Random Access to Fibonacci Codes

Shmuel T. Klein[1] and Dana Shapira[2,3]

[1] Computer Science Department, Bar Ilan University, Israel
[2] Computer Science Department, Ashkelon Academic College, Israel
[3] Department of Computer Science and Mathematics, Ariel University, Israel
tomi@cs.biu.ac.il, shapird@gmail.com

Abstract. A Wavelet tree 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.

1 Introduction and previous work

Variable length codes, such as Huffman and Fibonacci codes, were 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 since the beginning position of the ith element depends on 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 bit vector to indicate the beginning of each block. The time complexity of random access becomes $O(b)$, 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.

Another line of investigation applies efficiently implemented *rank* and *select* operations on bit-vectors [23,20] to develop a data structure called a *Wavelet Tree*, suggested by Grossi et al. [11], 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 recursively with the children.

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 a small amount of additional memory storage as compared to the memory usage of the compressed file itself. However, since it enables efficient direct access, it is a price we are 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, one of the implementations 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.

Shmuel T. Klein, Dana Shapira: Random Access to Fibonacci Codes, pp. 96–109.
Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2

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 [19]. 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. [4] use a variant of a Wavelet tree on Byte-Codes, which 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 (e.g., 128 for ETDC). 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 256-ary one for byte-codes, using less space.

In another work, Brisaboa et al. [6] introduced directly accessible codes (DACs) by integrating rank dictionaries into byte aligned codes. Their method is based on Vbyte coding [25], 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 binary form into blocks of b bits and prepends each block with a flag-bit as follows. The highest bit is 0 in the extended block holding the most significant bits of x_i. and 1 in the others. Thus, the 0 bits acts as a comma between codewords. For example, if $b = 3$, and $x_i = 25$, the 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 [18] 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 also 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 static codes are preferred, for various reasons, to Huffman codes. These static codes include, among others, the different Elias codes, dense codes like ETDC and SCDC [5], and Fibonacci codes.

The rest of the paper is organized as follows. Section 2 brings some technical details on rank and select, as well as 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 brings experimental results.

2 Preliminaries

We bring here some technical details on the rank and select operations, as well as on Fibonacci codes, which will be useful for the understanding of the ideas below.

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 $rank_{1-b}(B,i) = i - rank_b(B,i)$, thus, only one of the two, say, $rank_0(B,i)$ needs to be computed. Jacobson [14] 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 $\log^2 n$ bits of B, using $\log n$ bits per sample, and then storing rank answers relative to the last sample every $\frac{\log n}{2}$ bits (requiring $\log(\log^2 n) = 2\log\log n$ bits per sub-sample, and using a universal table to complete the answer to a rank query within a subtable.

Raman et al. [23] partition the input bitmap B into blocks of length $t = \lceil \frac{\log n}{2} \rceil$. These are assigned to classes: a block with k 1s belongs to class k. Class k contains $\binom{t}{k}$ elements, so $\lceil \log \binom{t}{k} \rceil$ bits are used to refer to an element of it. A block is identified with a pair (k,r), where k is its class ($0 \le k \le t$) using $\lceil \log(t+1) \rceil$ bits, and r is the index of the block within the class using $\lceil \log \binom{t}{k} \rceil$ bits. A global universal table for each class k translates in $O(1)$ time any index r into the corresponding t bits. The sizes of these tables add up to $t2^t$ bits. The sequences of $\lceil \frac{n}{t} \rceil$ class identifiers k is stored in one array, K, and that of the indexes r is stored in another, R. The blocks are grouped into superblocks of length $s = \lfloor \log n \rfloor$. Each superblock stores the rank up to its beginning, and a pointer to R where its indexes start. The size of R is upper bounded by $nH_0(B)$, and the main space overhead comes from K which uses $n\lceil \log t + 1 \rceil$ bits.

To solve $\mathsf{rank}_b(B,i)$, the superblock of i is first computed, and its rank value up to its beginning is obtained. Second, the classes from the start of the superblock are scanned, and their k values are accumulated. The pointer to R is obtained in parallel by attaining the pointer value from the start of the superblock and adding $\lceil \log \binom{t}{k} \rceil$ bits for each class k which is processed. This scanning continues up to the block i belongs to, whose index is extracted from R, and its bits are recovered from the universal table.

The $\mathsf{select}_b(B,i)$ operation can be done by applying binary search in B on the index j so that $\mathsf{rank}_b(B,j) = i$ and $\mathsf{rank}_b(B,j-1) = i-1$. Using the $O(1)$ data structure of Jacobson or that of Raman et al. for rank implies an $O(\log n)$ time solution for select. 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. In more details, B is partitioned into blocks of two kinds, each containing exactly $\log^2 n$ 1s. The first kind are the blocks that are long enough 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 the "short" blocks, of size $O(\log^c n)$, where c is a constant. Recording the 1-positions inside them requires only $O(\log\log n)$ bits by repartitioning these blocks, and storing their relative position. The remaining blocks are short enough to be handled in constant time using a universal table.

González et al. [10] give a practical solution for rank and select data structures. Okanohara and Sadakane [21] 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. [1] 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 [20] 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 universal tables of [23]'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.

2.2 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 finite prefix of this infinite sequence 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 [8,16]. 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

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

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 [8] 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 {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% [8], and may be less, around 9%, on much larger alphabets [16]. On the other hand, Fibonacci coding may be significantly better than other static codes such as Elias coding, End-tagged dense codes (ETDC) and (s,c)-dense codes (SCDC) [16].

3 Random Access to Fibonacci Encoded Files

3.1 Using an Auxiliary Index

A trivial solution for gaining random access is to use an additional auxiliary index constructed in the following way:

1. Compress the input file, T, using a Fibonacci Code, resulting in the file $\mathcal{E}(T)$ of size u.
2. Generate a bitmap B of size u so that $B[i] = 1$ if and only if $\mathcal{E}(T)[i]$ is the first bit of a codeword.
3. Construct a rank and select succinct data structure for B.

Recall that $u = |\mathcal{E}(T)|$ is the size of the uncompressed text, and Σ denote the alphabet. In the suggested solution, the space used to accomplish constant time rank and select operations (excluding the encoded file) is

$$u + \mathcal{B}(u, |\Sigma| + u) + o(u) + O(\log\log|\Sigma|),$$

where $\mathcal{B}(x, y) = \lceil \log \binom{y}{x} \rceil$ (the information theoretic lower bound in bits for storing a set of x elements from a universe of size y) using Raman et al.'s implementation [23]. 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 $\mathcal{E}(T[i])$ as a single 1-bit in B, and other bits in $\mathcal{E}(T[i])$ as a 0 in B. The memory storage is therefore reduced to $\mathcal{B}(u, |\Sigma| + u) + o(u) + O(\log\log|\Sigma|)$. Even better solutions are presented below.

3.2 Using Wavelet Trees

We adjust the Wavelet tree to Fibonacci codes in the following way. Given is an alphabet Σ and a text $T = t_1 t_2 \cdots t_n$ of size n, where $t_i \in \Sigma$. Let $\mathcal{E}_{fib}(T) = f(t_1) \cdots f(t_n)$ be the encoding of T using the first $|\Sigma|$ codewords of the Fibonacci code. 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 has been given above in the introduction.

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.

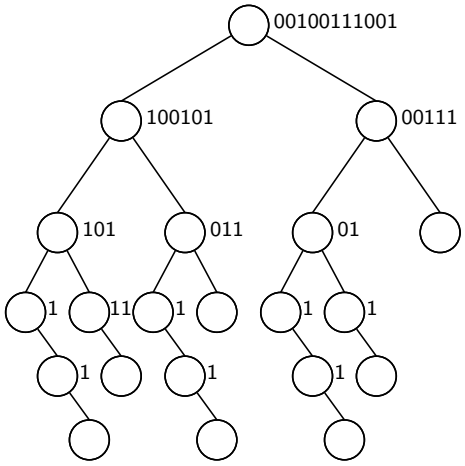

Figure 1. Fibonacci Wavelet Tree for the text $T =$`COMPRESSORS`.

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 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 [17] as Fibonacci trees.

Consider, for example, the text $T =$ `COMPRESSORS` over an 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 39 bits is the following binary string, in which spaces have been added for clarity.

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

The corresponding FWT, including the annotating bitmaps, is given in Figure 1.

The Wavelet tree for $\mathcal{E}_{fib}(T)$ 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 $O(|f(t_i)|)$, where $f(x)$ is the Fibonacci encoding of x. The algorithm for extracting t_i from an FWT rooted by v_{root} is given in Figure 2 using the function call $\mathsf{extract}(v_{root},i)$. 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 universal one, the decoding of *code* in line 5 is done by a fixed lookup table.

```
extract(v_root, i)
1    code ⟵ ε
2    while v is not a leaf
3        if B_v[i] = 0
3.1          v ⟵ left(v)
3.2          code ⟵ 0 · code
3.3          i ⟵ rank_0(B_v, i)
4        else
4.1          v ⟵ right(v)
4.2          code ⟵ 1 · code
4.3          i ⟵ rank_1(B_v, i)
5    return 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 a symmetric way. More precisely, we use the notation $\mathsf{select}_x(T,i)$ for returning the position of the ith occurrence of x in T.

Computing $\mathsf{select}_x(T,i)$ is done in the opposite way. We start from the leaf, ℓ, representing the Fibonacci codeword $f(x)$ 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, $O(|f(x)|)$.

```
select_x(T, i)
1    ℓ ⟵ leaf corresponding to f(x)
2    v ⟵ father of ℓ
3    while v ≠ v_root
3.1      if ℓ is a left child of v
3.1.1            i ⟵ index of the ith 0 in B_v
3.2      else // ℓ is a right child of v
3.2.1            i ⟵ index of the ith 1 in B_v
3.3      v ⟵ father of v
4    return i
```

Figure 3. select the ith occurrence of x in T.

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 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. The binary tree corresponding to the Fibonacci code is therefore not complete, as can be seen, e.g., in the example in Figure 1, and the nodes corresponding to this suffix, at least for the lowest levels of the tree, are redundant. We can therefore eliminate all nodes which are single children of their parent nodes. 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. They have also been applied on Huffman trees [15] producing a compact tree for efficient use, such as compressed pattern matching [24]. Applying this strategy on the FWT of Figure 1 results in the pruned Fibonacci Wavelet Tree given in Figure 4.

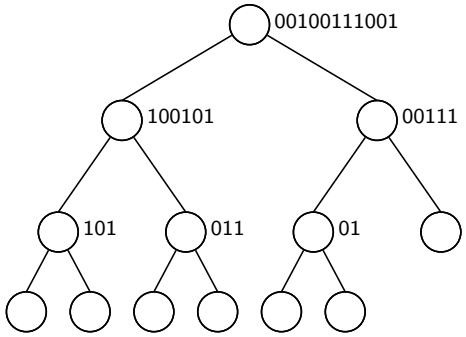

Figure 4. Pruned Fibonacci Wavelet Tree for the text $T =$ COMPRESSORS.

The $\mathsf{select}_x(T, i)$ 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 pruned FWT requires minor adjustments for concatenating the pruned parts. The following lines should be added instead of line 5 in the algorithm of Figure 2.

```
5    if suffix of code = 0
5.1      code ⟵ code · 11
6    if suffix of code ≠ 11
6.1      code ⟵ code · 1
7    return 𝒟(code)
```

Figure 5. Extracting t_i from the pruned Fibonacci Wavelet Tree.

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 only uses a partial set of the bits of the encoded file. The main savings of pruned FWTs 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 pruned FWT.

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

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 [16], 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 $n \in \{F_h - 1 | h \geq 3\}$. We defer the more involved calculations for general n to the full paper.

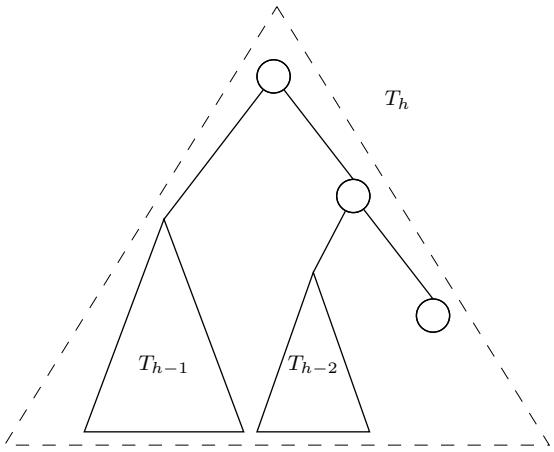

Figure 7. Recursive definition of a Fibonacci Wavelet Tree of depth h.

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

$$N_{h+1} = N_h + N_{h-1} + 3. \tag{2}$$

The second way 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 grey 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

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

The pruned FWT 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,

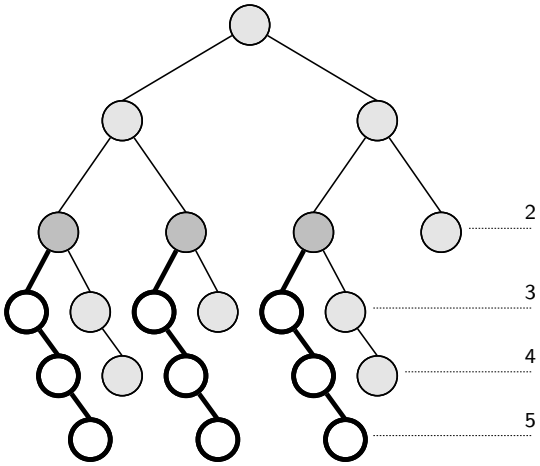

Figure 8. Extending a Fibonacci Wavelet Tree

only a single node is erased. Denoting by S_h the number of nodes in a pruned FWT of height h, we get

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

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

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

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

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

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.

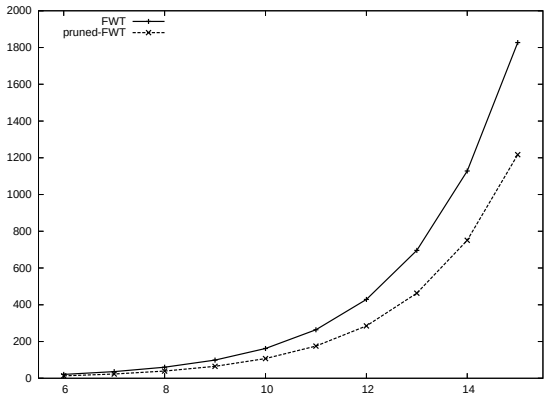

Figure 9. Number of nodes in original and pruned FWT as function of height.

5 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.

File	n	height	FWT	pruned	Huffman
English	26	8	4.90	4.43	4.19
Finnish	29	8	4.76	4.44	4.04
French	26	8	4.53	4.14	4.00
German	30	8	4.70	4.37	4.15
Hebrew	30	8	4.82	4.42	4.29
Italian	26	8	4.70	4.32	4.00
Portuguese	26	8	4.67	4.28	4.01
Spanish	26	8	4.71	4.30	4.05
Russian	32	8	5.13	4.76	4.47
English-2	378	14	8.78	8.56	7.44
Hebrew-2	743	15	9.13	8.97	8.04

Table 1. Compression Performance

The distribution of the 26 letters and the 371 letter pairs of English was taken from Heaps [12]; the distribution of the 29 letters of Finnish is from Pesonen [22]; 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 [3]); for German, the distribution of 30 letters (including blank and *Umlaute*) is given in Bauer & Goos [2]; for Hebrew (30 letters including two kinds of apostrophes and blank, and 735 bigrams), the distribution has been computed using the database of The Responsa Retrieval Project (RRP) [7] of about 40 million Hebrew and Aramaic words; the distribution for Italian, Portuguese and Spanish (26 letters each) can be found in Gaines [9], and for Russian (32 letters) in Herdan [13].

Note that the input of our tests consists of published probability distributions, not of actual texts. There are therefore no available texts that could be compressed. We can only calculate the average codeword lengths, from which the expected size of the compressed form of some typical natural language text can be extrapolated. To still get some idea on the compression performance, we add as comparison the average codeword length of an optimal Huffman code.

The results are summarized in Table 1, the two last lines corresponding to the bigrams. The second column shows the size n of the alphabet. The column entitled height is the height of the original FWT tree for the given distribution, FWT shows the average codeword length for the original FWT, and pruned the corresponding value for the pruned FWT. As can be seen, there is a 7–10% gain for the smaller alphabets, and 2–3% for the larger ones. The reduced savings can be explained by the fact that though a third of the nodes has been eliminated, they correspond to the leaves with lowest probabilities, so the expected savings are lower. The last column, entitled Huffman, gived the average codeword length of an optimal Huffman code. We see that the increase, relative to the Huffman encoded files, of the size of the

FWT compressed files can roughly be reduced to half by the pruning technique. For example, for English, the FWT compressed file is 17 % than the Huffman compressed one, but the pruned FWT reduces this excess to 6 %. All the numbers have been calculated for the given sizes n of the alphabets, and not been approximated by trees with full levels.

References

1. J. Barbay, T. Gagie, G. Navarro, and Y. Nekrich: *Alphabet partitioning for compressed rank/select and applications.* Algorithms and Computation, Lecture Notes in Computer Science, 6507 2010, pp. 315–326.
2. F. Bauer and G. Goos: *Informatik, Eine einführende Übersicht, Erster Teil*, Springer Verlag, Berlin, 1973.
3. A. Bookstein, S. T. Klein, and D. A. Ziff: *A systematic approach to compressing a full text retrieval system.* Information Processing & Management, 28 1992, pp. 795–806.
4. N. R. Brisaboa, A. Fariña, G. Ladra, and G. Navarro: *Reorganizing compressed text*, in Proc. of the 31th Annual Internetional ACM SIGIR Conference on Research and Developing in Information Retrieval (SIGIR), 2008, pp. 139–146.
5. N. R. Brisaboa, A. Fariña, G. Navarro, and M. F. Esteller: *(S,C)-dense coding: an optimized compression code for natural language text databases*, in Proc. Symposium on String Processing and Information Retrieval SPIRE'03, LNCS, vol. 2857, Springer Verlag, 2003, pp. 122–136.
6. N. R. Brisaboa, S. Ladra, and G. Navarro: *DACs: Bringing direct access to variable length codes.* Information Processing and Management, 49(1) 2013, pp. 392–404.
7. 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, pp. 149–156.
8. A. S. Fraenkel and S. T. Klein: *Robust universal complete codes for transmission and compression.* Discrete Applied Mathematics, 64 1996, pp. 31–55.
9. H. F. Gaines: *Cryptanalysis, a study of ciphers and their solution.* Dover Publ. Inc. New York, 1956.
10. R. González, S. Grabowski, V. Mäkinen, and G. Navarro: *Practical implementation of rank and select queries*, in Poster Proceedings of 4th Workshop on Efficient and Experimental Algorithms (WEA05), Greece (2005), 2005, pp. 27–38.
11. R. Grossi, A. Gupta, and J. S. Vitter: *High-order entropy-compressed text indexes*, in Proceedings of the 14th Annual SIAM/ACM Symposium on Discrete Algorithms (SODA), 2003, pp. 841–850.
12. H. S. Heaps: *Information Retrieval Computational and Theoretical Aspects*, Academic Press, New York, 1978.
13. G. Herdan: *The Advanced Theory of Language as Choice and Chance*, Springer-Verlag, New York, 1966.
14. G. Jacobson: *Space efficient static trees and graphs*, in Proceedings of FOCS, 1989, pp. 549–554.
15. S. T. Klein: *Skeleton trees for the efficient decoding of Huffman encoded texts.* in the Special issue on Compression and Efficiency in Information Retrieval of the Kluwer Journal of Information Retrieval, 3 2000, pp. 7–23.
16. S. T. Klein and M. Kopel Ben-Nissan: *On the usefulness of Fibonacci compression codes.* The Computer Journal, 53 2010, pp. 701–716.
17. D. Knuth: *The Art of Computer Programming, Sorting and Searching*, vol. III, Addison-Wesley, Reading, MA, 1973.
18. M. Külekci: *Enhanced variable-length codes: Improved compression with efficient random access*, in Proc. Data Compression Conference DCC–2014, Snowbird, Utah, 2014, pp. 362–371.
19. D. Morrison: *Patricia - practical algorithm to retrieve information coded in alphanumeric.* Journal of the ACM, 15(4) 1968, pp. 514–534.
20. G. Navarro and E. Providel: *Fast, small, simple rank/select on bitmaps.* Experimental Algorithms, LNCS, 7276 2012, pp. 295–306.

21. D. Okanohara and K. Sadakane: *Practical entropy-compressed rank/select dictionary*, in Proc. ALENEX, SIAM, 2007.
22. J. Pesonen: *Word inflexions and their letter and syllable structure in finnish newspaper text.* Research Rep. 6, Dept. of Special Education, University of Jyräskylä, Finland (in Finnish, with English summary), 1971.
23. R. Raman, V. Raman, and S. Rao Satti: *Succinct indexable dictionaries with applications to encoding k-ary trees and multisets.* Transactions on Algorithms (TALG), 2007, pp. 233–242.
24. D. Shapira and A. Daptardar: *Adapting the Knuth-Morris-Pratt algorithm for pattern matching in Huffman encoded texts.* Information Processing and Management, IP & M, 42(2) 2006, pp. 429–439.
25. H. Williams and J. Zobel: *Compressing integers for fast file access.* The Computer Journal, 42(30) 1999, pp. 192–201.