Enhanced Extraction from Huffman Encoded Files

Shmuel T. Klein[1] and Dana Shapira[2]

[1] Dept. of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel
tomi@cs.biu.ac.il

[2] Dept. of Computer Science, Ariel University, Ariel 40700, Israel
shapird@ariel.ac.il

Abstract. Given a file T, and the Huffman encoding of its elements, we suggest using a pruning technique for Wavelet trees that enables direct access to the i-th element of T by reordering the bits of the compressed file and using some additional space. When compared to a traditional Wavelet tree for Huffman Codes, our different reordering of the bits usually requires less additional storage overhead by reducing the need for auxiliary rank structures, while improving processing time for extracting the i-th element of T.

1 Introduction and previous work

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. One of the fundamental components of these structures is known as a *Wavelet tree*, suggested by Grossi et al. [11], which has meanwhile become a subject of investigation in its own right, as ever more of its useful properties are discovered [7]. It is on enhancing the usefulness of the extract operation of Wavelet trees when applied to Huffman encoded text that we wish to concentrate in this paper.

The simple way to encode our digital data is by using some standard fixed length code, like ASCII. This has many advantages, for example, allowing direct access to the ith codeword for any i, which might be useful when partial or parallel decoding is required. 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 to variable length codes 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 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.

Brisaboa et al. [4] introduced directly accessible codes (DACs), based on Vbyte coding [20], in which the codewords represent integers. The Vbyte code splits the

Shmuel T. Klein, Dana Shapira: Enhanced Extraction from Huffman Encoded Files, pp. 67–77.
Proceedings of PSC 2015, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05787-2

$\lfloor \log x_i \rfloor + 1$ 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 act as a comma between codewords. 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. Using a space overhead of $O(\frac{n \log\log n}{b \log n})$, DACs achieve direct access to the i^{th} codeword in $O(\frac{\log(M)}{b})$ processing time, where M is the maximum integer to be encoded, and n is the size of the encoded file.

Another line of investigation led to the development of Wavelet trees, which allow direct access to any codeword, and in fact recode the compressed file into an alternative form. Wavelet trees can be defined for any prefix code and the tree structure is inherited from the tree usually associated with the code. The internal nodes of the Wavelet tree are annotated with bitmaps. The root 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.

The data structures associated with a Wavelet tree for general prefix codes require some amount of additional storage (compared to the memory usage of the compressed file itself). Given a text string of length n over an alphabet Σ, the space required by Grossi et al.'s implementation can be bounded by $nH_h + O(\frac{n \log\log n}{\log_{|\Sigma|} n})$ 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.

Brisaboa et al. [3] used a variant of a Wavelet tree on Byte-Codes. This induces a 128 or 256-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 to be better than explicit main memory inverted indexes, built on the same collection of words, when using the same amount of space.

Külekci [15] 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 Wavelet tree 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.

Recently Klein and Shapira [14] adapted the Wavelet tree to Fibonacci Codes, so that in addition to supporting direct access to the Fibonacci encoded file, the compression savings when compared to the original Fibonacci compressed file are

increased. We use a similar approach in this paper and prune the traditional Wavelet trees for general prefix codes without losing the direct access property. The topology of the reduced Wavelet tree is a *Skeleton Huffman* tree suggested by Klein [13], so that there are fewer internal nodes, and shorter paths from the root to the leaves, resulting in better processing time and less memory storage. This compact representation of Huffman trees was also used for improving the processing time for compressed pattern matching [19].

The skeleton Huffman tree used herein groups alphabet symbols according to their frequencies. A similar, yet different, alphabet partition according to frequencies has already been suggested by Gagie et al. [8], who study the problem of efficient representation of prefix codes, under the assumption that the maximum codeword length is $O(w)$, where w is the length of a machine word. They divide the alphabet into frequent and rare characters according to their Huffman codeword length, and store information just for the frequent ones, while the rare ones are lexicographically sorted. Using a multiary Wavelet tree, constant time encoding and decoding is achieved for small enough alphabets, at the price of increasing the codeword length of the rare characters, hurting the optimality of the Huffman code. Our approach is designed for all sizes of alphabets and the optimality of the Huffman codewords is retained at the price of slower processing.

Another data structure based on partitioning the alphabet into group of characters of similar frequencies is due to Barbay et al. [1]. This data structure stores the text in $nH_0 + o(n)(H_0 + 1)$ bits and supports operations in worst-case time $O(\log\log|\Sigma|)$ and average time $O(\log H_0)$. The sequences of sub-alphabet identifiers are stored in a multiary Wavelet tree, while the subsequences corresponding to each group are stored in uncompressed format.

Many of the data structures mentioned above use efficient access to bit vectors 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:

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

Note that $\mathsf{rank}_{1-b}(B, i) = i - \mathsf{rank}_b(B, i)$, thus, only one of the two, say, $\mathsf{rank}_1(B, i)$ needs to be computed. However, for the select operation the structures for both $\mathsf{select}_0(B, i)$ and $\mathsf{select}_1(B, i)$ are necessary [16]. Jacobson [12] showed that rank, on a bit-vector of length n, can be computed in $O(1)$ time using $n + O(\frac{n \log\log n}{\log n}) = n + o(n)$ bits.

It is important to stress that the overhead $o(n)$ of the rank and select data structures for a bitmap of size, say, $n = 2^{32}$ is about $0.66n$, which is not at all negligible. We suggest to reduce the size of the Wavelet tree without hurting the direct access capabilities. Methods proposed in [10] suggest practical implementations for rank and select, reducing the storage overhead to merely a few percent, at the price of losing the constant time access but with only a negligible increase in processing time. By applying our suggested strategy, these implementations can further be improved.

The $\mathsf{select}_b(B, i)$ operation can be done by applying binary search on the index j so that $\mathsf{rank}_b(B, j) = i$ and $\mathsf{rank}_b(B, j-1) = i-1$. As for the constant time solution for select [5], the bitmap B is partitioned into blocks, similar to the solution for the rank operation. Other efficient implementations are due to Raman et al. [18], Okanohara and Sadakane [17], Barbay et al. [2] and Navarro and Providel [16]. We refer to the thesis of Clark [5] for more details.

The rest of the paper is organized as follows. Section 2 deals with random access to Huffman encoded files, using Wavelet trees especially adapted to Huffman compressed files. Section 3 improves the self-indexing data structure by pruning the Wavelet tree using a skeleton Huffman tree. Section 4 further improves the overhead storage by pruning the Wavelet tree even further by means of a reduced skeleton tree. Finally, Section 5 concludes.

2 Random Access to Huffman Encoded Files

Recall that 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.

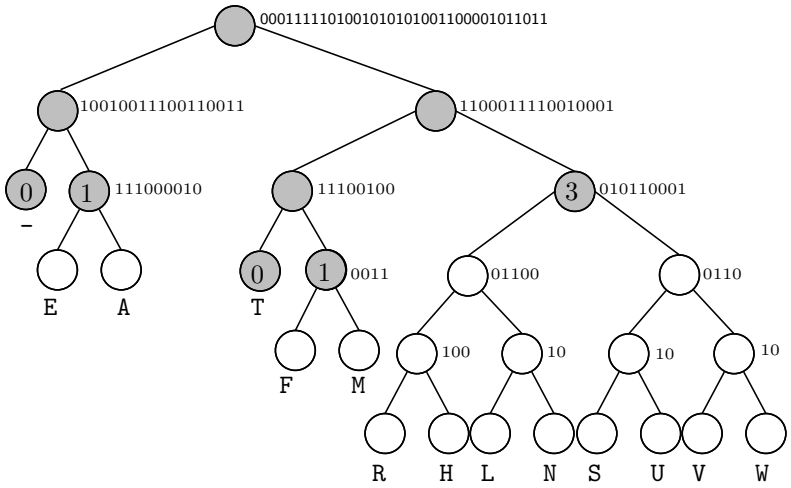

Figure 1. The Wavelet tree induced by the canonical Huffman tree corresponding to the frequencies {8,5,4,4,2,2,2,1,1,1,1,1,1,1} of {`-`,`E`,`A`,`T`,`F`,`M`,`R`,`H`,`L`,`N`,`S`,`U`,`V`,`W`}, respectively, assigned to the leaves, left to right.

A Huffman tree is *canonical* if, when scanning its leaves from left to right, they appear in non-decreasing order of their depth. To build a canonical tree, Huffman's algorithm is only used for generating the 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 $1 \leq i \leq n$ [9].

As mentioned above, the nodes of the Wavelet tree 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`. The Wavelet tree of our running example is the entire figure including the annotating bitmaps. It should be noted that the shape of the traditional Wavelet tree 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 Wavelet trees 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 Wavelet tree rooted by v_{root} is given in Figure 2, using the function call extract(v_{root},i). B_v denotes the bitmap belonging to vertex v of the Wavelet tree, 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.

```
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)
```

Figure 2. Extracting the i-th element of T from a Wavelet tree rooted at v.

3 Enhanced Direct Access

A *Skeleton* Huffman tree [13], or sk-tree for short, 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 an 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 skeleton tree contains the height, $h(v)$, of the subtree that has been pruned ($h(v) = 0$ for leaves that were also leaves in the canonical Huffman tree). In Figure 1, the sk-tree nodes are colored in gray, and the numbers $h(v)$ are given in the leaves of the sk-tree.

We adjust the Wavelet tree to Huffman skeleton codes in the following way. The shape of the Wavelet tree will be that of the sk-tree, to which the children of those nodes have been added, which were leaves in the sk-tree but not in the original Huffman tree, that is, the leaves v for which $h(v) \geq 1$. Bitmaps will be stored for the internal nodes of the Wavelet tree, as well as for the leaves that are children of leaves v of the sk-tree for which $h(v) > 1$, albeit the nature of these latter bitmaps will be different. The internal nodes will store the bitmaps as in the original Wavelet tree, whereas the annotated leaves will store the binary strings obtained by the concatenation of the suffixes of length $h-1$ of the corresponding codewords, in the same order as they appear in the compressed text. That is, each such suffix appears the same number of times as the number of occurrences of the corresponding alphabet symbol $\sigma \in \Sigma$ in T.

Continuing with the running example, the resulting pruned Wavelet tree is given in Figure 3. Consider the node labeled 3; it refers to the prefix 11 of several codewords, and the bitmap stored in it relates to the third bit of these codewords, which are all

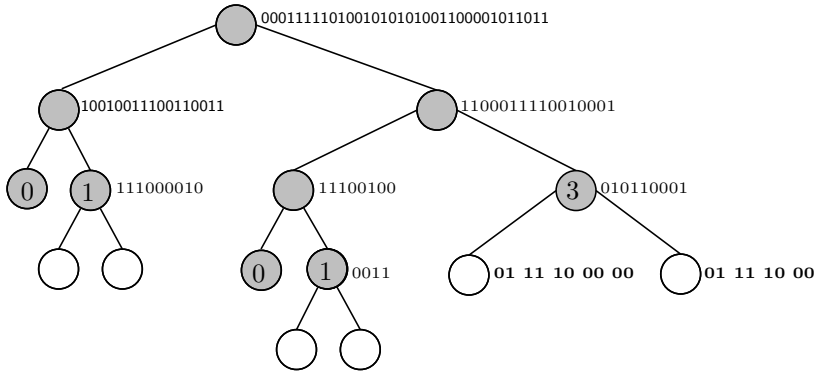

Figure 3. Pruned Huffman Wavelet tree for the text $T =$ `A--HUFFMAN--WAVELET--TREE--MATTERS`

of length 5. We thus eliminate the 3 bits that were already taken care of (110 for the left child and 111 for the right one), and consider only the remaining suffixes of size 2. In our example, the left child corresponds to the codewords {11000, 11001, 11010, 11011}, prefixed by 110 and refer to the symbols {`R,H,L,N`} of Figure 1, respectively. Their suffixes occur in the bitmap in the same order they appear in T, namely 01 11 10 00 00, corresponding to the order `HNLRR`. A similar idea to this collapsing strategy is applied on suffix or position trees in order to attain an efficient *compacted* suffix trie [6], and has also been applied on Fibonacci Wavelet trees, producing a compact Wavelet tree in [14].

The algorithm for extracting the i-th element of T from a pruned Huffman Wavelet tree requires some adjustments for concatenating the pruned parts. Figure 4 is the suitable extract function. Line 2.2.1 concatenates the fixed length suffix of size $h(v)-1$ bits 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 $(h(v)-1)i$ and ending at position $(h(v)-1)(i+1)-1$. 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 ⟵ ϵ
2     while v is not a leaf
2.1       if h(v) = 0 then
2.1.1         if B_v[i] = 0 then
2.1.1.1           v ⟵ left(v)
2.1.1.2           cw ⟵ cw · 0
2.1.1.3           i ⟵ rank_0(B_v, i)
2.1.2         else
2.1.2.1           v ⟵ right(v)
2.1.2.2           cw ⟵ cw · 1
2.1.2.3           i ⟵ rank_1(B_v, i)
2.2       else // h(v) ≠ 0
2.2.1         cw ⟵ cw · B_v[(h(v) − 1)i .. (h(v) − 1)(i + 1) − 1]
3     return D(cw)
```

Figure 4. Extracting the i-th element of T from the pruned Huffman Wavelet tree.

The following discussion refers to the select operation, however, a similar approach could be applied in order to answer the rank operation. Computing $\mathsf{select}(x, i)$ for selecting the i^{th} occurrence of x is done in the traditional Wavelet tree by processing

the tree upwards. One starts from the leaf, ℓ, representing the Huffman codeword $c(x)$ of x, initializes v to be the father of ℓ, and works its way up to the root. In each iteration, i is assigned a new value $\mathsf{select}_0(B_v, i)$ or $\mathsf{select}_1(B_v, i)$, depending on ℓ being a left or right child of v, respectively. The node v then proceeds to its father for the following stage. The running time for $\mathsf{select}(x, i)$ is $O(|c(x)|)$.

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. The skeleton Huffman tree does not have the ability to distinguish between elements of the same class. Thus, when applying $\mathsf{select}(x, i)$ on our pruned data structure, only partial information is attained. Instead of returning the i^{th} occurrence of x, x becomes a representative of its class, and the i^{th} 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 constant time using the pruned Wavelet tree, 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 $\mathsf{extract}(v_{root}, j) = x$, the first occurrence of x is found at index j. If $\mathsf{extract}(v_{root}, j) \neq x$, but $\mathsf{select}(x, 2) = k$ and $\mathsf{extract}(v_{root}, k) = x$, the first occurrence of x is found at index k. Otherwise the process continues until there exists some ℓ for which $\mathsf{select}(x, \ell) = m$ and $\mathsf{extract}(v_{root}, m) = x$. For larger i, the $\mathsf{select}(x, i)$ query can be computed as follows:

```
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++
```

It should be noted that the negative impact of using the pruned Wavelet tree on the select queries is not as bad as it might seem on the first sight. The equivalence classes of the codewords that have been pruned may be quite large, as can be seen, for example, in Figure 5 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 $\mathsf{select}(x, \ell)$ 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, see the examples in [13].

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 Wavelet trees 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 Wavelet tree. 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.

Note that the bits in the bitmaps stored in the leaves of the pruned Wavelet tree are the same as for the original Wavelet tree, only their order may have changed. In our example, the 18 bits appearing in boldface in Figure 3 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 pruned Huffman Wavelet tree as compared the original one 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. 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 prefix of all the codewords corresponding to this node. So, $\mathsf{pref}(root) = \Lambda$, denoting the empty string, and in Figure 1, if t is the node on level 3 annotated by the bitmap 0011, then $\mathsf{pref}(t) = 110$. 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 $\mathsf{freq}(x)$ 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 Wavelet tree 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 $O(\frac{n \log\log n}{\log n})$ to allow constant time, and although this size is $o(n)$, we mentioned above that it is still not negligible, even for very large n.

As alternative, $\mathcal{R}(n)$ can be reduced to $\frac{n}{20}$, 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. 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

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

For example, for the tree in Figure 4, the outer summation refers to all the leaves of the sk-tree, which are the gray nodes labeled by the numbers $h(v)$, but only for one node, the condition $h(v) > 1$ is satisfied. The inner summation goes over all the internal nodes, except the root of the subtree.

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 Wavelet tree.

4 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 would be processed. The idea is replacing the Skeleton tree topology of the Wavelet tree by a *Reduced Skeleton tree* suggested in [13]. The Reduced Skeleton tree prunes the Skeleton Huffman tree at some internal node at which the length of the current codeword is only partially determined. That is, when getting to a leaf of a Reduced Skeleton Tree, it is not yet possible to deduce the length of the current codeword, but some partial information is already available: the possible lengths belong to a set of size at most 2.

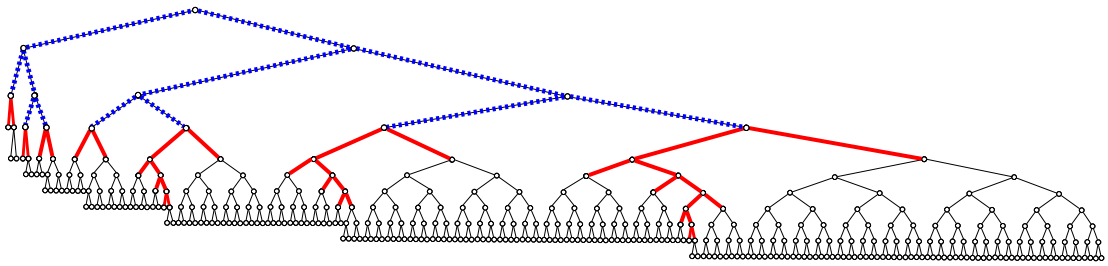

Figure 5. Canonical Huffman tree, sk-tree (bold, red and blue) and reduced sk-tree (broken lines, blue) for 200 elements of a Zipf distribution, defined by the weights $p_i = 1/(i\,H_n)$, for $1 \le i \le n$, where $H_n = \sum_{j=1}^{n}(1/j)$ is the n-th harmonic number.

Consider, for example, the canonical Huffman tree given in Figure 5. It corresponds to the probability distribution of $n = 200$ elements implied by Zipf's law [21], 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 Wavelet tree is constructed according to the reduced skeleton tree is similar to the algorithm presented earlier in Figure 4, and is given in Figure 6. 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 5 has this property, but for other distributions, 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 Wavelet trees. The value $h(v)$ now stores the length of the suffix of the longer codeword if v is a leaf, and 0 if v is an internal node.

When a leaf v is reached, the current suffix is initialized as having length $h(v)$. This is the correct setting when $flag(v) = 0$. When $flag(v) = 1$, we compare the integer value j obtained by using the retrieved suffix with that of the first codeword of length $|cw|$. If j is smaller or equal, we know that the length of the codeword is $|cw| - 1$, hence we remove the trailing 0 from the current codeword.

```
...
4     else // h(v) ≠ 0
4.1       cw ⟵ cw · B_v[(h(v) − 1)i . . (h(v) − 1)(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)
```

Figure 6. Extracting the i-th element of T from a Wavelet tree based on a reduced skeleton tree.

5 Conclusion

We have presented a new data structure for reducing the space overhead of a Huffman shaped Wavelet tree when used to support extract queries to the underlying text by means of a Skeleton Huffman tree. The running time is expected to be improved as compared to the running time of the traditional Wavelet tree, since shorter paths outgoing the root down to the leaves are processed. We intend to implement the pruned data structure and include experimental results in the full version of this paper.

References

1. J. Barbay, F. Claude, T. Gagie, G. Navarro, Y. Nekrich, Efficient Fully-Compressed Sequence Representations, *Algorithmica* **69**(1) (2014) 232–268.
2. 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.
3. N.R. Brisaboa, A. Fariña, S. Ladra, G. Navarro, Reorganizing compressed text, *Proc. of the 31th Annual Internetional ACM SIGIR Conference on Research and Developing in Information Retrieval (SIGIR)* (2008) 139–146.
4. N.R. Brisaboa, S. Ladra, G. Navarro, DACs: Bringing direct access to variable length codes, *Information Processing and Management*, **49**(1) (2013) 392–404.
5. D. Clark, Compact Pat Trees, Ph.D. Thesis, University of Waterloo, Canada, (1996).
6. M. Crochemore, W. Rytter, *Jewels of Stringology*, World Scientific (2002).
7. T. Gagie, G. Navarro, S.J. Puglisi, New algorithms on Wavelet trees and applications to Information Retrieval, *Theoretical Computer Science* **426** (2012) 25–41.
8. T. Gagie, G. Navarro, Y. Nekrich, Fast and Compact Prefix Codes. *Proc. SOFSEM'10*, (2010) 419–427.
9. E.N. Gilbert, E.F. Moore, Variable-length binary encodings, *The Bell System Technical Journal*, **38** (1959) 933–968.
10. R. González, S. Grabowski, V. Mäkinen, G. Navarro, Practical implementation of rank and select queries, *Poster Proceedings of 4th Workshop on Efficient and Experimental Algorithms (WEA05)*, Greece (2005) 27–38.
11. 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.
12. G. Jacobson, Space efficient static trees and graphs, *Proc. Foundations of Computer Science (FOCS)* (1989), 549–554.
13. 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.
14. S.T. Klein, D. Shapira, Random access to Fibonacci Codes, *The Prague Stringology Conference PSC-2014* (2014) 96–109.
15. M.O. Külekci, Enhanced Variable-Length Codes: Improved Compression with efficient random access, *Proc. Data Compression Conference DCC–2014*, Snowbird, Utah (2014) 362–371.
16. G. Navarro, E. Providel, Fast, small, simple rank/select on bitmaps, *Experimental Algorithms,* Lecture Notes in Computer Science (LNCS), **7276** (2012) 295–306.
17. D. Okanohara, K. Sadakane, Practical entropy-compressed rank/select dictionary, *Proc. ALENEX, SIAM* (2007).
18. R. Raman, V. Raman, S. Rao Satti, Succinct indexable dictionaries with applications to encoding k-ary trees and multisets, *Transactions on Algorithms (TALG)* (2007) 233–242.
19. D. Shapira, A. Daptardar, Adapting the Knuth-Morris-Pratt algorithm for pattern matching in Huffman encoded texts, *Information Processing and Management, IP & M* **42**(2) (2006) 429–439.
20. H.E. Williams, J. Zobel, Compressing integers for fast file access. *The Computer Journal* **42**(30) (1999) 192–201.
21. G.K. Zipf, *The Psycho-Biology of Language*, Boston, Houghton (1935).