATTERS`.

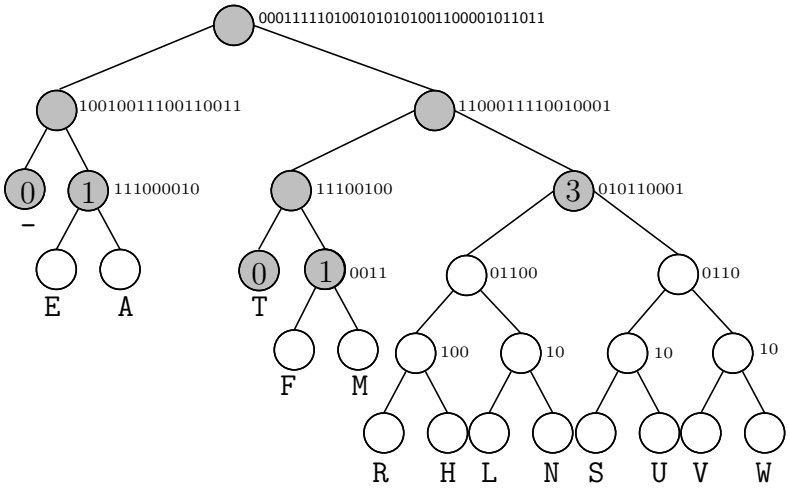


FIGURE 1: *The WT induced by the canonical Huffman tree built for* T = `A--HUFFMAN--WAVELET--TREE--MATTERS`, *corresponding to the frequencies* {*8,5,4,4,2,2,2,1,1,1,1,1,1,1*} of {`-,E,A,T,F,M,R,H,L,N,S,U,V,W`}, respectively, assigned to the leaves, left to right.

The WT of our running example is the entire figure including the annotating bitmaps. The sk-tree nodes are colored in gray, and the numbers $h(v)$ are given in the leaves of the sk-tree. It should be noted that the shape of the traditional WT is not restricted to the underlying canonical Huffman tree. For any distribution, there are many different Huffman trees, and for some distributions, there might even exist Huffman trees of different depths. Different topologies would imply different WTs and for convenience, we refer to the canonical one for the discussion in the next sections.

The algorithm for extracting the i-th element of the text T by means of a Huffman WT rooted by v_{root} is given in Algorithm 1, using the function call $\mathsf{extract}(v_{root},i)$. B_v denotes the bitmap belonging to vertex v of the WT, and $\cdot$ denotes concatenation. Computing the new index in the following bitmap is done by the rank operation in lines 2.1.3 and 2.2.3. The decoding of the codeword cw in line 3 by means of the decoding function $\mathcal{D}$ can be done by a preprocessed lookup table.

4. Enhanced Direct Access

The shape of the proposed WT is the sk-tree, for which the leaves contain the lengths of the corresponding remaining codeword suffixes. The internal nodes of the skeleton Huffman WT, which we shall call sk-WT, store the same bitmaps as the original WT. A leaf v of the sk-WT, for which $h(v) \geq 1$, stores the binary string obtained by the concatenation of the suffixes of length $h(v)$ of the codewords corresponding to v. That is, each such suffix appears the same number of times as the number of occurrences in T of the corresponding alphabet symbol $\sigma \in \Sigma$. Eliminating some additional nodes has a direct effect on improving both time and space performance.

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

ALGORITHM 1: *Extracting the i-th element of T from a WT rooted at v.*

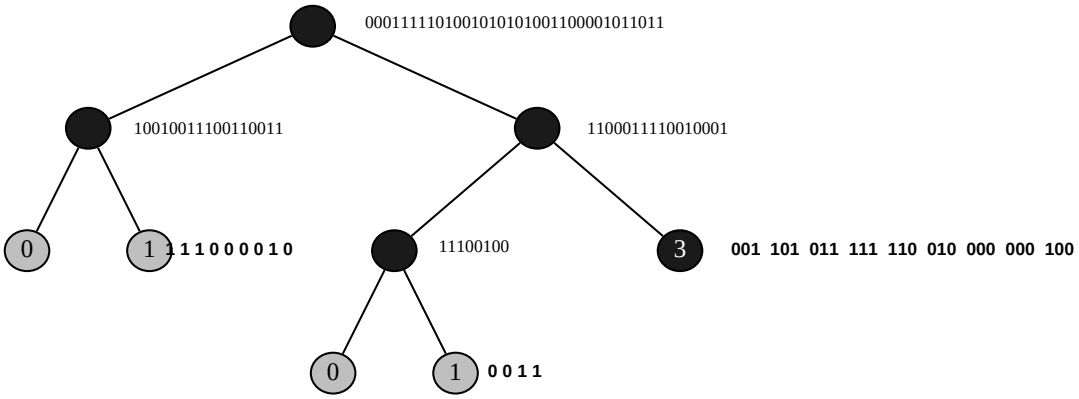


FIGURE 2: *Pruned Huffman WT for the text* $T =$`A--HUFFMAN--WAVELET--TREE--MATTERS`

Continuing with the running example, the resulting pruned WT is given in Figure 2. The leaf labeled 3 corresponds to the codewords {11000, 11001, 11010, 11011, 11100, 11101, 11110, 11111}, all sharing the same prefix 11. The list of their suffixes of length 3, in the order they appear in T, is 001 101 011 111 110 010 000 000 100, and their concatenation is the bitmap stored in the leaf labeled 3. A similar idea to this collapsing strategy is applied on suffix or position trees in order to attain an efficient *compacted* suffix trie [5], and has also been applied on Fibonacci Wavelet trees [17].

The algorithm for extracting the i-th element of T from a pruned Huffman WT requires some adjustments for concatenating the pruned parts. Algorithm 2 is the suitable extract function which is done as follows: first, the extracted codeword is initialized by the empty string, and the tree traversal starts at the root, going the way down until a leaf is reached. Branching left or right depends on whether the bit b at position i of the bitmap B in the current node is 0 or 1, respectively. At each such iteration, i is updated according to the value of $\mathsf{rank}_b(B, i)$, and b is concatenated to the end of the codeword. When a leaf is eventually reached in line

3.1, the fixed length suffix of size $h(v)$ bits is concatenated to the end of the codeword. The correct suffix can be accessed directly using the computed index i by simply extracting the substring of B_v starting at position $h(v)i$ and ending at position $h(v)(i+1)-1$. We use the notation $B[x\ :\ y]$ to denote the substring from position x to, and including, position y of a bit-string B.

---

extract(v, i)

1 $cw \longleftarrow \epsilon$
2 while v is not a leaf
2.1 if $B_v[i] = 0$ then
2.1.1 $cw \longleftarrow cw \cdot 0$
2.1.2 $i \longleftarrow \mathsf{rank}_0(B_v, i)$
2.1.3 $v \longleftarrow left(v)$
2.2 else
2.2.1 $cw \longleftarrow cw \cdot 1$
2.2.2 $i \longleftarrow \mathsf{rank}_1(B_v, i)$
2.2.3 $v \longleftarrow right(v)$
3 if $h(v) > 0$ then
3.1 $cw \longleftarrow cw \cdot B_v\big[h(v)i\ :\ h(v)(i+1)-1\big]$
4 return $\mathcal{D}(cw)$

ALGORITHM 2: *Extracting the i-th element of T from the pruned Huffman Wavelet Tree.*

Taking a closer look at our suggested data structure, the nodes that store the values $h(v)$ induce a partition of the alphabet into several equivalence classes. Some of these classes are singletons, while the others are of size 2^k for some $k > 0$. The skeleton Huffman tree does not have the ability to distinguish between elements of the same class. The following discussion refers to the select operation, however, a similar, yet easier approach could be applied in order to process the rank operation, which is shortly discussed later. Even though random access can be improved using sk-WTs rather than Huffman WTs, only partial information is attained when applying $\mathsf{select}(x, i)$ for retrieving the ith occurrence of x on our pruned data structure. Instead of returning the ith occurrence of x, x becomes a representative of its class, and the ith occurrence of elements which are in the same class as x is returned.

However, the classes are formed according to the probabilities of their elements, which does not necessarily imply any other connection. Nevertheless, whereas the exact values cannot be calculated using the original $\mathsf{select}(x, i)$ algorithm, this algorithm can still be used to derive a *lower bound* on the index of the i^{th} occurrence of x. If $\mathsf{select}(x, i) = j$, then the index of the i^{th} occurrence of x is $\geq j$. It is equal to j if all occurrences of elements belonging to the class of x correspond only to occurrences of x itself. If $\mathsf{extract}(v_{root}, j) \neq x$, a larger lower bound can be computed by applying select again with increasing i, until $\mathsf{extract}(v_{root}, j) = x$.

Although the select query cannot be answered in time proportional to the length of the codeword using the pruned WT, the exact value can still be derived iteratively. For example, finding the index of the *first* occurrence of x can be done in the following way: if $\mathsf{select}(x, 1) =$

j and extract$(v_{root}, j) = x$, the first occurrence of x is found at index j. If extract$(v_{root}, j) \neq x$, but select$(x, 2) = k$ and extract$(v_{root}, k) = x$, the first occurrence of x is found at index k. Otherwise the process continues until there exists some ℓ for which select$(x, \ell) = m$ and extract$(v_{root}, m) = x$. For larger i, the select(x, i) query on pruned WTs, Skeleton_rank, can be computed as follows:

```
Skeleton_select(x, i)
1     counter ⟵ 0; ℓ ⟵ 1; m ⟵ 0;
2     while counter < i and m ≤ n
3           m ⟵select(x, ℓ);
3.1         if extract(v_root, m) = x
3.1.1             counter++
3.2         ℓ++
4     return m
```

ALGORITHM 3: select(x, i) *on pruned WTs.*

The rank operation on pruned WTs, Skeleton_rank, also suffers from slower processing time as compared to the rank on regular WTs. When computing the traditional $rank(x, i)$ on a skeleton WT, the number of occurrences of all members of the class associated with x, up to and including position i, is attained. To get the exact number, one should scan linearly the fixed length suffixes, and count the number of occurrences of a known suffix of x in the bit vector associated with x, as shown in Algorithm 4. We use suff(x, ℓ) to denote the suffix of x of length ℓ. At line 2, the traditional rank is applied on the Skeleton WT, and the rank and the leaf it terminated at, is returned in variables num and v, respectively. The node v is used to determine the bit vector, B_v, that should be scanned, as well as the length of the suffix, $h(v)$, of x that all suffixes in B_v (of length $h(v)$ as well) should be compared to.

```
Skeleton_rank(x, i)
1     counter ⟵ 0;
2     (num, v) ⟵rank(x, i);
3     for j = 0 to num do
3.1         if B_v[j] =suff(x, h(v))
3.1.1             counter++
4     return counter
```

ALGORITHM 4: rank(x, i) *on pruned WTs.*

Let h denote the height of the skeleton WT. For a given i $(1 \leq i \leq n)$, the asymptotic processing time for extract(v, i) is $O(h)$, for Skeleton_rank(x, i) is $O(h + i)$, while the asymptotic time for Skeleton_select(x, i) is $O(h \times i)$. Empirical evaluation of these processing times is given in Section 7.

It should be noted that the negative impact of using the pruned WT on the rank and select queries is not as bad as it might seem on the first sight. The equivalence classes of the code-

words that have been pruned may be quite large, as can be seen, for example, in Figure 3 below, but the large classes correspond to the smaller probabilities. There is, of course, no knowledge about which elements will have to be retrieved, and we might be asked to perform a Skeleton_rank(x, i) or Skeleton_select(x, i) query for any x. Nonetheless, a reasonable assumption would be to assume that the appearance of codewords x in such queries will be according to their probability of occurrence in the text. In that case, the weighted average size of the equivalence classes will be quite small, so that an iterative search as suggested above is not such a burden. An indication for this asymmetric behavior of skeleton trees can be found by comparing the savings they imply on the space and time complexities: while the number of nodes can be reduced by 95% or more on large distributions, the weighted average path length for the same distributions is only shortened to about half, again because the short codewords correspond to the leaves with the higher probability. For example, in [15] a file of 500 MB (87 million words) of the *Wall Street Journal* [21] was considered. It was shown that while the total number of nodes in the Huffman tree was 289,101 and reduced to only 425 nodes in the corresponding skeleton tree, the average codeword length was only reduced from 11.2 to 5.7.

The extract operation is much easier to apply on fixed length codes than on variable length codes. In our pruned data structure, nodes v with $h(v) > 0$ store fixed length suffixes, hence, the improvement of the extract operation on our data structure over WTs for Huffman codes is clear. However, this is not the case when processing fixed length codes in order to locate and count the occurrences of a given codeword. Counting occurrences or locating the i^{th} occurrence of a given codeword in the pruned data structure requires to perform a rank or select operation on the fixed length suffixes stored in the leaves of the pruned WT. It seems, that if no auxiliary structure is used, then the rank and select queries must be performed sequentially, and the advantage of using fixed length suffixes disappears.

One could ask, therefore, whether rank and select queries can be done in a more efficient way for fixed length than for variable length codes. If this is the case, we can apply such a strategy on the fixed length suffixes of our data structure and support efficient rank and select queries as well, gaining faster processing time since the lengths of many of the codewords are shortened.

It is important to stress that our proposed data structure only reorders the bits in the bitmaps, implying the same bit counts in the full and pruned WTs, excluding auxiliary indexes. In our example, the 18 bits appearing in boldface in Figure 2 in the subtree rooted by the node labeled 3 are the same bits as those appearing in the bitmaps of the nodes in the corresponding subtree of Figure 1, that has been pruned. The savings of the sk-WT as compared to Huffman WTs of Section 2 stem thus from the fact that the rank and select data structures corresponding to the nodes are not all necessary for gaining the ability of direct access, because the bits corresponding to codeword suffixes are stored explicitly, and need not be extracted from bitmaps. The processing time is improved by accessing a smaller number of nodes.

5. Savings Analysis

To evaluate the savings induced by the pruning (restricting the analysis only to the rank function), we introduce the following notations. For an internal node v of the canonical Huffman tree, define $\mathsf{pref}(v)$ as the largest common prefix of all the codewords corresponding to this node. So, $\mathsf{pref}(root) = \Lambda$, denoting the empty string, and in Figure 2, if t is the node on level 3 annotated by the bitmap 0011, then $\mathsf{pref}(t) = 101$. Let C be the set of all the codewords. For a codeword $c \in C$ denote by $x(c)$ the corresponding character of the alphabet, and let $\mathsf{freq}(x(c))$ be the number of occurrences of x in the text. The length of the bitmap B_v stored at node v of the Wavelet Tree is then given by

$$|B_v| = \sum_{\{c \in C \mid \mathsf{pref}(v) \text{ is a prefix of } c\}} \mathsf{freq}\big(x(c)\big).$$

In particular, if v is the root, we get that $|B_v|$ is the sum of the frequencies of all the elements of the alphabet, which is equal to the length of the text in characters.

Summing the lengths of all the bitmaps in the WT gives the size, in bits, of the compressed file:

$$\text{Size of compressed file} = \text{lengths of all bitmaps} = \sum_{\{v \mid v \text{ is an internal node}\}} |B_v|.$$

Let $\mathcal{R}(n)$ denote the size of the data structures required by the rank function for a bitmap of size n. This could be $O(\frac{n \log\log n}{\log n})$ using Jacobson's implementation to allow constant time, and although this size is $o(n)$, it is still not negligible, even for very large n For example, if the size of the bit vector is $n = 2^{32}$, then the overhead is 0.66n using the following calculation: The rank answers are stored every $\log^2 n = 1024$ bits using $\log n = 32$ bits per sample, for a total of 2^{27} bits. The second level stores relative rank answers every $\frac{\log n}{2} = 16$ bits using $2 \log\log n = 10$ bits per subsample, using in total $\frac{10}{16} 2^{32}$ bits. The table, in this case, has 2^{16} entries, one for each of the possible 16-bit strings. For example, the entry indexed 42072 will contain 6, which is the the number of 1-bits in the binary representation 1010010001011000 of this index; the table entries are stored using $\log 16 = 4$ bits per entry for a total of $2^{16} \cdot 4 = 2^{18}$ bits, only for the exhaustive table. As alternative, $\mathcal{R}(n)$ can be reduced to $0.0625n$ using Vigna's implementation, at the price of increased processing time. The overall size, RSW, required by the rank structure of the original Wavelet Tree is thus

$$\text{RSW} = \sum_{\{v \mid v \text{ is an internal node}\}} \mathcal{R}(|B_v|).$$

When using the pruned version, the rank structures for the bitmaps corresponding to pruned subtrees are not needed, as well as the rank structures for the leaves of the skeleton tree. Denote by T_w the subtree rooted at the node w and by SKL the set of leaves of the sk-tree. The number of bits saved for the rank structures by the pruning process, RSW', is given

by

$$\text{RSW'} = \sum_{\{w \mid w \in \text{SKL} \wedge h(w) \geq 1\}} \sum_{\{v \mid v \in T_w\}} \mathcal{R}(|B_v|).$$

For example, for the tree in Figure 2, the outer summation refers to all the leaves of the sk-tree, which are the gray nodes labeled by the numbers $h(v)$. The inner summation goes over all the internal nodes, including the root of the subtree, which is left in the sk-tree.

It follows that the savings depend on the shape of the canonical tree and the corresponding sk-tree. In the worst cases, the skeleton tree yields no savings at all, but this happens only for highly skewed distributions implying a depth of $\Omega(|\Sigma|)$ for the Huffman tree, which is extremely rare for large alphabets. In general, the number of pruned nodes is substantial, and the overhead for the rank structures, RSW $-$ RSW', will be significantly smaller for the pruned version of the WT. Examples of numerical values on our data files are given below in the experimental section.

6. Reduced skeleton trees

Extending the pruning idea, we wish to prune the Huffman tree even more, possibly suggesting a tradeoff between space efficiency and processing time. However, it is not clear that processing time would be hurt by this further reduction, since less internal nodes will be processed. The idea is replacing the Skeleton tree topology of the WT by a *Reduced Skeleton tree* suggested in [15]. The Reduced Skeleton tree prunes the Skeleton Huffman tree at some internal node at which the length of the current codeword may only be partially determined. That is, when getting to a leaf of a Reduced Skeleton Tree, it is not necessarily possible to deduce the exact length of the current codeword, but some partial information is already available: the possible consecutive lengths belong to a set of size at most 2. The black nodes of Figure 2 are those from the Reduced Skeleton Tree of the WT presented in Figure 1.

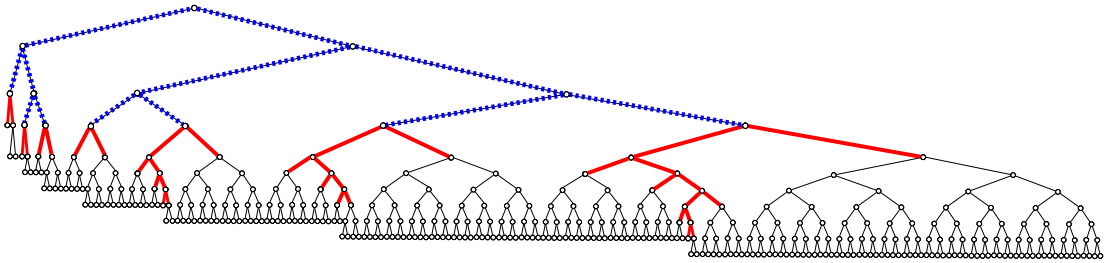


FIGURE 3: *Canonical Huffman tree, sk-tree (bold, red and blue) and reduced sk-tree (broken lines, blue) for 200 elements of a Zipf distribution, defined by the weights* $p_i = 1/(i\,H_n)$, *for* $1 \leq i \leq n$, *where* $H_n = \sum_{j=1}^{n}(1/j)$ *is the n-th harmonic number.*

As another example consider the canonical Huffman tree given in Figure 3. It corresponds to the probability distribution of $n = 200$ elements implied by Zipf's law [31], which is believed to govern the distribution of the most common words in a large natural language text. The bold (red or blue) edges are the corresponding sk-tree, and the subset of the bold edges, those with broken lines (blue), are the reduced sk-tree. For instance, when one gets to the

leaf of the reduced sk-tree corresponding to 110, one already knows that the codeword will be of length 8 or 9, so a single comparison suffices to decide it.

The algorithm for extracting the i-th element of T when the WT is constructed according to the reduced skeleton tree is similar to the algorithm presented earlier in Algorithm 2, and is given in Algorithm 5. We now need a *flag field* for each leaf v, with $flag(v) = 0$ if v is also a leaf in the skeleton Huffman tree (i.e., the length of the codeword is known when getting to this leaf while traversing the tree with an encoded string starting at the root; note that no leaf of the reduced sk-tree in Figure 3 has this property, but for other distributions, as the one presented in Figure 2, such leaves do exist), and $flag(v) = 1$ otherwise. In the latter case, the suffixes rooted at v are not of the same length, and we adjust the shorter suffixes to be of the length of the longer ones by padding them at their right end with a single 0. We then concatenate all these equal sized reconstructed suffixes in the same order as they appear in the text, as in skeleton WTs. The value $h(v)$ now stores the length of the suffix of the longer codeword if v is a leaf.

Random access to reduced sk-WTs is performed in a similar way to that of sk-WTs. When a leaf v is reached, the current suffix is initialized as having length $h(v)$. This is the correct setting when $flag(v) = 0$. Otherwise, the retrieved suffix is appended to the currently constructed codeword cw, and the value j represented by cw is compared with that of the first codeword of length $|cw|$. If j is smaller or equal, we know that the length of the codeword should be $|cw| - 1$, hence we remove the trailing 0 from the current codeword.

```
...
4    else // h(v) ≠ 0
4.1      cw ⟵ cw · B_v[h(v)i : h(v)(i + 1) − 1]
4.2      if flag(v) = 1 then
4.2.1        if cw ≤ first codeword of length |cw| then
4.2.1.1          remove trailing 0 from cw
5    return D(cw)
```

ALGORITHM 5: *Extracting the i-th element of T from a WT based on a reduced skeleton tree.*

7. Experimental Results

We considered six texts of different languages and alphabet sizes, encoded as sequences of *words* rather than of characters, so that our alphabets — and the corresponding Huffman trees — are quite large. More precisely, to create our vocabulary, we split the text into words (a maximal sequence of non whitespace characters) where white spaces were the only separators. We have not performed any additional pre-processing on the text. The datasets were as follows:

ebib is the Bible (King James version) in English, in which the text was stripped of all punctuation signs; *Einstein* is the collection of all the versions of the Wikipedia page about Albert Einstein in English; *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 [28]; *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; *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 *EnWiki-sml* is formed by downloading a dump of a prefix of the English Wikipedia. Our implementation used the *Succinct Data Structure Library* [11], which is an open-source library implementing succinct data structures efficiently in C++.

Table 1 presents some information on the data files involved. The second and third columns present the original file sizes in MB and millions of words. The fourth column gives the size of the alphabets in thousands of (different) words.

File	size	# of words	$\|\Sigma\|$
ebib	3.5	0.6	11
Einstein	446.0	62.3	23
ftxt	7.6	1.2	75
sources	200.0	25.8	2436
English	200.0	37.0	836
EnWiki-sml	65.1	12.7	282

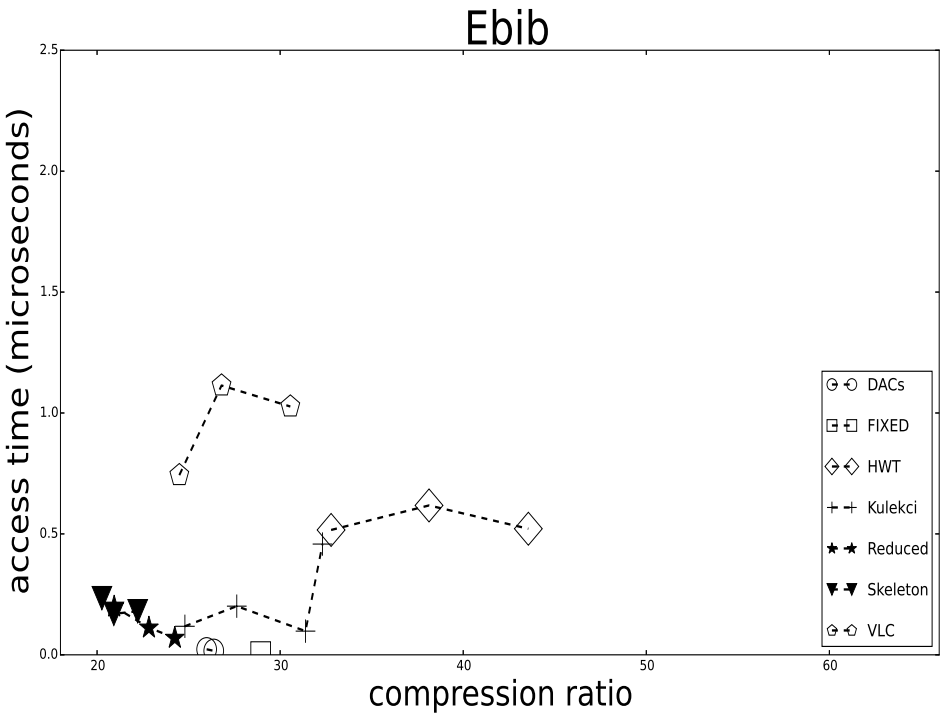
TABLE 1: *Information about the used datasets*

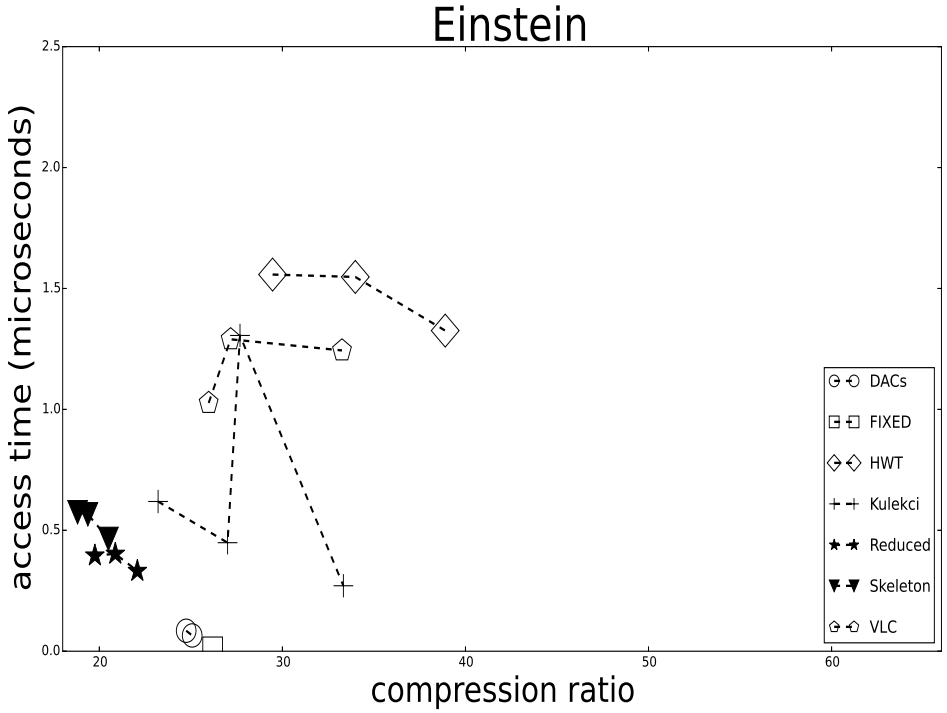
Table 2 presents the numerical values which were analytically derived in Section 4. The second column shows the lengths of all the bitmaps in the WT which is equal to $\sum_{\{v \mid v \text{ is an internal node}\}} |B_v|$. The third and forth columns give the sizes of the rank structures using Vigna's [29] and Gog's [12] implementations, respectively. Each column presents the sizes of the rank structures for the original WT, RSW, and the size saved by the pruning process, RSW'. All figures are given in MBs. As can be seen, the pruning yields about 50% gain due to the saving in the rank structures, which supports our theoretical evaluation.

File	$\sum_v(\|B_v\|)$	v		v5	
		RSW	RSW'	RSW	RSW'
ebib	0.6	0.16	0.09	0.04	0.02
Einstein	77.5	19.38	9.55	4.85	2.39
ftxt	1.5	0.37	0.19	0.09	0.05
sources	40.7	10.17	4.37	2.54	1.09
English	50.5	12.62	6.37	3.15	1.59
EnWiki-sml	16.5	4.12	2.08	1.03	0.52

TABLE 2: *Numerical values of our theoretical evaluation given in Section 5*

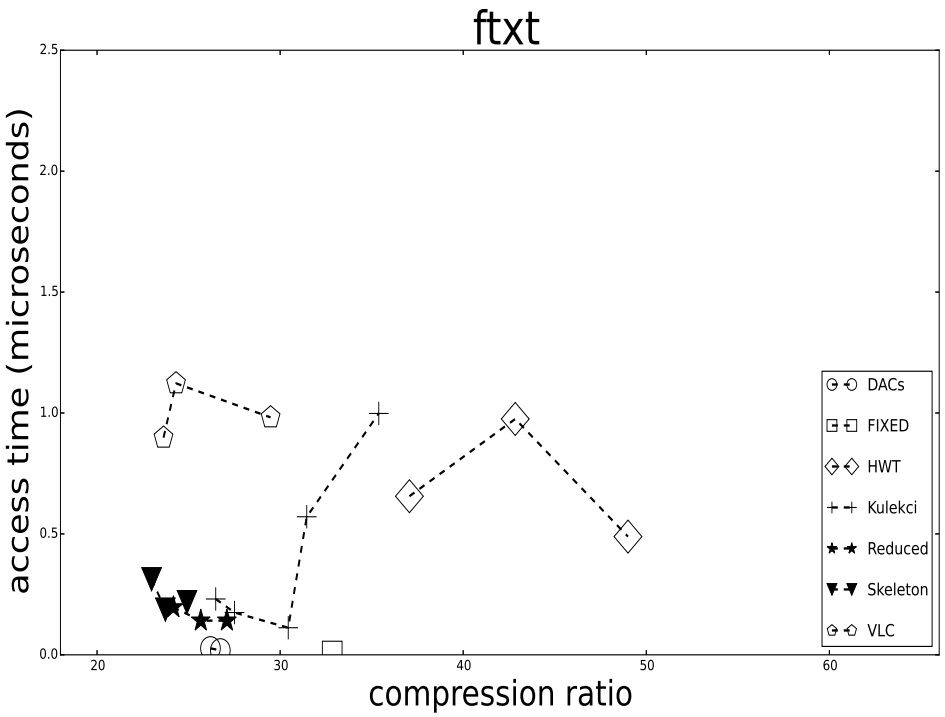
Figure 8(a-f) depicts, for each of the methods, the compression ratio and processing time as a single dot in a two-dimensional space. Compression is measured as the relative size, in percent, of the compressed file, including all necessary data structures, relative to the original file. The processing time is given in microseconds, averaging the time for retrieving a single word at 100,000 randomly chosen locations using the different methods. The experiments were conducted on a machine running 64 bit Linux Ubuntu with an Intel Core i7-4720 at 2.60GHz processor, 6144K L3 cache size of the CPU, and 4GB of main memory.

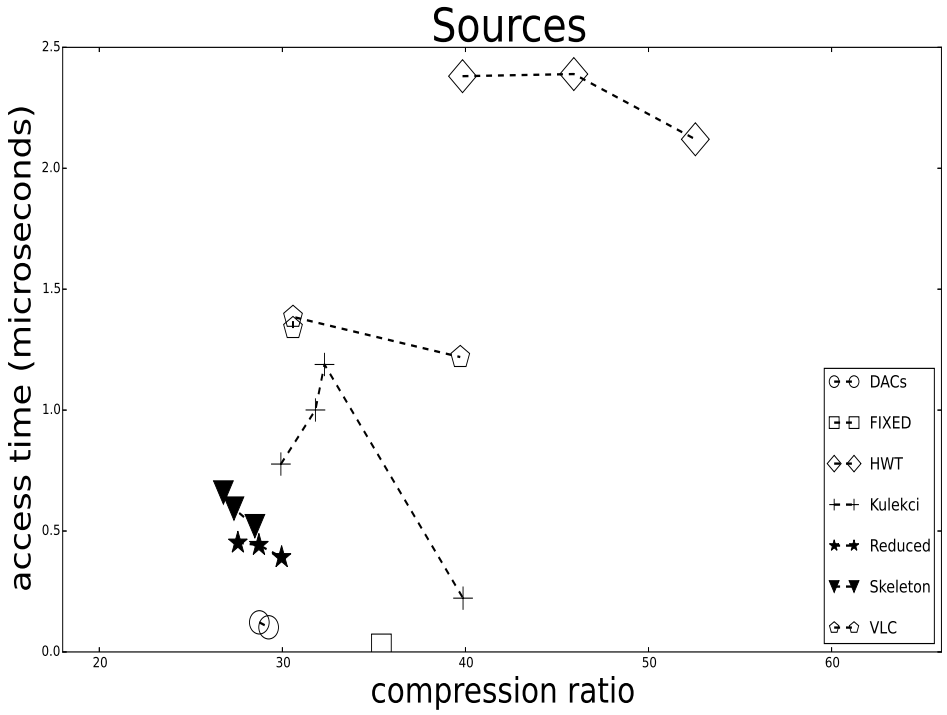




8(A-B)

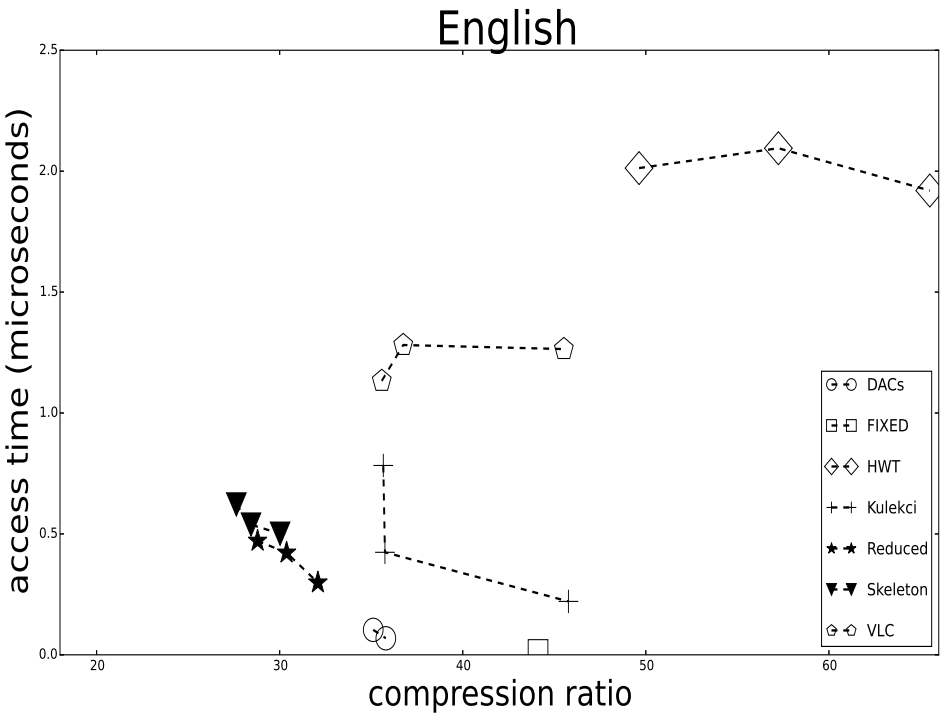
Several variants, usually based on different implementations, were checked for each method. To facilitate the reading of the results, each method is represented by a different symbol, the





8(C-D)

variants of a given method are represented by the same symbol, and identical symbols are connected in a systematic way. The following methods were compared, where the symbol



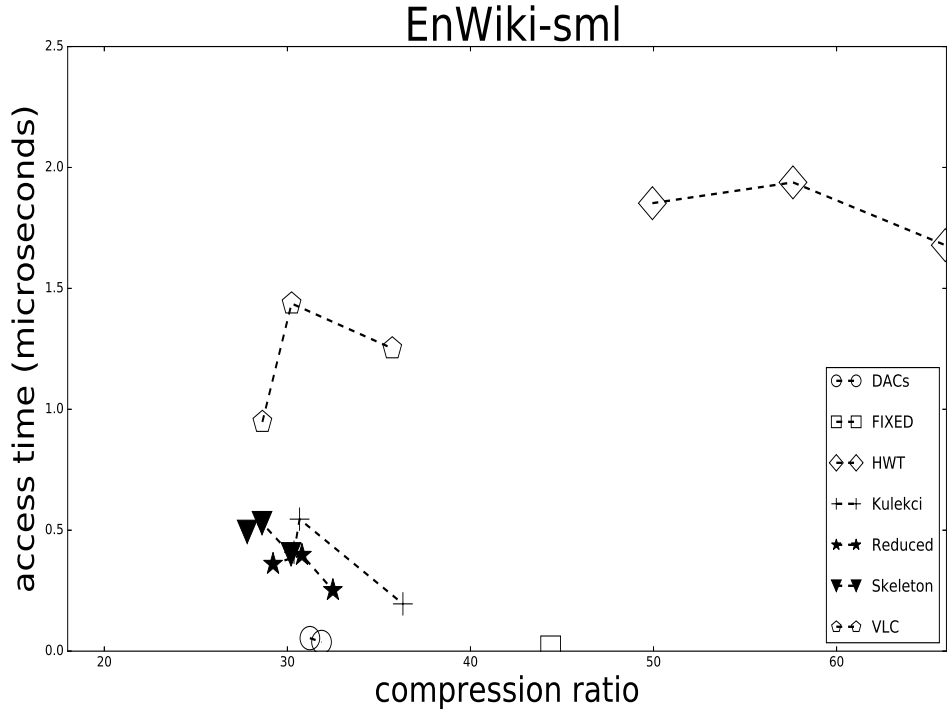


FIGURE 8(E-F)

used to visualize the method in the graphs is given in parentheses:

1. **DACs** (oval): using the best parameter b;

2. **fixed** (square): fixed length codes, without any Wavelet tree structure;
3. **HWT** (rhombus): the Huffman Wavelet Tree, implemented by using the Wavelet matrix [4];
4. **Külekci** (plus): given for both Elias-γ and Rice codings, for which the unary and binary parts of the codewords are treated separately;
5. **reduced** (star): reduced sk-WTs;
6. **skeleton** (triangle): regular sk-WTs;
7. **VLC** (pentagon): uses fixed size blocks of 128 symbols, encoded using self delimiting variable length codes.

For the methods which use rank (DACs, HWT, Skeleton, Reduced), three different implementations are used: the first and second approach of Vigna [29] and Gog and Petri's method [12] called interleaving, all providing a 64-bit implementation and producing different tradeoffs between time and space. The results appear from left to right (Gog, Vigna1, Vigna2) in the graphs for these four methods. The four variants for Külekci are, from left to right, Elias-γ with compressed bitmaps, Rice with compressed bitmaps, Elias-γ and Rice. The three variants of VLC are, from left to right, Fibonacci, Elias-δ coding and Elias-γ.

These implementations were adjusted to our sk-WT in order to support direct access alone. We therefore eliminated the data structures associated with the select operation; we also removed two pointers that are not needed when only going top-down the tree. In order to produce a fair comparison, we also removed the same elements from the Wavelet matrix used to implement HWT. Unlike the regular implementation of the WTs that associates each leaf with a single symbol of the alphabet, we have adjusted the implementation so that a leaf corresponds to several symbols. We used the property that each such leaf corresponds to several symbols, for which their corresponding fixed length suffixes are stored sequentially, and arranged the pruned symbols in an array sorted left to right. The symbol stored in a leaf that refers to a pruned subtree was then replaced by the starting index of the corresponding pruned characters of this array. This strategy could not be extended to the reduced sk-WT variant, since the extra 0 bit which was added to the shorter codewords caused the existence of nodes having only a single child, rather than all internal nodes having both children, as in the original implementation. We therefore shifted all leaves to the left, filling in the holes so they are consecutive.

As can be seen, the Skeleton Wavelet tree consistently achieves a significant compression improvement relative to the HWT, and always improves on the other two methods, which give only direct access. The time complexity is cut to about a third relative to Huffman, always better than the Külekci variants, but is of course slower than DACs and fixed length codes.

Although our pruned data structures are especially suited for enhancing direct access, it is interesting to empirically evaluate their processing times for rank and select, as discussed in Section 4. The results, in milliseconds, are presented in Table 3. Each result is the average of 100 runs. At each run a character, w, (a word in our case), and index i, were randomly

chosen, whereas w was uniformly picked from the text, and i was uniformly chosen over the amount of occurrences of w. The first column is the data file's name, followed by 4 columns for the rank and 4 columns for the select timing results. For each operation we present the performance of the traditional rank and select on a regular WT (WT), the performance of Algorithms 3 and 4 (Ours), the exhaustive rank and select applied on skeleton WTs (Ex-Sk), which checks each location sequentially for an occurrence, and rank and select on the appropriate fixed length code (FLC), which again performs a linear search starting from the first index and up to i.

As can be seen, the results are as expected. Regular WTs are superior to all methods when rank and select are considered. Obviously, FLC are better than exhaustive rank and select over WTs because of its constant direct access, unlike the $O(\log \Sigma)$ direct accesses in skeleton WTs. Nevertheless, Algorithms 3 and 4 are much faster than trivial rank and select performed by exhaustive searches on the pruned WT.

File	rank				select			
	WT	Ours	Ex-SK	FLC	WT	Ours	Ex-SK	FLC
ebib	0.002	0.018	3.126	0.242	0.004	46.098	251.94	12.087
Einstein	0.006	0.021	303.492	14.837	0.011	4033.133	18775.47	922.26
ftxt	0.002	0.004	0.2	0.012	0.005	74.674	438.521	20.394
sources	0.002	0.016	31.153	0.953	0.016	1086.177	9587.363	391.588
English	0.002	0.099	273.343	9.976	0.01	2197.97	15726	600.648
EnWiki-sml	0.002	0.074	30.06	1.543	0.009	704.922	3763.243	178.57

TABLE 3: rank *and* select *processing times comparison*

8. Conclusion

We have presented new data structures for reducing the space overhead of a Huffman shaped WT when used to support extract queries to the underlying text by means of a Skeleton Huffman tree, and their reduced variants. We give empirical evidence that the running time and compression performance is significantly improved as compared to the running time of the traditional HWT, since shorter paths from the root down to the leaves are processed. When storage is of main concern, our suggested data structure outperforms other direct access techniques such as those due to Külekci, DACs and sampling, with a slowdown as compared to DACs and fixed length encoding.

Acknowledgement: We would like to thank Simon Gog and M. Oguzhan Külekci for their help and for providing their implementations.

References

[1] J. BARBAY, T. GAGIE, G. NAVARRO, Y. NEKRICH, Alphabet partitioning for compressed

rank/select and applications, *Algorithms and Computation,* Lecture Notes in Computer Science LNCS, **6507** (2010) 315–326.

[2] N.R. Brisaboa, A. Fariña, S. Ladra, G. Navarro, Implicit indexing of natural language text by reorganizing bytecodes. *Information Retrieval*, **15**(6) (2012) 527–557.

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

[4] F. Claude, G. Navarro, The Wavelet Matrix, *Proc. 19th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS, **7608** (2012) 167–179.

[5] M. Crochemore, W. Rytter, *Jewels of Stringology*, World Scientific (2002).

[6] P. Elias, Universal codeword sets and representations of the integers, *IEEE Transactions on Information Theory* **21** (1975) 194–203.

[7] P. Ferragina, R. Venturini, A simple storage scheme for strings achieving entropy bounds, *Theoretical Computer Science* **372** (2007) 115–121.

[8] A.S. Fraenkel, S.T. Klein, Robust universal complete codes for transmission and compression, *Discrete Applied Mathematics* **64** (1996) 31–55.

[9] T. Gagie, G. Navarro, Y. Nekrich, Fast and compact prefix codes, *Proc. SOFSEM'10*, (2010) 419–427.

[10] E.N. Gilbert, E.F. Moore, Variable-length binary encodings, *The Bell System Technical Journal* **38** (1959) 933–968.

[11] S. Gog, T. Beller, A. Moffat, M. Petri, From theory to practice: plug and play with succinct data structures, *13th International Symposium on Experimental Algorithms, (SEA 2014)*, Copenhagen (2014) 326–337.

[12] S. Gog, M. Petri, Optimized succinct data structures for massive data, *Software, Practice and Experience*, **44**(11) (2014) 1287–1314.

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

[14] G. Jacobson, Space efficient static trees and graphs, *Proc. Foundations of Computer Science (FOCS)* (1989), 549–554.

[15] S.T. Klein, Skeleton trees for the efficient decoding of Huffman encoded texts, in the *Special issue on Compression and Efficiency in Information Retrieval* of the *Kluwer Journal of Information Retrieval* **3** (2000) 7–23.

[16] S.T. Klein, M. Kopel Ben-Nissan, On the usefulness of Fibonacci compression codes, *The Computer Journal* **53** (2010) 701–716.

[17] S.T. Klein, D. Shapira, Random access to Fibonacci encoded files, to appear in *The Discrete Applied Mathematics Journal, DAM.*

[18] S.T. Klein, D. Shapira, Enhanced extraction from Huffman encoded files, *The Prague Stringology Conference PSC–2015* (2015) 67–77.

[19] M.O. Külekci, Enhanced Variable-Length Codes: Improved compression with efficient random access, *Proc. Data Compression Conference DCC–2014*, Snowbird, Utah (2014) 362–371.

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

[21] A. Moffat, J. Zobel,N. Sharman, Text Compression for Dynamic Document Databases, *IEEE Trans. on Knowl. and Data Eng.*, **9**(2), (1997) 302–313.

[22] G. Navarro, E. Providel, Fast, small, simple rank/select on bitmaps, *Experimental Algorithms,* Lecture Notes in Computer Science (LNCS), **7276** (2012) 295–306.

[23] D. Okanohara, K. Sadakane, Practical entropy-compressed rank/select dictionary, *Proc. Workshop on Algorithm Engineering and Experiments, ALENEX*, New Orleans, Louisiana, (2007).

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

[25] D. Shapira, A. Daptardar, Adapting the Knuth-Morris-Pratt algorithm for pattern matching in Huffman encoded texts, *Information Processing and Management*, **42**(2) (2006) 429–439.

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

[27] A. Turpin, A. Moffat, Fast file search using text compression, *Proc. 20th Australasian Computer Science Conference,* Sydney (1997) 1–8.

[28] J. Véronis, P. Langlais, Evaluation of parallel text alignment systems: The arcade project, in *Parallel Text Processing*, J. Véronis, ed., Kluwer Academic Publishers, Dordrecht (2000) 369–388.

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

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

[31] G.K. Zipf, *The Psycho-Biology of Language*, Boston, Houghton (1935).