Random Access to Fibonacci Encoded Files*

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

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

[b]*Department of Computer Science and Mathematics, Ariel University, Israel*
shapird@gmail.com

Abstract

A Wavelet tree is a data structure adjoined to a file that has been compressed by a variable length encoding, which allows direct access to the underlying file, resulting in the fact that the compressed file is not needed any more. We adapt, in this paper, the Wavelet tree to Fibonacci Codes, so that in addition to supporting direct access to the Fibonacci encoded file, we also increase the compression savings when compared to the original Fibonacci compressed file. The improvements are achieved by means of a new pruning technique.

Keywords: Fibonacci codes, Wavelet trees, rank and select

1. Introduction and previous work

Variable length codes, such as Huffman and Fibonacci codes, have been suggested long ago as alternatives to fixed length codes, since they might improve the compression performance. However, random access to the ith codeword of a file encoded by a variable length code is no longer trivial because the beginning position of the ith element is the sum of the lengths of all the preceding ones.

A possible solution to allow random access is to divide the encoded file into blocks of size b codewords, and to use an auxiliary vector to indicate the beginning of each block. The time complexity of random access depends on the size b, as we can begin from the sampled bit address of the $\frac{i}{b}$th block to retrieve the ith codeword. This method thus suggests a processing time

*This is an extended version of a paper that has been presented at the Prague Stringology Conference (PSC'14) in 2014, and appeared in its Proceedings, 96–109.

vs. memory storage tradeoff, since direct access requires decoding $i - \lfloor \frac{i}{b} \rfloor b$ codewords, i.e., less than b.

Consider for example the text $T = \texttt{COMPRESSORS}$ over the alphabet {`C, M, P, E, O, R, S`} of size 7, whose elements appear {1, 1, 1, 1, 2, 2, 3} times, respectively. The Fibonacci encoded file of length 39 bits is the following binary string, in which spaces have been added for clarity:

$$\mathcal{E}_{\mathit{fib}}(T) = 01011 \quad 0011 \quad 10011 \quad 00011 \quad 011 \quad 1011 \quad 11 \quad 11 \quad 0011 \quad 011 \quad 11.$$

block index	0	1	2	3
index of first bit in block	0	14	26	34

Table 1: Example of table of indices of 1-bits

If b, the number of codewords of each block, is equal to 3, the auxiliary vector A is initialized as shown in Table 1, whose first line indicates the indices of the block, and each cell on the second line records the address of the first bit of the corresponding block. In order to access the 8th character of T, one uses the cell of A indexed $\lfloor \frac{8-1}{3} \rfloor = 2$ to retrieve 26, and decodes $1 + ((8-1) \bmod 3) = 2$ codewords, to get to 11; the 8th character of T is thus $\mathcal{E}_{\mathit{fib}}(11) = \texttt{S}$.

Another line of investigation applies efficiently implemented rank and select operations on bit-vectors [28, 25] to develop a data structure called a *Wavelet Tree*, suggested by Grossi et al. [14], which allows direct access to any codeword, and in fact recodes the compressed file into an alternative form. The root of the Wavelet Tree holds the bitmap obtained by concatenating the *first* bit of each of the sequence of codewords in the order they appear in 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 with the grand-children of the root that hold the bitmap obtained by concatenating the *third* bit of the sequence of codewords; the fourth level nodes hold the *fourth* bit, and so on.

In this paper, we study the properties of Wavelet trees when applied to Fibonacci codes, and show how to improve the compression beyond the savings achieved by Wavelet trees for general prefix free codes. It should be noted that a Wavelet tree for general prefix free codes requires some amount of additional memory storage as compared to the memory usage of the compressed file itself. However, since it enables efficient direct access, this is a price one is often willing to pay. Wavelet trees, which are different implementations of compressed suffix arrays, yield a tradeoff between search time and memory storage. Given a string T of length n and an alphabet Σ, Grossi et al.'s implementation requires space $nH_h + O(\frac{n \log\log n}{\log_{|\Sigma|} n})$ bits, where H_h denotes the hth-order empirical entropy of the text, which is bounded by $\log|\Sigma|$, and processing time just $O(m \log|\Sigma| + \text{polylog}(n))$ for searching any pattern sequence of length m.

We concentrate on Wavelet trees for Fibonacci codes and suggest pruning the trees in order to save space, but without impairing their functionality. It has already been suggested in previous research to alter the shape of the Wavelet tree for different purposes. We mention here several of these suggestions.

Grossi and Ottaviano introduce the Wavelet *trie*, which is a compressed indexed sequence of strings in which the shape of the tree is induced from the structure of the Patricia trie [24]. This enables efficient prefix computations (e.g. count the number of strings up to a given index having a given prefix) and supports dynamic changes to the alphabet.

Brisaboa et al. [5] use a variant of a Wavelet tree on Byte-Codes. It encodes the sequence and provides direct access. 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 root has as many children as the number of different bytes (128 for End-Tagged Dense Codes (ETDC), since the first bit is reserved to indicate the end of a codeword). The second level nodes store the second byte of those codewords whose first byte corresponds to that child (in the same order as they appear in the text), 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. We use, in this paper, a binary Wavelet tree rather than a 128-ary one for byte-codes, using less space.

In another work, Brisaboa et al. [7] introduced directly accessible codes (DACs) by integrating rank dictionaries into byte aligned 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-bit 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 δ-Elias 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 comma bits for attaining $\frac{\log(M)}{b}$ processing time, where M is the maximum integer to be encoded. In the current work, not only do we use the Fibonacci encoding which is better than δ-Elias encoding in terms of memory space, we even eliminate some of the bits of the original Fibonacci encoding, while still allowing direct access with better processing time.

Recently, Külekci [23] suggested the usage of Wavelet trees and the rank and select data structures for *Elias* and *Rice* variable length codes. This 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 by two select queries. 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.

It should be noted that better compression can obviously be obtained by the optimal Huffman codes. The application field of the current work is thus restricted to those instances in which fixed, predefined codeword sets are preferred, for various reasons, to Huffman codes. These codes include, among others, the different Elias codes, dense codes like ETDC and (s, c)-Dense Codes (SCDC) [6], and Fibonacci codes.

The rest of the paper is organized as follows. Section 2 brings some technical details on Fibonacci codes. Section 3 deals with random access to Fibonacci encoded files, first suggesting the use of an auxiliary index, then showing how to apply Wavelet trees especially adapted to Fibonacci compressed files. Section 4 further improves the self-indexing data structure by pruning the Wavelet tree, and Section 5 extends the work to higher order Fibonacci codes. Finally, Section 6 brings experimental results, Section 7 describes time/space tradeoffs for rank and select, and Section 8 concludes.

2. Fibonacci Codes

The Fibonacci code is a universal variable length encoding of the integers based on the Fibonacci sequence rather than on powers of 2. A code is called *universal*, if the expected length of its codewords, for any finite probability distribution P, is within a constant factor of the expected length of an optimal code for P [9]. A finite prefix of the infinite sequence of Fibonacci codewords can be used as a fixed alternative to Huffman codes, giving obviously less compression, but adding simplicity (there is no need to generate a new code every time), robustness and speed [11, 21]. The particular property of the binary Fibonacci encoding is that it contains no adjacent 1's, so that the string 11 can act like a *comma* between codewords. More precisely, the codeword set consists of all the binary strings for which the substring 11 appears exactly once, at the left end of the string.

The connection to the Fibonacci sequence can be seen as follows: just as any integer k has a standard binary representation, that is, can be uniquely represented as a sum of powers of 2, $k = \sum_{i\geq 0} b_i 2^i$, with $b_i \in \{0,1\}$, there is another possible binary representation based on Fibonacci numbers, $k = \sum_{i\geq 2} f_i F_i$, with $f_i \in \{0,1\}$, where it is convenient to define the Fibonacci sequence here by

$$F_0 = 0, F_1 = 1 \qquad \text{and} \qquad F_i = F_{i-1} + F_{i-2} \quad \text{for } i \geq 2. \qquad (1)$$

This Fibonacci representation will be unique if, when encoding an integer, one repeatedly tries to fit in the largest possible Fibonacci number.

For example, the largest Fibonacci number fitting into 19 is 13, for the remainder 6 one can use the Fibonacci number 5, and the remainder 1 is a Fibonacci number itself. So one would represent 19 as $19 = 13 + 5 + 1$, yielding the binary string 101001. Note that the bit positions correspond to F_i for increasing values of i from right to left, just as for the standard binary representation, in which $19 = 16 + 2 + 1$ would be represented by 10011. Each such Fibonacci representation has a leading 1, so by preceding it with an additional 1, one gets a sequence of uniquely decipherable codewords.

Decoding, however, would not be instantaneous, because the set lacks the prefix property. For example, a first attempt to start the parsing of the encoded string 1101111111110 by 110 11 11 11 11 would fail, because the remaining suffix 10 is not the prefix of any codeword. So only after having read 5 codewords in this case (and the example can obviously be extended) would one know that the correct parsing is 1101 11 11 11 110. To overcome this problem, the Fibonacci code defined in [11] simply reverses each of the codewords. The adjacent 1s are then at the right instead of at the left end of each codeword, yielding the prefix code $\mathcal{E}_{fib}$ = {11, 011, 0011, 1011, 00011, 10011, 01011, 000011, 100011, 010011, 001011, 101011, 0000011,...}.

Since the set of Fibonacci codewords is fixed in advance, and the codewords are assigned by non-increasing frequency of the elements, but otherwise independently from the exact probabilities, the compression performance of the code depends on how close the given probability distribution is to one for which the Fibonacci codeword lengths would be optimal. The lengths are 2, 3, 4, 4, 5, 5, 5, 6, …, so the optimal (infinite) probability distribution would be $(\frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{16}, \frac{1}{32}, \frac{1}{32}, \frac{1}{32}, \frac{1}{64}, \ldots)$. 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 a typical distribution of English characters, the excess of Fibonacci versus Huffman encoding is about 17% [11], and may be less, around 9%, on much larger alphabets [21]. On the other hand, Fibonacci coding may be significantly better than other fixed codes such as Elias coding, ETDC and SCDC [21], as stated above.

3. Random Access to Fibonacci Encoded Files

3.1. Using an Auxiliary Index

We are given an alphabet Σ and a text $T = t_1 t_2 \cdots t_n$ of size n, where $t_i \in \Sigma$. Let $\mathcal{E}(T) = \mathcal{E}_{fib}(T)$ be the encoding of T using the first $|\Sigma|$ codewords of the Fibonacci code. $\mathcal{D}$ is the decoding that corresponds to $\mathcal{E}$ so that $\mathcal{D}(\mathcal{E}(T)) = T$.

A trivial solution for gaining random access to a Fibonacci encoded file $\mathcal{E}(T)$ is to create an auxiliary bitmap B of size $|\mathcal{E}(T)|$ indicating the codeword boundaries, e.g., by setting $B[i] = 1$ if and only if $\mathcal{E}(T)[i]$ is the first bit of a codeword. The following statement can then be used in order to extract t_i, the character in the ith position of T, using the select command of the succinct data structures for B mentioned above:

```
extract$(T, i)$
    return $\mathcal{D}\big(\mathcal{E}(T)[\mathsf{select}_1(B, i)..\mathsf{select}_1(B, i+1) - 1]\big)$
```

The select operation will return the position of the ith and $(i + 1)$st occurrences of a 1-bit in $\mathcal{E}(T)$. The decoding function is then used to decode the substring corresponding to these returned positions.

In the suggested solution, the space used to accomplish constant time rank and select operations, *excluding* the encoded file, is $u + o(u)$, where $u = |\mathcal{E}(T)|$ denotes the length of the encoded file. A better approach would be to omit the bitmap B of the first implementation and rather embed the index into the Fibonacci encoded file. This can be accomplished by treating two consecutive 1 bits in $\mathcal{E}(T)$ as a single 1-bit in B, and other bits in $\mathcal{E}(T)$ as a 0 in B. The storage overhead is therefore reduced to $o(u)$. Even better solutions are presented below.

3.2. Using Wavelet Trees

We adjust the Wavelet tree to Fibonacci codes in the following way. The Wavelet tree is in fact a set of annotations to the nodes of the binary tree corresponding to the given prefix code. These annotations are bitmaps, which together form the encoded text, though the bits are reorganized in a different way to enable the random access. The exact definition of the stored bitmaps, and some technical details on the rank and select operations are given in the following subsection, followed by a detailed description of Wavelet trees adapted to Fibonacci codes.

3.2.1. Rank and Select

Given a bit vector B and a bit $b \in \{0, 1\}$,

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

$\mathsf{select}_b(B, i)$ – 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 $\mathsf{select}_1(B, i)$ need to be stored [25]. Jacobson [19] 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.

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, the bitmap B is partitioned into blocks, similar to the solution for the rank operation. For simplicity, let us assume that $b = 1$. The case in which $b = 0$ is dealt with symmetrically. We refer to the work of [8] in more details, in which B is partitioned into blocks of two kinds, each containing exactly $\lceil \log n \rceil \lceil \log\log n \rceil$ 1s. The first kind are the blocks that are long enough (longer than $\lceil \log n \rceil^2 \lceil \log\log n \rceil^2$ bits) to store all their 1-positions within sublinear space. These positions are stored explicitly using an array, in which the answer is read from the desired entry i. The second kind of blocks are those we call *short*, of size $O(\log^c n)$, where c is a constant. Recording the 1-positions inside them requires again only subliniear space by repartitioning these blocks, and storing their relative position. The remaining blocks are short enough to be handled in constant time using a precomputed exhaustive table, and thus use three levels of auxiliary directories.

Okanohara and Sadakane [26] introduce four practical rank and select data structures, with different tradeoffs between memory storage and processing time. The difference between the methods is based on the treatment of sparse sets and dense sets. Although their methods do not always guarantee constant time, experimental results show that these data structures support fast query results and their sizes are close to the zero order entropy. Barbay et al. [2] propose a data structure that supports rank in time $O(\log\log|\Sigma|)$ and select in constant time using $nH_0(T) + o(n)(H_0(T) + 1)$ bits.

Navarro and Providel [25] present two data structures for rank and select that improve the space overheads of previous work. One using the bitmap in plain form and the other using the compressed form. In particular, they concentrate on improving the select operation since it is less trivial than rank and requires the computation of select_0 and select_1, unlike the symmetrical nature of rank. The memory storage improvement is achieved by replacing the exhaustive tables of [28]'s implementation by on-the-fly generation of their cells. In addition, they combine the rank and select samplings instead of solving each operation separately, so that each operation uses its own sampling, possibly using also that of the other operation.

3.2.2. Fibonacci Adapted Wavelet Trees

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 bitstring 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 bitstrings associated with its leaves is the code C. Figure 1 is the tree corresponding to the first 7 elements of the Fibonacci code. Since the bitmaps used by the Wavelet tree algorithms use the tree T_C as underlying structure, we shall refer to this tree as the Wavelet tree, for the ease of discourse.

The bitmaps in the nodes of the Wavelet tree 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 u of the text is given in the header of the file. We shall henceforth refer to the Wavelet

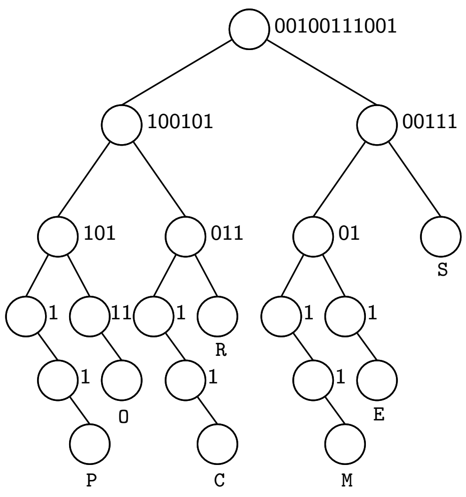

FIGURE 1: *Fibonacci Wavelet Tree for the text* $T =$ COMPRESSORS.

tree built for a Fibonacci code as the *Fibonacci Wavelet tree* (FWT). They are related, but not quite identical, to the trees defined by Knuth [22] as Fibonacci trees. We shall come back to the differences between FWTs and Knuth's Fibonacci trees later.

The FWT of our running example, including the annotating bitmaps, is given in Figure 1. Recall that the Fibonacci encoding of the sample string $T =$ COMPRESSORS is $\mathcal{E}_{fib}(T) =$ 01011 0011 10011 00011 011 1011 11 11 0011 011 11. The bitmaps stored in the nodes of the FWT are in fact a very specific reordering of the bits of the encoded file. The bitmap stored in the root consists of eleven bits 00100111001, one for each of the characters of T, and more specifically, the bitmap is the concatenation of the *first* bits of the eleven codewords in the encoding of T. These codewords are then partitioned into those starting with a 0-bit, in positions 1, 2, 4, 5, 9, and 10, and those starting with a 1-bit, in the other positions. The root's left child then refers to the six codewords starting with a 0-bit. Collecting the *second* bits of these codewords in the order they appear in the sample text, results in the bitmap 100101, which is stored in the root's left child. Similarly, the *second* bits of the five codewords starting with 1 are concatenated to yield 00111, which is stored in the right child of the root. This process of splitting the set of codewords corresponding to some node into two sub-sets that are assigned to the node's children, and collecting the i-th bit of the codewords for nodes on level i, continues for all internal nodes.

The Wavelet tree for $\mathcal{E}(T)$ is a succinct data structure for T as it takes space asymptotically equal to the Fibonacci encoding of T, and it enables accessing any symbol t_i in time $O(|\mathcal{E}(t_i)|)$, where $\mathcal{E}(x)$ is the Fibonacci encoding of the symbol x, under the assumption of a constant time rank implementation.

The algorithm for extracting t_i from an FWT rooted by v_{root} is given in Figure 2 using the function call extract(v_{root},i). B_v denotes the bit vector belonging to vertex v of the Wavelet tree, and $\cdot$ denotes concatenation. Computing the new index in the following bit vector is done by the rank operation, given in lines 3.3 and 4.3. As the Fibonacci code is a fixed one, the decoding of *code* in line 5 is done by a preprocessed lookup table.

```
extract($v, i$)
1     $code \longleftarrow \epsilon$
2     while $v$ is not a leaf
3           if $B_v[i] = 0$
3.1               $v \longleftarrow left(v)$
3.2               $code \longleftarrow code \cdot 0$
3.3               $i \longleftarrow \mathsf{rank}_0(B_v, i)$
4           else
4.1               $v \longleftarrow right(v)$
4.2               $code \longleftarrow code \cdot 1$
4.3               $i \longleftarrow \mathsf{rank}_1(B_v, i)$
5     return $\mathcal{D}(code)$
```

FIGURE 2: *Extracting t_i from a Fibonacci Wavelet Tree rooted at v_{root}.*

We extend the definition of $\mathsf{select}_b(B, i)$, which was defined on bitmaps, to be defined on the text T for general alphabets, in the obvious way. More precisely, we use the notation $\mathsf{select}_x(T, i)$ for returning the position of the ith occurrence of the symbol x in T.

Computing $\mathsf{select}_x(T, i)$ is done in the opposite way of the computation of rank. We start from the leaf, ℓ, representing the Fibonacci codeword $\mathcal{E}(x)$ of x, and work our way up to the root. The formal algorithm is given in Figure 3. The running time for $\mathsf{select}_x(T, i)$ is, again, $O(|\mathcal{E}(x)|)$, assuming a constant time select implementation.

4. Enhanced Wavelet Trees for Fibonacci codes

In this section, we suggest to prune the Wavelet Tree, so that the attained pruned Wavelet Tree still achieves efficient rank and select operations, and

```
select$_x(T, i)$
1      $\ell \longleftarrow$ leaf corresponding to $\mathcal{E}(x)$
2      $v \longleftarrow$ father of $\ell$
3      while $v \neq v_{root}(T)$
3.1        if $\ell$ is a left child of $v$
3.1.1          $i \longleftarrow$ select$_0(B_v, i)$
3.2        else // $\ell$ is a right child of $v$
3.2.1          $i \longleftarrow$ select$_1(B_v, i)$
3.3        $v \longleftarrow$ father of $v$
4      return $i$
```

FIGURE 3: Select *the ith occurrence of x in T.*

even improves the processing time. The proposed compressed data structure not only provides efficient random access capability, but also improves the compression performance as compared to the original Wavelet Tree.

4.1. Pruning the tree

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 FWT are redundant. As can be seen, e.g., in the example in Figure 1, 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. A similar idea to this collapsing strategy is applied on suffix or position trees in order to attain an efficient *compacted* suffix trie. This has also been applied on Huffman trees [20] producing a compact tree for efficient use, such as compressed pattern matching [29]. Applying this strategy on the FWT of Figure 1 results in the Pruned Fibonacci Wavelet Tree, PFWT for short, given in Figure 4.

The select$_x(T, i)$ algorithm for selecting the ith occurrence of x in T is the same as in Figure 3, gaining faster processing time since the lengths of the longer codewords were shortened. However, the algorithm for extracting t_i from a PFWT requires minor adjustments for concatenating the pruned

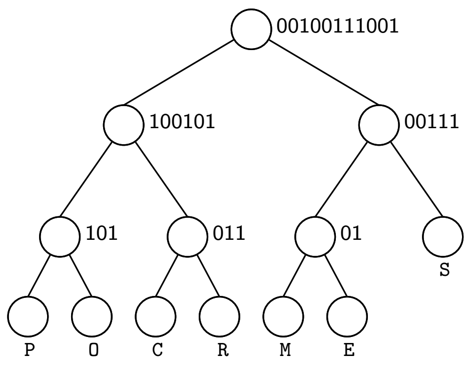

FIGURE 4: *PFWT for the text $T =$ COMPRESSORS.*

parts. The lines of Figure 5 should be added instead of line 5 in the algorithm of Figure 2.

```
5     if suffix of $code = 0$
5.1       $code \longleftarrow code \cdot 11$
6     if suffix of $code \neq 11$
6.1       $code \longleftarrow code \cdot 1$
7     return $\mathcal{D}(code)$
```

FIGURE 5: *Extracting t_i from the PFWT.*

The FWT of an alphabet of finite size is well defined and fixed. Therefore, only the size of the alphabet is needed for recovering the topological structure of the tree, as opposed to Huffman Wavelet Trees. Recall that the Wavelet tree for general prefix free codes is a reorganization of the bits of the underlying encoded file. The suggested pruned Fibonacci Wavelet tree uses only a partial set of the bits of the encoded file. The main savings of PFWTs as compared to the original FWTs of Section 3 stems from the fact that the bitmaps corresponding to the nodes are not all necessary for gaining the ability of direct access. These non-pruned nodes, therefore, are in a one-to-one correspondence with the bits of the encoded Fibonacci file. The bold bits of Figure 6 correspond to those bits that should be encoded; the others can be removed when we use the PFWT.

T	C					O				M					P				
$\mathcal{E}_{fib}(T)$	**0**	**1**	**0**	1	1	**0**	**0**	**1**	1	**1**	**0**	**0**	1	1	**0**	**0**	**0**	1	1

R			E				S		S		O				R			S	
0	**1**	**1**	**1**	**0**	**1**	1	**1**	**1**	**1**	**1**	**0**	**0**	**1**	1	**0**	**1**	**1**	**1**	**1**

Figure 6: *The Bitmap Encoding*

4.2. Analysis

We now turn to evaluate the number of nodes in the original and pruned FWTs, from which the compression savings can be derived. Two parameters have to be considered: the number of nodes in the trees, which relate to the storage overhead of applying the Wavelet trees, and the cumulative size of the bitmaps stored in them, which is the size of the compressed file. A certain codeword may appear several times in the compressed file, but will be recorded only once in the FWT.

Since we are interested in asymptotic values, we shall restrict our discussion here to prefixes of the Fibonacci code corresponding to full levels, that is, since the number of codewords of length $h+1$ is a Fibonacci number F_h [21], we assume that if the given tree is of depth $h+1$, then all the F_h codewords of length $h+1$ are in the alphabet. This restricts the size n of the alphabet to belong to the sequence 1, 2, 4, 7, 12, 20, 33, etc., or generally $n \in \{F_h - 1 | h \geq 3\}$.

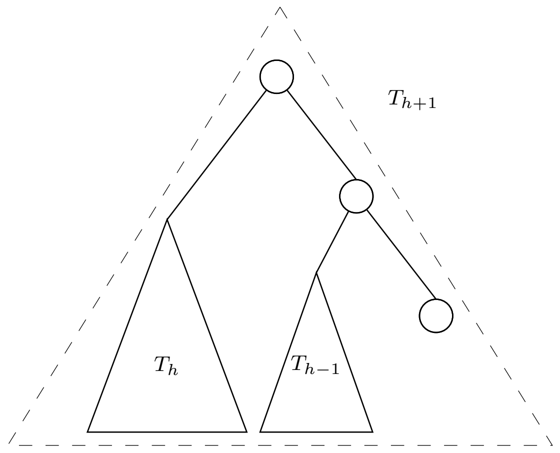

Figure 7: *Recursive definition of a FWT of height $h+1$.*

There are two ways to obtain the FWT of height $h+1$ from that of height h. The first is to consider the defining inductive process, as given in Figure 7. The left subtree of the root is the FWT of height h, while the right subtree of the root consists itself of a root, with a left subtree being the FWT of height $h-1$, and the right subtree being a single node. Denote by N_h the number of nodes in the FWT of height h, we then have $N_0 = N_1 = 0$ and

$$N_{h+1} = N_h + N_{h-1} + 3 \qquad \text{for } h \geq 1. \tag{2}$$

Note that the Fibonacci tree by Knuth [22] is based on a similar recursion, but with a different layout: the right subtree of the root of T_{h+1} would be T_{h-1}.

The second way to derive N_{h+1} is by adding the paths corresponding to the F_h longest codewords (of length $h+1$) to the tree for height h. This is done by referring to the nodes on level $h-2$ which have a single child, and there are again exactly F_h such nodes. The single child of these nodes corresponds to the bit 1, and their parent nodes are extended by adding trailing outgoing paths corresponding to the terminating string 011, turning each of them into a node with two children. For example, the gray nodes in Figure 8 are the FWT of height $h=4$. The three darker nodes are those on level 2 which are internal nodes with only one child. In the passage to the FWT of height $h+1=5$, the bold edges and nodes (representing the suffix 011) are appended to these nodes. This yields the recursion

$$N_{h+1} = N_h + 3F_h. \tag{3}$$

Applying eq. (3) repeatedly gives

$$N_{h+1} = N_{h-1} + 3(F_{h-1} + F_h) = N_{h-2} + 3(F_{h-2} + F_{h-1} + F_h),$$

and in general after k stages we get that

$$N_{h+1} = N_{h-k} + 3(\sum_{i=h-k}^{h} F_i).$$

When substituting $h-k$ by 2 we get that

$$N_{h+1} = N_2 + 3(\sum_{i=2}^{h} F_i).$$

By induction it is easy to show that

$$\sum_{i=2}^{h} F_i = F_{h+2} - 2.$$

Since $N_2 = 3$, we get that

$$N_{h+1} = 3 + 3(F_{h+2} - 2) = 3F_{h+2} - 3. \tag{4}$$

This is also consistent with our first derivation, since the basis of the induction is obviously the same, and assuming the truth of eq. (2) for values up to h, we get by inserting eq. (4) for N_h and N_{h-1} that

$$N_{h+1} = (3F_{h+1} - 3) + (3F_h - 3) + 3 = 3F_{h+2} - 3.$$

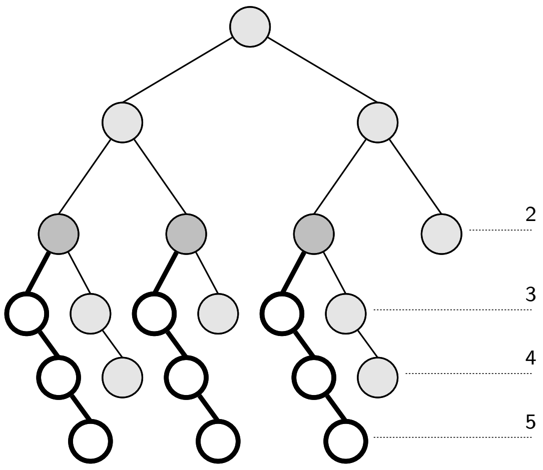

FIGURE 8: *Extending a FWT.*

The PFWT corresponding to the FWT of height $h+1$ is of height $h-1$ and obtained by pruning all single child nodes of the FWT: for each of the F_h leaves of the lowest level $h+1$, two nodes are saved, and for each of the F_{h-1} leaves on level h, only a single node is erased. Denoting by P_h the number of nodes in a PFWT of height h, we get

$$P_{h-1} = N_{h+1} - 2F_h - F_{h-1}. \tag{5}$$

But

$$2F_h + F_{h-1} = F_{h+1} + F_h = F_{h+2},$$

so substituting the value for N_{h+1} from eq. (4), we get

$$P_{h-1} = 3F_{h+2} - 3 - F_{h+2} = 2F_{h+2} - 3.$$

The ratio of the sizes of the pruned to the original FWTs is therefore

$$\frac{P_{h-1}}{N_{h+1}} = \frac{2F_{h+2}-3}{3F_{h+2}-3} \quad \underset{h\to\infty}{\longrightarrow} \quad \frac{2}{3},$$

when the size of the tree grows to infinity, so that about one third of the nodes will be saved. Figure 9 plots the number of nodes in both original and pruned FWTs as a function of the tree's heights.

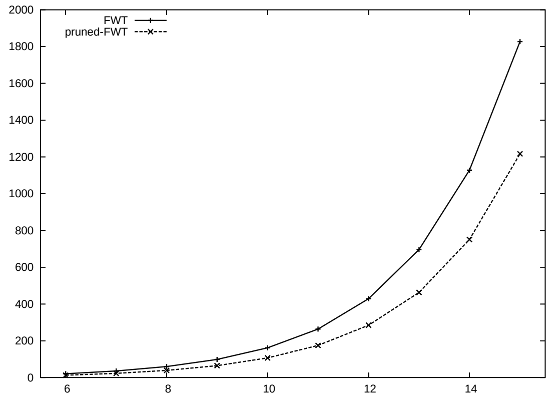

Figure 9: *Number of nodes in original and pruned FWT as function of height.*

5. Wavelet Trees for Higher Order Fibonacci Encoded files

The idea of the previous sections can be generalized to *higher-order Fibonacci codes* [11]. Fibonacci numbers of order $m \geq 2$ are defined by the recursive formula:

$$F_i^{(m)} = F_{i-1}^{(m)} + F_{i-2}^{(m)} + ... + F_{i-m}^{(m)} \qquad \text{for } i \geq 1,$$

with boundary conditions $F_0^{(m)} = 1$ and $F_i^{(m)} = 0$ for $i < 0$. A representation of the integers on the basis of higher order Fibonacci numbers has the property that there is no occurrence of a string of m consecutive 1s. For $m = 2$ one gets the standard Fibonacci sequence defined above, $F_i^{(2)} = F_i$. Such a generalization seems natural from a theoretic point of view, but has moreover also practical value, as for certain applications, the optimal performance might be reached for $m > 2$. For example, the best compression in [21] is consistently obtained for the codes of order $m = 3$; higher order Fibonacci codes are suggested in [1] for the transmission of binary strings in an unbounded range; another generalization has been designed for the encoding of data on CD-ROMs [18] and is known as *Eight-to-Fourteen-Modulation*: every byte of 8 bits is mapped to a bit-string of length 14 in which there are at least two zeros between any two 1s.

At first sight, Fibonacci codes of order m should have been defined as a direct extension of the definition for $m = 2$, uniquely representing any integer k as a sum of Fibonacci numbers of order m and appending a string of $m - 1$ 1s. However, the code generated by this generalization is not uniquely decodable. We therefore use the definition of [21] for higher order of Fibonacci Codes as follows: $\mathcal{F}^{(m)}$ denotes the set of binary codewords of length $\geq m$, such that every codeword contains exactly one occurrence of m consecutive 1s, and this occurrence is the suffix of every codeword. For $m = 2$ this definition is equivalent to the basic definition of a Fibonacci code as given in section 2.2. For example, for $m = 3$, we get

$$\begin{aligned}\mathcal{F}^{(3)} \;=\; & \{111, 0111, 00111, 10111, 000111, 100111, 010111, 110111, 0000111,\\ & 1000111, 0100111, 1100111, 0010111, 1010111, 0110111, \cdots\}\end{aligned}$$

There is still a connection to Fibonacci numbers, namely, it can be shown that for $m \geq 2$ and $i \geq 0$, the code $\mathcal{F}^{(m)}$ consists of $F_i^{(m)}$ codewords of length $i + m$ [1].

We denote the Wavelet tree for Fibonacci codes of order m by FWT$^{(m)}$, and evaluate now the number of nodes in its original and pruned versions. As in the case of $m = 2$, we restrict our discussion to prefixes of the code corresponding to full levels of the tree.

Given an FWT$^{(m)}$ of height h, we add the paths corresponding to the longest codewords of length $h+1$ in order to construct FWT$^{(m)}$ of height $h+1$. We thus adjoin $F_{h-m+1}^{(m)}$ new paths to the tree, and the new nodes added by these paths are those corresponding to their common suffix 01^m of length $m+1$. This results in the following recurrence: $N_h^{(m)} = 0$ for $h < m$, and

$$N_{h+1}^{(m)} = N_h^{(m)} + (m+1) \cdot F_{h-m+1}^{(m)} \qquad \text{for } h \geq m-1. \tag{6}$$

For example, for $m = 3$ and $h \geq 2$, $N_{h+1}^{(3)} = N_h^{(3)} + 4 \cdot F_{h-2}^{(3)}$, and Figure 10 depicts the nodes of FWT$^{(3)}$ of height $h = 6$, starting from FWT$^{(3)}$ of height $h = 5$ (gray and darker nodes), extending the darker nodes on level 2 by the bold edges and white nodes, representing the suffix 0111.

Applying equation (6) repeatedly $h - m + 2$ times gives

$$N_{h+1}^{(m)} = N_{m-1}^{(m)} + (m+1) \cdot \left(\sum_{i=0}^{h-m+1} F_i^{(m)} \right).$$

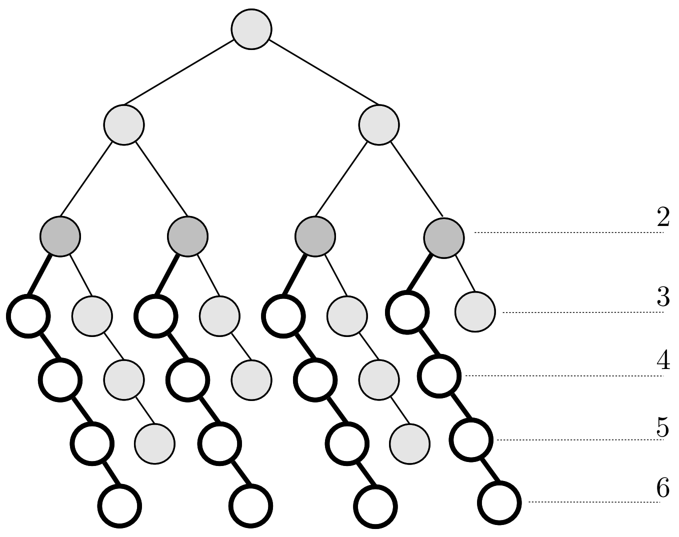

FIGURE 10: *Constructing FWT$^{(3)}$ of height 6 from that of height 5.*

Let $S_k^{(m)}$ denote the summation of the first $k+1$ Fibonacci numbers of order m, that is, $S_k^{(m)} = \sum_{i=0}^{k} F_i^{(m)}$. Note that we have included also the first element, indexed 0, in this summation, $F_0^{(m)} = 1$, even though the Fibonacci representations start only from index 1. $N_{h+1}^{(m)}$ can be simply rewritten as

$$N_{h+1}^{(m)} = (m+1) \cdot S_{h-m+1}^{(m)}. \tag{7}$$

The pruned tree corresponding to the FWT$^{(m)}$ of height h is of height $h-m$, and obtained by pruning all single child nodes of the FWT$^{(m)}$: for each of the $F_{h-m}^{(m)}$ leaves of the lowest level h, m nodes are saved, and for each of the $F_{h-m-1}^{(m)}$ leaves on level $h-1$, $m-1$ nodes are erased, and so on, up to level $h-2m+1$ in which a single node is saved for each of the $F_{h-2m}^{(m)}$ codewords. Denoting by $P_h^{(m)}$ the number of nodes in a pruned FWT$^{(m)}$ of height h, we get

$$\begin{aligned}
P_{h-m}^{(m)} &= N_h^{(m)} - \Big(mF_{h-m}^{(m)} + (m-1)F_{h-m-1}^{(m)} + \cdots + 2F_{h-2m+2}^{(m)} + F_{h-2m+1}^{(m)}\Big) \\
&= N_h^{(m)} - \sum_{i=1}^{m} i \cdot F_{h-2m+i}^{(m)} \\
&= N_h^{(m)} - \sum_{j=1}^{m} \sum_{i=j}^{m} F_{h-2m+i}^{(m)} \\
&= N_h^{(m)} - \sum_{j=1}^{m} (S_{h-m}^{(m)} - S_{h-2m+j-1}^{(m)}) \\
&= (m+1)S_{h-m}^{(m)} - mS_{h-m}^{(m)} + \sum_{k=h-2m}^{h-m-1} S_k^{(m)}.
\end{aligned}$$

To evaluate the rightmost summation, we use the following:

LEMMA: *For every order $m \geq 2$ and any starting index $j \geq 0$, the sum, starting at index j, of m consecutive summation elements of the mth order Fibonacci sequence is the next higher summation element, minus 1, i.e.,*

$$\sum_{i=j}^{j+m-1} S_i^{(m)} = S_{j+m}^{(m)} - 1 \qquad \text{for all} \quad j \geq 0, \quad m \geq 2.$$

PROOF: It follows from the definition that the first $m+1$ elements of the $F^{(m)}$ sequence are powers of 2, $F_0^{(m)} = F_1^{(m)} = 1$, and

$$F_i^{(m)} = 2^{i-1} \qquad \text{for } 2 \leq i \leq m,$$

so their sum $S_t^{(m)} = 1 + \sum_{i=1}^{t} 2^{i-1} = 2^t$ is also a power of 2 for $0 \leq t \leq m$, and therefore the sum of the first m such summations is

$$\sum_{t=0}^{m-1} S_t^{(m)} = \sum_{t=0}^{m-1} 2^t = 2^m - 1 = S_m^{(m)} - 1.$$

Assume the truth of the claim for $j-1 \geq 0$, and let us show it for j:

$$\begin{aligned}
S_j^{(m)} &+ S_{j+1}^{(m)} + \cdots + S_{j+m-1}^{(m)} \\
&= \left(F_j^{(m)} + S_{j-1}^{(m)}\right) + \left(F_{j+1}^{(m)} + S_j^{(m)}\right) + \cdots + \left(F_{j+m-1}^{(m)} + S_{j+m-2}^{(m)}\right) \\
&= \sum_{t=j}^{j+m-1} F_t^{(m)} + \sum_{t=j-1}^{j+m-2} S_t^{(m)}.
\end{aligned}$$

But using the definition of Fibonacci numbers for the first summation, and the inductive hypothesis for the second, we get

$$= \quad F_{j+m}^{(m)} + S_{j+m-1}^{(m)} - 1 = S_{j+m}^{(m)} - 1. \qquad \blacksquare$$

Returning to the evaluation of the number of nodes in the pruned trees, we get

$$P_{h-m}^{(m)} = (m+1) \cdot S_{h-m}^{(m)} - m \cdot S_{h-m}^{(m)} + S_{h-m}^{(m)} - 1 = 2 \cdot S_{h-m}^{(m)} - 1.$$

and the ratio of the sizes of the pruned to the original FWTs of order m is therefore

$$\frac{P_{h-m}^{(m)}}{N_{h+1}^{(m)}} = \frac{2 \cdot S_{h-m}^{(m)} - 1}{(m+1) \cdot S_{h-m}^{(m)}} \quad \underset{h\to\infty}{\longrightarrow} \quad \frac{2}{m+1}.$$

Incidentally, the lemma can also be used to derive another form for $N_{h+1}^{(m)}$, as we did in Section 4.2. We have, using equation (7),

$$\begin{aligned}
N_{h+1}^{(m)} &= (m+1)S_{h+1}^{(m)} \\
&= m+1+(m+1)\left(S_{h+1}^{(m)} - 1\right) \\
&= m+1+(m+1)\sum_{i=h-m+1}^{h} S_{i-m}^{(m)} \\
&= m+1+\sum_{i=h-m+1}^{h} N_i^{(m)}. \qquad (8)
\end{aligned}$$

This equality illustrates the inductive process, shown in Figure 11, for generating the FWT$^{(m)}$ of height $h+1$ from those of height $h, h-1$, down to $h-m+1$, on which the construction of the Wavelet tree could have been based.

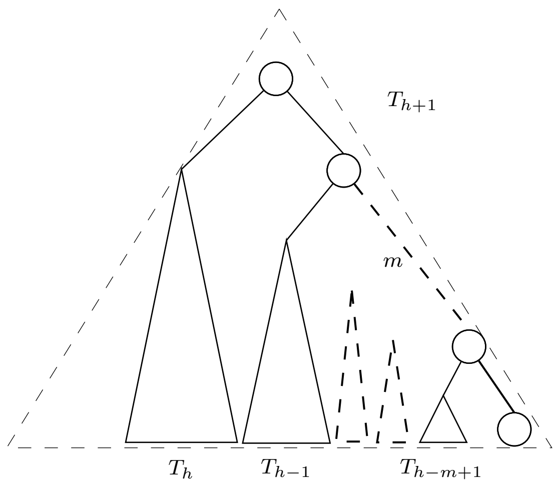

FIGURE 11: *Recursive definition of a FWT$^{(m)}$ of height $h+1$.*

The rightmost path of the tree, starting at the root and ending at the rightmost leaf, corresponds to the substring 1^m. We call the nodes of this path, top down, the ith right descendant of the root, for $0 \le i \le m$. The left subtree of the root is the FWT$^{(m)}$ of height h, and more generally, the left subtree rooted at the i^{th} right descendant of the root is the FWT$^{(m)}$ of height $h-i$, $0 \le i \le m-1$. This yields equality (8).

6. Experimental Results

While the number of nodes saved in the pruning process could be analytically derived in the previous section, the number of bits to be saved in the compressed file will depend on the distribution of the different encoded elements. It might be hard to define a "typical" distribution of probabilities, so we decided to calculate the savings for the distribution of characters in several real-life languages.

The distribution of the 26 letters and the 378 letter pairs of English was taken from Heaps [16]; the distribution of the 29 letters of Finnish is from Pesonen [27]; the distribution for French (26 letters) has been computed from the database of the *Trésor de la Langue Française* (TLF) of about 112 million words (for details on TLF, see [4]); for German, the distribution of 30 letters (including blank and *Umlaute*) is given in Bauer & Goos [3]; for Hebrew (30 letters including two kinds of apostrophes and blank, and 743 letter-pairs), the distribution has been computed using the database

File	n	FWT$^{(m)}$			pruned			Huff
		$m=2$	$m=3$	$m=4$	$m=2$	$m=3$	$m=4$	
English	26	4.90	5.74	6.72	4.41	4.65	4.87	4.19
Finnish	29	4.76	5.63	6.61	4.41	4.65	4.88	4.04
French	26	4.68	5.53	6.52	4.23	4.56	4.83	4.00
German	30	4.70	5.54	6.53	4.34	4.55	4.85	4.15
Hebrew	30	4.82	5.64	6.62	4.40	4.54	4.80	4.29
Italian	26	4.70	5.56	6.55	4.28	4.65	4.88	4.00
Portuguese	26	4.67	5.52	6.51	4.23	4.60	4.87	4.01
Russian	32	5.13	5.92	6.89	4.74	4.73	4.89	4.47
Spanish	26	4.71	5.57	6.56	4.27	4.61	4.87	4.05
English-2	378	8.78	8.95	9.75	8.24	7.89	8.45	7.44
Hebrew-2	743	9.13	9.22	10.01	8.81	8.30	9.03	8.04
English-w	289101	12.358	11.77	12.37	12.356	11.76	12.32	11.20
French-w	439191	11.313	10.94	11.60	11.311	10.93	11.59	10.48
Hebrew-w	296933	15.00	13.89	14.36	14.99	13.88	14.29	13.06

TABLE 2: *Compression Performance: average codeword lengths, in bits, using FWT$^{(m)}$ or PFWT$^{(m)}$, of the distributions of single characters, letter-pairs and words, for various natural languages.*

of the Responsa Retrieval Project (RRP) [10] of about 40 million Hebrew and Aramaic words; the distribution for Italian, Portuguese and Spanish (26 letters each) can be found in Gaines [12], and for Russian (32 letters) in Herdan [17].

To get even larger distributions, we considered natural texts in several languages encoded as sequences of *words* rather than of characters or character pairs, which yields distributions with hundreds of thousands of elements. For English, the text is 500MB (87 million words) of the *Wall Street Journal*, and for French and Hebrew, we chose subsets of TLF and RRP.

The results, summarized in Table 2, are partitioned into three blocks. The upper block consists of the small single character alphabets, the middle block of the letter-pairs, and the lower block of the distributions of the different words. The second column shows the size n of the encoded alphabets. The next three columns show the average codeword length in bits for the original FWT$^{(m)}$, for $m = 2, 3, 4$, and the following three columns, headed pruned, are the corresponding values for the pruned trees. To get some idea on the relative compression performance, we add, in the last column, entitled Huff, the average codeword length of an optimal Huffman code.

As can be seen, the pruning yields, for $m = 2$, a 7–10% gain for the smaller alphabets, and 2–3% for the letter-pairs; for $m = 3$, the gain is 16–20% for the small and about 10% for the moderately large alphabets; and for $m = 4$, the corresponding reductions are 25–29% and 10–13%. For the very large distributions of the lower block, the reduction is hardly noticeable. The reduced savings can be explained by the fact that though a third for $m = 2$, a half for $m = 3$ and 60% for $m = 4$, of the nodes have been eliminated, they correspond to the leaves with lowest probabilities, so the expected savings are lower.

The increase, relative to the Huffman encoded files, of the size of the FWT$^{(m)}$ compressed files can roughly be reduced to a third by the pruning technique for the smaller files. For example, for English, the FWT$^{(2)}$ compressed file is 17% larger than the Huffman compressed one, but the pruned FWT$^{(2)}$ reduces this excess to only 5%. For the small alphabets, all the numbers have been calculated for the given sizes n, and not been approximated by trees with full levels.

The best compression, among the Fibonacci codes, is obtained for $m = 2$ for the smaller files, and for $m = 3$ for the very large ones. This is not surprising, as the appearance of m 1s at the end of each codeword is a very high price to pay, which is profitable only for very large alphabets. However, these are exactly the bits targeted by the pruning technique, resulting, for each alphabet, in similar average codeword sizes for different m values, as can be seen by the numbers in the columns for the pruned trees in Table 2. In particular, note that for the letter-pairs (middle block) and for the single characters of Russian, the best performance of the pruned trees is achieved for $m = 3$, while for the original FWT$^{(m)}$, $m = 2$ is best.

Referring to the time complexity, direct access, rank and select operations on Wavelet trees for any prefix free codes take time proportional to the corresponding codeword length, under the assumption of constant time rank and select implementations on bit vectors. Thus, Table 2 also shows that the algorithms on PFWT are always faster than those on FWT, while both are slower than for Huffman.

7. Time/Space tradeoffs for rank and select

As mentioned before, Jacobson [19] 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. His solution is based on storing rank answers every $\lceil \log^2 n \rceil$ bits of B, using $\log n$ bits per sample, and then storing rank answers relative to the last sample every $\lceil \frac{\log n}{2} \rceil$ bits, requiring $\log(\log^2 n) = 2 \log\log n$ bits per sub-sample. Finally, a precomputed table is used to store rank answers of every bit stream of length $\lceil\frac{\log n}{2}\rceil$ using $O(\sqrt{n}\log n\log\log n)$ bits. The final answer of the rank query thus uses three memory accesses, and a total of $O(\frac{n}{\log n}+\frac{n\log\log n}{\log n}+\sqrt{n}\log\log n)$ bits.

For example, if the size of the bit vector is $n=2^{32}$, then 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.

It is important to stress that the numbers here demonstrate that the overhead $o(n)$ of the rank and select data structures for a bitmap of size n is more than $\frac{4n\log\log n}{\log n}$, which for $n=2^{32}$ is at least $\frac{20}{32}n=0.66n$, — this is not at all negligible. Furthermore, the size of the Wavelet tree, which is based on this data structure and uses it in all of its internal nodes, might reduce the compression gain achieved by the corresponding encoding. The present work reduces the size of the Wavelet tree without hurting the direct access capabilities, so that the Wavelet tree becomes useful also in practice. Methods in [13] 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, the extra space of these implementations can also be saved.

8. Concluding Remarks

Fibonacci codes are universal variable length codes consisting, similarly to Elias γ and δ codes, End-tagged dense codes and (s,c) dense codes, of sets of codewords that are fixed in advance and need not be generated for each new input distribution. This property helps to reduce the processing time at the cost of reduced compression, but Fibonacci codes have also other advantages [21].

A Wavelet tree is a data structure adjoined to a file that has been compressed by a variable length encoding, which allows direct access to the underlying file, resulting in the fact that the compressed file is not needed any more. The direct access can be achieved by means of rank and select queries on a set of bitmaps stored in the nodes of the Wavelet tree. There have recently been many works adapting the Wavelet tree to various variable length encodings. We followed this line of investigation, showing how to apply Wavelet trees to Fibonacci codes. Moreover, an enhanced Wavelet tree for Fibonacci encoded files was proposed, pruning a part of the nodes.

We showed, both analytically and in experiments on real life data, that the number of nodes is reduced by a factor of about $2/(m+1)$, where m is the order of the Fibonacci code, which improves the space complexity of the Wavelet trees. As to the weighted average number of bits used for each codeword, which affects the time complexity, the reduction for the pruned Wavelet tree is relatively less, since the nodes that are eliminated correspond to those of the lowest probabilities. For very large distributions, there may thus hardly be any gain in time, and the advantage of the pruning is only the improved space complexity. Nevertheless, for smaller files, the pruning may yield an improvement of $7-10\%$ in the number of needed comparisons for $m=2$, and even more for larger m.

References

[1] A. Apostolico, A.S. Fraenkel, Robust transmission of unbounded strings using Fibonacci representations, *IEEE Trans. Inform. Theory* **33** (1987) 238–245.

[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] F.L. Bauer, G. Goos, *Informatik, Eine einführende Übersicht, Erster Teil,* Springer Verlag, Berlin (1973).

[4] A. Bookstein, S.T. Klein, D.A. Ziff, A systematic approach to compressing a full text retrieval system, *Information Processing & Management* **28** (1992) 795–806.

[5] 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.

[6] N.R. Brisaboa, A. Fariña, G. Navarro, M.F. Esteller, (s,c)-dense coding: an optimized compression code for natural language text databases, *Proc. Symposium on String Processing and Information Retrieval SPIRE'03*, *LNCS* **2857**, Springer Verlag (2003) 122–136.

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

[8] D. Clark, Compact Pat Trees, Ph.D. Thesis, University of Waterloo, Canada, (1996).

[9] P. Elias, Universal codeword sets and representation of the integers, *IEEE Trans. on Inf. Th.,* **IT–12** (1975) 194–203.

[10] A.S. Fraenkel, All about the Responsa Retrieval Project you always wanted to know but were afraid to ask, Expanded Summary, *Jurimetrics J.* **16** (1976) 149–156.

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

[12] H.F. Gaines, *Cryptanalysis, A Study of Ciphers and their solution*, Dover Publ. Inc., New York (1956).

[13] 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.

[14] 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.

[15] R. Grossi, G. Ottaviano, The wavelet trie: maintaining an indexed sequence of strings in compressed space, *Symposium on Principles of Database Systems (PODS)* (2012) 203–214.

[16] H.S. Heaps, *Information Retrieval, Computational and Theoretical Aspects*, Academic Press, New York (1978).

[17] G. Herdan, *The Advanced Theory of Language as Choice and Chance*, Springer-Verlag, New York (1966).

[18] K.A. Immink, J.G. Nijboer, H. Ogawa, K. Odaka, Method of coding binary data, US Patent 4,501,000, issued Feb 19, 1985.

[19] G. Jacobson, Space efficient static trees and graphs, *Proceedings of FOCS* (1989), 549–554.

[20] 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.

[21] S.T. Klein, M. Kopel Ben-Nissan, On the Usefulness of Fibonacci Compression Codes, *The Computer Journal* **53** (2010) 701–716.

[22] D.E. Knuth, *The Art of Computer Programming,* Vol. **III**, *Sorting and Searching,* Addison-Wesley, Reading, MA (1973).

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

[24] D.R. Morrison, Patricia - practical algorithm to retrieve information coded in alphanumeric, *Journal of the ACM*, **15**(4) (1968) 514–534.

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

[26] D. Okanohara, K. Sadakane, Practical entropy-compressed rank/select dictionary, *Proc. ALENEX, SIAM* (2007).

[27] J. Pesonen, Word inflexions and their letter and syllable structure in Finnish newspaper text, Research Rep. 6/1971, Dept. of Special Education, University of Jyräskylä, Finland (in Finnish, with English summary).

[28] 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.

[29] 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.

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