Smaller Compressed Suffix Arrays*

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

[1]*Department of Computer Science, Ariel University, Ariel 40700, Israel*
[2]*Department of Computer Science, Bar Ilan University, Ramat-Gan 52900, Israel*
Email: benzakate@gmail.com, tomi@cs.biu.ac.il, shapird@g.ariel.ac.il

An alternative to compressed suffix arrays is introduced, based on representing a sequence of integers using Fibonacci encodings, thereby reducing the space requirements of state of the art implementations of the suffix array, while retaining the searching functionalities. Empirical tests support the theoretical space complexity improvements, and show that there is no deterioration in the processing times.

Received 00 Month 2017; revised 00 Month 2017

1. INTRODUCTION

The research field of *compressed data structures* [2] combines two different, yet correlated, disciplines in computer science: data compression and data structures, and has recently attracted much attention, mainly because of the necessity of handling *Big Data*. The goal is storing massive information amounts in compact memory space, while still supporting efficient query processing.

One of the fundamental *succinct data structures* using only $o(Z)$ bits beyond the information-theoretic lower bound of Z bits, is *bitmaps*, that support fast implementations of operations known as rank and select. These are defined for any bit vector B of length n, bit value $b \in \{0, 1\}$ and index $0 \le i < n$ as:

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

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

Efficient implementations for rank and select are due to Jacobson [3], Raman et al. [4], and Navarro and Providel [5], to list only a few.

The *Wavelet tree* (WT) data structure can be seen as an extension of the rank and select operations on bits to a general alphabet, and may be stored in a compressed form in case it is constructed by means of some compression method such as an entropy encoder, described precisely as follows. A Wavelet tree, suggested by Grossi et al. [6], is a data structure which reorders the bits of the compressed file into an alternative form, thereby enabling direct access, as well as rank and select operations to a general alphabet. WT can be constructed upon any prefix-free code, and their topology is determined by this code. The internal nodes of the WT are annotated with bitmaps: the root of the WT holds the bitmap obtained by concatenating the *first* bit of the codewords of each of the encoded elements, which we shall call characters, in the order they appear in the encoded text. The left and right children of the root hold the bitmaps obtained by concatenating, again in the given order, the *second* bit of each of the codewords starting with 0, respectively of those starting with 1. This process is repeated recursively on the sub-sequences of characters whose codewords start with 0 and 1, respectively.

Recently, several variants have been proposed for adapting the Wavelet tree to various coding schemes. Brisaboa et al. [7] use WT on Byte-Codes. In another work, Brisaboa et al. [8] introduce directly accessible codes (DACs) by integrating rank data structures into variable lengths codes. Külekci [9] suggests the usage of WT for *Elias* and *Rice* variable length codes. The method is based on handling separately the unary and binary parts of each codeword so that random access is supported in constant time. As an alternative implementation, the usage of a WT over the lengths of the unary parts of each of the Elias or Rice codewords is proposed, while storing the corresponding binary parts; this allows direct access in time $\log r$, where $r = O(\log \log M)$ is the number of distinct values of the lengths that are encoded in unary in the file.

Klein and Shapira [10] apply a pruning strategy to Wavelet trees based on *Fibonacci Codes*, so that in addition to supporting improved rank, select and random access to the corresponding Fibonacci encoded file, the size of the Fibonacci based WT is reduced. A new data structures for reducing the space overhead of a Huffman shaped WT is used in [11] to support extract queries from a text by means of a Skeleton Huffman tree and its reduced variants [10]. Range decoding in WT is proposed in [12].

*This is an extended version of a paper that has been presented at the Prague Stringology Conference (PSC'18) in 2018, and appeared in its proceedings [1].

This paper concentrates on another compressed data structure known as a *compressed suffix array* (CSA), introduced by Grossi and Vitter [13] as a text index that uses $2n \log \sigma$ bits in the worst case, and supports $O(m)$ processing time for searching a pattern of length m.

Given a text and some pattern we wish to locate in it, the *suffix array* of the text is a *self index*, meaning that the retrieval is done directly on the suffix array itself, without the use of the text. That is, the text is implicitly encoded, and the searching process decompresses only the necessary portion of the text. More formally, let T be a string of length $n-1$ over an alphabet Σ of size σ, which we assume in this paper to be constant. A *suffix array* for $T\$$, $\$ \notin \Sigma$, is an array $SA[0 : n-1]$ of the indices of all the n suffixes of $T\$$ which have been arranged in lexicographic order. By convention, \$ is lexicographically smaller than all other characters.

Suffix arrays have been introduced by Manber and Myers [14], and are more space efficient than *suffix trees* (compact tries), because suffix trees generally require additional space to store all the internal pointers in the tree. Sadakane [15] extends the searching functionality of a CSA to a self index, and proves that it uses search time $O(m \log n)$, and space $\epsilon^{-1} n H_0 + O(n \log\log \sigma)$ bits, where $0 < \epsilon < 1$ and $\sigma \leq \log^{O(1)} n$, H_0 being the 0-order empirical entropy of T.

Grossi et al. [6] present an implementation of compressed suffix arrays that asymptotically achieves entropy space as well as fast pattern matching. More precisely, the CSA uses $nH_k + O(\frac{n \log\log n}{\log_\sigma n})$ bits and $O(m \log \sigma + \text{polylog}(n))$ searching time, where H_k is the k-th order empirical entropy of T.

Ferragina and Manzini [16] introduce the *FM-index*: a text index based on the Burrows-Wheeler Transform [17], which supports efficient pattern matching using the *Backward Search* algorithm. The FM-index uses at most $5nH_k + o(n)$ bits for a constant alphabet size σ, and $O(m + occ \log^{1+\epsilon} n)$ searching time to locate the *occ* occurrences of the pattern. We refer the reader to the book of Navarro [2] for a comprehensive review on compact data structures in general and compressed suffix arrays in particular.

Huo et al. [18] construct a space efficient CSA, and Huo et al. [19] extend their work for the reference genome sequence and propose approximate pattern matching on the compressed suffix array for short read alignment. Their implementation uses $2nH_k + n + o(n)$ bits of space, for $k \leq c \log_\sigma n - 1$ and any $c < 1$. They report on extensive experiments to evaluate their CSA compression, construction time, and pattern matching processing time performance. The results suggest that their compression performance is better than that of the implementation of Sadakane [15] and the FM-index [16], except for evenly distributed data like that of DNA files.

In this paper we suggest the usage of Fibonacci Codes instead of Elias' C_γ code used in [18, 19], and show how decompression can be avoided using our scheme. Elias [20] considered mainly two *fixed* codeword sets, C_γ and C_δ, in what he calls *universal* codes, in which the integers are represented by binary sequences, a sample of which appears in Table 1.

The implementation of the Fibonacci based suffix array requires $1.44\, n\, H_k + n + o(n)$ bits of space, for $k \leq c \log_\sigma n - 1$ and any $c < 1$, while retaining the searching functionalities. Empirical results support this theoretical bound improvement, and show that on most files, our implementation saves space as compared to the one of [18, 19].

The paper is organized as follows. We recall the definitions related to Fibonacci coding in Section 2, and those of Compressed Suffix Arrays in Section 3. Section 4 then presents implementation details of the CSA including its space analysis and the summation process applied on the compressed form. Section 5 describes the functionality as self-index supported by CSAs. Finally, empirical results are given in Section 6.

2. FIBONACCI ENCODINGS

The lengths of the C_γ codewords grow logarithmically, which yields good asymptotic behavior. However, C_γ is then often efficient only for quite large alphabets, whereas the number of different elements in the CSA for natural language texts is usually small. The same is true for several other universal codeword sets such as ETDC [21] and (s, c)-dense codes [22]. This was also the motivation of using C_γ instead of the asymptotically better C_δ representation in the implementation of Huo et al. [18]. The Fibonacci code is yet another universal variable length encoding of the integers, based on the sum of Fibonacci numbers rather than on the sum of powers of 2, as in the *standard* binary representation. More precisely, any number $x \geq 0$ can be uniquely represented by a binary string $b_r b_{r-1} \cdots b_2 b_1$, with $b_i \in \{0, 1\}$, such that $x = \sum_{i=1}^{r} b_i F_i$, where the Fibonacci numbers F_i are defined by:

$$F_i = F_{i-1} + F_{i-2} \qquad \text{for } i \geq 1,$$

and the boundary conditions

$$F_0 = 1 \qquad \text{and} \qquad F_{-1} = 0.$$

The uniqueness of the representation for every integer x is achieved by building the representation according to the following procedure: find the largest Fibonacci number F_r smaller than or equal to x, and repeat the process recursively with $x - F_r$. For example, $79 = 55+21+3 = F_9+F_7+F_3$, so its Fibonacci representation would be 101000100. As a result of this encoding scheme, there are never consecutive Fibonacci numbers in any of these sums, implying that in the corresponding binary representation, there are no adjacent 1s. It thus

i	C_γ	C_δ	Fib_1	Fib_2
1	**1**	**1**	11	**1**
2	**01 0**	010 0	**011**	**101**
3	**01 1**	010 1	0011	1001
4	001 00	011 00	**1011**	10001
5	001 01	011 01	00011	10101
6	**001 10**	**011 10**	**10011**	100001
7	**001 11**	**011 11**	**01011**	101001
8	0001 000	00100 000	**000011**	**100101**
9	0001 001	00100 001	**100011**	1000001
10	0001 010	00100 010	**010011**	1010001
30	00001 1110	00101 1110	**10001011**	100000101
100	0000001 100100	**00111 100100**	**0010100001**1	100100100001

Table 1: *Several codewords of universal codes C_γ, C_δ, Fib_1 and Fib_2.*

suffices to precede the Fibonacci based representation of any integer by a single 1-bit, which can act like a comma, to obtain a uniquely decipherable code.

The properties of Fibonacci codes have been exploited in several useful applications: robustness to errors [23], direct access [24], fast decoding and compressed search [25, 10], compressed matching in dictionaries [26], rewriting codes for flash memory [27], etc. The present work is yet another application of this idea.

One variant of the Fibonacci code, denoted here by Fib_1, simply reverses the codewords in order to achieve an instantaneous code [28]. The adjacent 1s are then at the right instead of at the left end of each codeword, yielding a prefix code, a sample of which is presented in Table 1 in the column headed by Fib_1.

Another variant, denoted here by Fib_2, was introduced in [28], and found to be often preferable for the $\Delta\Phi$ encoding described in the following section. The set of codewords Fib_2 is constructed from the set Fib_1 by omitting the rightmost 1-bit of every codeword and prefixing each codeword by 10; for example, 0100011 (for encoding the number 15 in Fib_1) is transformed into **10**010001 (for encoding the number 16 in Fib_2). As a result, every codeword now starts and ends with a 1-bit, so codeword boundaries may still be detected by the occurrence of the string 11. Since, as a result of this transformation, the shortest codeword 101 is of length three, one may add 1 as a single codeword of length 1, which explains the shift in the indices of corresponding codewords. Table 1 presents several codewords for Elias C_γ and C_δ, presented in the first two columns, followed by Fib_1 and Fib_2. For each presented value, the codewords of shortest length are emphasized, unless all are of the same length. Although most of the codewords of Fib_1 are the shortest, its disadvantage relative to the other codes is the encoding of the most frequent value that uses two bits instead of a single one. This was found to be empirically crucial for our data sets, as the number 1 was by far the most common value to be encoded.

3. COMPRESSED SUFFIX ARRAY

We recall that a suffix array (SA) for $T\$$, where T is a string over Σ and $\$ \notin \Sigma$, is an array $SA[0:n-1]$ of the indices of the suffixes of $T\$$, stored in lexicographical order. That is, if $SA[i]=j$ then the suffix $T[j:n-1]$ starting at position j of T, is the item indexed i in the lexicographically sorted list of all n suffixes of $T\$$. The inverse function $SA^{-1}[j]$ gives the position of $T[j:n-1]$ in the sorted list of the suffixes of T.

The numbers in a suffix array can be stored using $n\log n$ bits, as they are a permutation of the numbers $\{0,\ldots,n-1\}$, that require $\log n! = \Omega(n\log n)$ bits, at least. However, not all permutations correspond to actual suffix arrays, as there are only σ^{n-1} different texts T of length $n-1$ over Σ. Thus a better lower bound is, in fact, $n\log\sigma$ bits. Grossi and Vitter [13] improve the space requirements of a suffix array by decomposing it based on the *neighbor function* Φ defined as follows:

$$\Phi[i]=j, \qquad \text{if } \ SA[j]=(1+SA[i]) \bmod n.$$

The function Φ can also be rewritten as:

$$\Phi[i]=SA^{-1}[(1+SA[i]) \bmod n].$$

If $SA[i]=j$, that is, it refers to the suffix $T[j:n-1]$, and i' is the index such that $SA[i']=j+1$, which refers to the following suffix $T[j+1:n-1]$, then the neighbor function connects these values by assigning $\Phi[i]=i'$. Therefore, if $SA[i]=j$ then $SA[\Phi[i]]=j+1$, $SA[\Phi[\Phi[i]]]=j+2$, and generally, $SA[\Phi^{(k)}[i]]=j+k$.

It has been shown that the values of Φ at consecutive positions referring to suffixes that start with the same symbol must be increasing. This claim is explained as follows. Let i and $i+1$ be two adjacent indices in SA that correspond to suffixes that start by the same symbol. The index i cannot be 1, as the symbol at the first position corresponds to \$ that occurs only once. Let $j=SA[i]$ and $j'=SA[i+1]$. Following our assumption that they belong to suffixes starting with the same symbol, we get that $T[j]=T[j']$. Since

	0	1	2	3	4	5	6	7	8	9	10	11
T	m	i	s	s	i	s	s	i	p	p	i	\$
SA	11	10	7	4	1	0	9	8	6	3	5	2
SA^{-1}	5	4	11	9	3	10	8	2	7	6	1	0
Φ	5	0	7	10	11	4	1	6	2	3	8	9
$T[SA]$	\$	i	i	i	i	m	p	p	s	s	s	s
Φ_c	$\Phi_{\$}$	$\Phi_{\mathtt{i}}$				$\Phi_{\mathtt{m}}$	$\Phi_{\mathtt{p}}$		$\Phi_{\mathtt{s}}$			

TABLE 2: *CSA example for* $T = \mathtt{mississippi\$}$

	0	1	2	3	4	5	6	7	8	9	10	11
SAM	5				11				2			
$\Delta\Phi$		7	7	3		5	9	5		1	5	1
$C_\gamma(\Delta\Phi)$	00111 00111 011				00101 0001001 00101				1 00101 1			
SB	0								30			
B	0				13				0			
$\mathsf{Fib}_1(\Delta\Phi)$	01011 01011 0011				00011 100011 00011				11 00011 11			
SB	0								30			
B	0				14				0			
$\mathsf{Fib}_2(\Delta\Phi)$	101001 101001 1001				10101 1000001 10101				1 10101 1			
SB	0								33			
B	0				16				0			
Φ	5	0	7	10	11	4	1	6	2	3	8	9

TABLE 3: *Super blocks and regular blocks parsing of* Φ *for* $a = 8$ *and* $b = 4$ *using three different implementations:* C_γ, Fib_1 *and* Fib_2 *codes.*

$T[j : n] \prec T[j' : n]$, it follows that $T[j+1, n] \prec T[j'+1, n]$, and $j'+1$ appears to the right of $j+1$ in SA. The position where $j'+1$ appears in SA is $SA^{-1}[j'+1] = SA^{-1}[SA[i+1]+1] = \Phi[i+1]$. Using the same argument, the position where $j+1$ appears in SA is $\Phi[i]$, thus, $\Phi[i] < \Phi[i+1]$.

As Φ is an increasing function for suffixes starting with the same symbol, Φ can be partitioned into σ arrays of increasing numbers $\Phi_c = [0 : n_c - 1]$, for all $c \in \Sigma$, where n_c is the number of occurrences of the character c in the text. As an example, consider the text $T = \mathtt{mississippi\$}$. The text T, its suffix array SA, its inverse function SA^{-1}, and Φ are given in the first rows of Table 2. The last row partitions the Φ row into subintervals, denoted by $\Phi_{\mathtt{i}}, \Phi_{\mathtt{m}}, \Phi_{\mathtt{p}}$ and $\Phi_{\mathtt{s}}$, each referring to a different character of T. The first cell does always refer to the special character \$, denoted by $\Phi_{\$}$. To better understand this partition, we have preceded it with a row showing $T[SA[i]]$ at position i, the first character of the corresponding suffix.

The implementation of CSA used in [18, 19] applies differential encoding. Instead of Φ itself, the values $\Delta\Phi[i] = \Phi[i] - \Phi[i-1]$ are encoded, except for the first which is not defined. More precisely, since universal codes are designed for positive integers, we define, for $i > 0$

$$\Delta\Phi[i] = \begin{cases} \Phi[i] - \Phi[i-1] & \text{if the difference is} > 0, \\ \Phi[i] - \Phi[i-1] + n & \text{if the difference is} < 0. \end{cases}$$

These differences are then encoded using the C_γ encoding method of Elias.

To provide faster access to the C_γ encoded sequence S of integers, which we denote as $C_\gamma(S)$, we partition it into so-called *super-blocks*, which in turn are sub-partitioned into *blocks*, and three auxiliary tables SB, B and SAM are defined. For given values a and b, which are defined in the following paragraph, $\mathsf{SB}[0 : \frac{n}{a} - 1]$ stores the starting position of the encoding of each super-block in $C_\gamma(S)$, i.e., the total number of bits in super-blocks preceding the current super-block; $\mathsf{B}[0 : \frac{n}{b} - 1]$ stores the starting position in $C_\gamma(S)$ of the encoding of every block relative to the beginning of the super-block containing this block; and $\mathsf{SAM}[0 : \frac{n}{b} - 1]$

contains the first in each block of Φ values as a sample from which the omitted values can be derived.

The sizes of the super-blocks and blocks are chosen in [18] as $a = \log^3 n$ elements and $b = \log^2 n$ elements. While the super-blocks store the absolute number of bits up to that position, the blocks record the relative position with respect to the beginning of the super-block. Table 3 uses our running example for $a = 8$ and $b = 4$ illustrating how the neighbor function Φ can be derived on the basis of the differences $\Delta\Phi$ and the SAM values. The third line of Table 3 displays the Elias' C_γ encoding of $\Delta\Phi$, followed by two lines for the values in the super-blocks and blocks. The table also contains the encodings corresponding to the Fibonacci variants Fib_1 and Fib_2, as well as the matching SB and B arrays.

Once the partition into super-blocks and blocks is given, the elements to be stored are the $\Delta\Phi$ values for each block; the first value of each block is not defined and therefore omitted. In other words, the sequence $\Delta\Phi[ib+1], \Delta\Phi[ib+2], \ldots, \Delta\Phi[(i+1)b-1]$ is encoded, and $\Delta\Phi[ib]$ are not defined, for all $0 \le i < \lfloor \frac{n}{b} \rfloor$. For the completeness of the example, we have copied the Φ array at the bottom of the table.

4. IMPLEMENTATION DETAILS

Using the encoding and decoding alternatives, as well as the arrays of the samples, super-blocks and blocks, the Φ values can be extracted in time $O(b)$ as follows. The arguments of the decoding function, denoted by $\mathcal{D}(\mathcal{E}, s, \ell)$, are: the encoded array $\mathcal{E}$ to be decoded, the starting position s within $\mathcal{E}$, and the number of codewords ℓ to be decoded and then summed up. The values of Φ are then computed using:

$$\Phi[i] = \Big[\mathsf{SAM}\big[\lfloor \tfrac{i}{b} \rfloor\big] + \mathcal{D}\Big(\mathcal{E},\ \mathsf{SB}\big[\lfloor \tfrac{i}{a} \rfloor\big] + \mathsf{B}\big[\lfloor \tfrac{i}{b} \rfloor\big],\ i \bmod b\Big)\Big] \bmod n. \tag{1}$$

To obtain $\Phi[i]$, SB and B are accessed to determine the corresponding bit position within $\mathcal{E}$. Starting at that position, $i \bmod b$ codewords are decoded and added to the sample values stored in SAM. The mod n at the end takes care of the negative values of the differences $\Delta\Phi[i]$. Recall that n has been added to each such negative number; these technical additions are cancelled by the modulus.

As an example using Fib_2,

$$\begin{aligned}\Phi[6] &= \big[\mathsf{SAM}[\tfrac{6}{4}] + \mathcal{D}(\mathcal{E}, \mathsf{SB}[\tfrac{6}{8}] + \mathsf{B}[\tfrac{6}{4}], 6 \bmod 4)\big] \bmod n \\ &= \big[\mathsf{SAM}[1] + \mathcal{D}(\mathcal{E}, \mathsf{SB}[0] + \mathsf{B}[1], 2)\big] \bmod 12 \\ &= \big[11 + \mathcal{D}(\mathcal{E}, 0 + 16, 2)\big] \bmod 12.\end{aligned}$$

Two consecutive values, 10101 and 1000001 are then retrieved and decoded as 5 and 9 (see Table 1) and are added to 11, so that the final result (11+5+9) mod 12 = 1 is returned. The relevant values of the example are highlighted in bold in Table 3.

4.1. Space Analysis

Huo et al. [18] prove that the space used for the Elias C_γ based $\Delta\Phi$ encoding is $2nH_k + n + o(n)$ bits in the worst case for any $k \le c \log_a n - 1$ and any constant $c < 1$. Navarro [2] shows that if $\Delta\Phi$ is encoded using C_δ, the space for CSA is $nH_k + n + O(n)$.

C_γ and C_δ require $2\lfloor \log x \rfloor + 1$ and $\lfloor \log x \rfloor + 1 + 2\lfloor \log(\lfloor \log x \rfloor + 1) \rfloor$ bits, respectively, to encode the number x. To evaluate the corresponding Fibonacci codeword lengths, let F_r be the largest Fibonacci number smaller than or equal to the given number x. Then r bits are necessary to encode x. A well known approximation of the Fibonacci number F_r is $\frac{\phi^r}{\sqrt{5}}$, where $\phi = \frac{1+\sqrt{5}}{2} = 1.618$ is the golden ratio. This approximation can be used to derive our estimate of the length of the encoding, because the difference between F_r and $\frac{\phi^r}{\sqrt{5}}$ is just $\frac{\hat{\phi}^r}{\sqrt{5}}$, where $\hat{\phi} = \frac{1-\sqrt{5}}{2} = -\frac{1}{\phi}$, so that $|\hat{\phi}| < 1$ and $\hat{\phi}^r$ decreases exponentially with growing r.

From the fact that $F_r \le x < F_{r+1}$ we may thus extract a bound for r to be

$$\begin{aligned} r \le \log_\phi(\sqrt{5}x) &= \log_\phi \sqrt{5} + (\log_\phi 2)\ \log_2 x \\ &= 1.672 + 1.4404\ \log_2 x. \end{aligned}$$

That is, the lengths of Fibonacci codewords are asymptoticly between those of C_γ and C_δ. However, in practice, Fibonacci codes may be preferable in case the numbers are not uniformly distributed, as in our application of compressed suffix arrays.

Emulating the space analysis given in [18] for the Elias C_γ encoded CSA, replacing the length estimates of $2 \log x$ for a value x by $1.44 \log x$, we get that at most $1.44\, n\, H_k(1 + o(1)) + O(n)$ bits are needed for the Fibonacci based representation of the CSA, for any $k \le c \log_\sigma n - 1$, and any constant $c < 1$, where H_k is the k-th order empirical entropy.

4.2. Compressed addition

According to equation (1), in order to obtain $\Phi[i]$, $i \bmod b$ codewords need to be decoded. The traditional approach is to decode each codeword and add the decoded values. One of the advantages of using a Fibonacci based representation of the integers is that it is possible to perform this addition directly on the compressed form of the CSA, without individually decoding each summand.

To add $i \bmod b$ Fibonacci encoded numbers, first strip the appended 1 for Fib_1 or the prepended 10 for Fib_2 (except for the first codeword 1, which is given a special treatment), and pad, if necessary, the shorter codewords with zeros at their right end so that all representations are of equal length ℓ. Considering this as an $(i \bmod b) \times \ell$ matrix, we record the number of 1-bits in each column into an array $W[1:\ell]$. The sought result is obtained by summing $\sum_{j=1}^{\ell} W[j]F_j$ for Fib_1, or by summing $\sum_{j=1}^{\ell} W[j]F_j + (i \bmod b)$ for Fib_2.

For example, assume that $(i \bmod b) = 5$ differences $\Delta\Phi[i]$, $2, 3, 5, 6$, and 4, should be added to obtain $2 + 3 + 5 + 6 + 4 = 20$. They are represented in Fib_1 as 011, 0011, 00011, 10011, and 1011, respectively.

The steps proposed are:

1. Strip the appended 1: resulting in 01-1, 001-1, 0001-1, 1001-1, and 101-1.
2. Pad the shorter codewords with 0s so that all of them are of length $\ell = 4$: 0100, 0010, 0001, 1001, and 1010.
3. Regard them as an $(i \bmod b) \times \ell = 5 \times 4$ matrix:

$$\begin{matrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{matrix}$$

4. Record the number of ones in each column in $W[1:4] = [2, 1, 2, 2]$.
5. $\sum_{j=1}^{\ell} W[j]F_j$ is $2 \cdot 1 + 1 \cdot 2 + 2 \cdot 3 + 2 \cdot 5 = 20$, as expected.

Encoding the same example, $2, 3, 5, 6, 4$, using Fib_2, attains 101, 1001, 10101, 100001, and 10001, respectively. Stripping the prepended 10 and padding by 0s, we receive 1-000, 01-00, 101-0, 0001, and 001-0. Finally, putting them in a matrix:

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{matrix}$$

$W[1:4]$ is then $[2, 1, 2, 1]$, and $\sum_{j=1}^{\ell} W[j]F_j + (i \bmod b)$ is $2 \cdot 1 + 1 \cdot 2 + 2 \cdot 3 + 1 \cdot 5 + 5 = 20$, as expected.

In fact, the matrix should be understood as a conceptual tool to explain the algorithm, and in an actual implementation the extracted bits are processed from left to right, only up to the rightmost 1-bits, which are the leading bits in the Fibonacci encoded numbers. The overall processing time of all these numbers is thus proportional to the number of bits used for their encoding, which is, asymptotically and empirically on our test files, smaller than the time necessary for the corresponding C_γ encodings used in [18].

Similarly, the Elias C_γ code could be partially used directly in its compressed form, as the summation of integers represented by codewords of the same length can be evaluated by adding the binary parts, and copying the common unary part, or extending it by a single 1-bit if there has been a carry in the addition. However, handling codewords of different lengths is more involved: in order to locate the binary parts, the number of preceding 1's in the unary part of each of the codewords has to be counted, whereas in the Fibonacci encoding, the codeword limits are easily detected by the occurrence of a pair of adjacent 1-bits.

A different approach for storing the increasing Φ values of the CSA suggests using the *Elias-Fano* encoding [29]. Based on this approach, [30] propose an enhanced block based representation of the Φ values that provides improved compression results as well as faster backward search for the counting query.

5. EXTRACT, COUNT AND LOCATE QUERIES ON CSA

For a given pattern P, the compressed suffix array CSA(T) supports the following operations without any use of T:

1. EXTRACT(ℓ, r) - for returning the substring $T[\ell : r]$.
2. COUNT(P) - for returning the number of occurrences of P in T.
3. LOCATE(P) - for returning the list of all occurrences of P in T.

Assume the alphabet $\Sigma = \{c_0, \ldots, c_{\sigma-1}\}$ is given in lexicographic order. In order to query the CSA, one uses an additional table C of size σ. It stores at entry i the total number of characters in T that are lexicographically smaller than c_i, for each character c_i. Formally, $C[i] = \sum_{j=1}^{i-1} occ(c_j)$, where $occ(c_j)$ denotes the number of occurrences of c_j in T. For simplicity, a sentinel element $C[\sigma] = |T| = n$ is added. The space used for C is $\sigma \log n$ bits. We shall use the inverse array C^{-1}, which is not stored explicitly, and its values are retrieved using binary search on array C. Define a function $\mathsf{ord}(x)$ giving the index, starting from 0, of the character x within Σ in lexicographical order. For the particular order of our alphabet, $\mathsf{ord}(c_i) = i$.

For our running example $T =$ `mississippi$`, the array C is

	\$	*i*	*m*	*p*	*s*	
C:	0	1	5	6	8	12

5.1. The EXTRACT Query

Next we show how to perform EXTRACT(ℓ, r) on the CSA for computing the substring $str = T[\ell : r]$. The index $k = SA^{-1}[\ell]$ is the location corresponding to suffix $T[\ell, n-1]$ in the suffix array, i.e., $T[\ell, n-1]$ is the element indexed k of the suffixes of T in lexicographic order. Thus, the first character of str, $T[\ell]$, is $C^{-1}[k]$. For the following characters, one must follow the Φ function and compute iteratively $k \leftarrow \Phi(k)$ and $C^{-1}[k]$, $r - \ell$ times, as presented in Algorithm 1.

As an illustration, we perform EXTRACT(6, 8) on our running example. $SA^{-1}[6] = 8$ and $C^{-1}[8] =$ `s` is the first character; $k = \Phi[8] = 2$, and $C^{-1}[2] =$ `i` is the second character. Finally, $k = \Phi[2] = 7$ and the last character is $C^{-1}[7] =$ `p`, so the returned string is $str =$ `sip`.

Note that although a contiguous region of T is sought, the Φ values are not necessarily adjacent to each other, and therefore each Φ value should be extracted independently. This non adjacency of the Φ values is similar to what happens in the backward search of the FM-index [16], where the sequence of intervals obtained when processing the characters right to left, are not contained in each other. However, some Wavelet trees implementations mentioned in the introduction do follow continuous regions.

In [11], the partial decoding of a contiguous portion of the file, or even its full decoding, is significantly accelerated relative to repeatedly performing random accesses on the consecutive indices.

```
EXTRACT($\ell, r$)
    $k \leftarrow SA^{-1}[\ell]$
    for $j \leftarrow \ell$ to $r$ do
        $str[j-\ell] \leftarrow C^{-1}[k]$
        $k \leftarrow \Phi[k]$
    return $str$
```

Algorithm 1: extract *a range in* T.

5.2. count and locate Queries

An occurrence of a given pattern P in T is a prefix of some suffix of T. The indices of several such occurrences of the same pattern appear in consecutive cells of SA. In particular, the suffixes that start in the text T at positions indexed $SA[C[i]], SA[C[i]+1], \cdots, SA[C[i+1]-1]$ all have c_i as their first character. As mentioned above, the suffixes of T can be reconstructed using the function Φ. The query locate(P) returns the range $[start, end]$ in SA of suffixes of T that start with P if P occurs in T, or an empty range, otherwise. That is, it returns the indices $\{SA[start], SA[start+1], \ldots, SA[end]\}$ that are the positions of the suffixes that are prefixed by P.

The algorithm locate(P) is presented for the completeness of the paper. The algorithm searches for the range bounds, going backwards from p_m to p_1. The initial interval is $[0, n-1]$, referring to the entire range of the text. At each iteration of the algorithm, the range for $P[i:m]$ is computed given the range for $P[i+1:m]$. The final range is a sub-interval of the suffix array where P occurs as a prefix of the corresponding suffixes. The computation for count(P) is simply the size of the interval.

```
LOCATE($P$)
$(\ell, r) \leftarrow (0, n-1)$
for $i \leftarrow |P|$ down-to 1 do
    $B \leftarrow \{j \mid \ell \le \Phi(j) \le r\}$
    $D \leftarrow \{C[\mathrm{ord}(p_i)], C[\mathrm{ord}(p_i)]+1, \ldots, C[\mathrm{ord}(p_i)+1]-1\}$
    if $B \cap D = \emptyset$ then
        return "pattern not found"
    $(\ell, r) \leftarrow (\min(B \cap D),\ \max(B \cap D))$
return $(\ell, r)$
```

Algorithm 2: locate P *in* T.

As an example, let $P =$ `issi`.

1. We start with the initial range $[0, 11]$ and the rightmost character of P, `i`; B is then the entire range $[0, 11]$, and since $\mathrm{ord}(p_m) = \mathrm{ord}(\mathtt{i}) = 1$, $C[1] = 1$, thus D is the range $[1, \ldots, 4]$, which yields $(\ell, r) = (1, 4)$.
2. The next character to be processed is the right `s`, with range $[1, 4]$. B is then $\{5, 6, 8, 9\}$, and since $\mathrm{ord}(p_3) = \mathrm{ord}(\mathtt{s}) = 4$, $C[4] = 8$, thus D is the range $[8, 9, 10, 11]$, which yields $(\ell, r) = (8, 9)$.
3. The next character to be processed is the left `s`, with range $[8, 9]$. B is then $\{10, 11\}$, and since $\mathrm{ord}(p_2) = \mathrm{ord}(\mathtt{s}) = 4$, $C[4] = 8$, thus D is the range $[8, \ldots, 11]$ as in the previous iteration, which yields $(\ell, r) = (10, 11)$.
4. Finally, the last character to be processed is the left `i`, starting with range $[10, 11]$. B is then $\{3, 4\}$, and D is $[1, \ldots, 4]$ as for the first `i` processed earlier, which yields as final range $(\ell, r) = (3, 4)$.

The SA at positions $[3, 4]$ are 4 and 1, as can be seen in Table 1, which are the positions in T that start with $P =$ `issi`.

In order to improve the running time, the subset B in Algorithm 2 does not have to be computed explicitly. Recall that Φ_c is increasing for each character c, thus for each character p_i of P, the search for ℓ and r can be done using binary search within the range Φ_{p_i}. The total running time in the worst case is thus $O(m \log n)$. However, the range is of size $\theta(n)$ only for the first few steps and is usually narrowed after several characters to a much smaller range, so the running time is smaller in practice.

6. EXPERIMENTAL RESULTS

We considered the same test files as Huo et al., taken from the Pizza & Chili[3] as well as from the Canterbury[4] corpora. We used the implementation[5] of [18] and adapted it to encode the $\Delta\Phi$ values with Fib_1 and Fib_2, instead of with Elias' C_γ code. Both the sizes of the CSA as well as the net sizes of the different encodings without the space overhead of the super-blocks and blocks have been considered.

The first column of Table 4 reports the sizes in MB of the datasets. The following columns present the compression ratio of the net encodings of $\Delta\Phi$ using these universal codes, in which the compression ratio is defined as the storage occupied by the net encodings, divided by the size of the original file. We also added Elias' C_δ code, as it is asymptotically the best of these four universal codes alternatives. Table 5 presents the compression ratio of the entire CSA, including the

[3] http://pizzachili.dcc.uchile.cl
[4] http://corpus.canterbury.ac.nz
[5] https://github.com/Hongweihuo-Lab/CSA

overhead of the blocks, super-blocks and samples, for the C_γ, Fib_1 and Fib_2 encodings. The best values in each row are emphasized.

Name	size (MB)	C_γ	C_δ	Fib_1	Fib_2
Canterbury					
E.coli	4.42	**0.435**	0.488	0.460	0.458
Bible	3.86	0.348	0.357	0.403	**0.342**
world192	2.36	0.329	0.327	0.391	**0.317**
news	0.36	0.494	0.486	0.508	**0.469**
book1	0.73	0.477	0.490	0.495	**0.467**
paper1	0.05	0.480	0.480	0.500	**0.460**
Kennedy	0.98	0.428	0.402	0.463	**0.388**
Pizza & Chili					
dna	100	**0.403**	0.450	0.438	0.422
proteins	100	0.655	0.649	**0.622**	0.627
xml	100	0.229	0.232	0.321	**0.225**
sources	100	0.319	0.317	0.385	**0.307**
english	100	0.375	0.382	0.424	**0.368**

Table 4: *Compression ratio for net encoding of $\Delta\Phi$.*

	C_γ	Fib_1	Fib_2
Canterbury			
E.coli	**0.57**	0.59	0.59
Bible	0.48	0.53	**0.47**
world192	0.46	0.52	**0.45**
news	0.62	0.63	**0.59**
book1	0.60	0.61	**0.59**
paper1	0.62	0.63	**0.60**
Kennedy	0.55	0.59	**0.51**
Pizza & Chili			
dna	**0.56**	0.59	0.58
proteins	0.81	**0.78**	0.78
xml	0.38	0.48	**0.38**
sources	0.48	0.54	**0.46**
english	0.53	0.58	**0.52**

Table 5: *Compression ratio including overhead of blocks and super blocks*

As can be seen, the overhead incurred by the super-blocks and blocks does not change the order of the relative performance of the coding alternatives on our test data. In both tables the Fib_2 based CSA encoding performs best on most files. For dna and E.coli, C_γ gives the best results, but these are two test files of [18] for which FM-index and Sadakane's CSA implementation produce better results than C_γ. Huo et al. explain this performance by the different frequencies of small values (1 and 2) in $\Delta\Phi$, which tend to be lower in these files than in the others, so it is not surprising that for such files, the performance of the Fibonacci encodings is also inferior to that of C_γ. The other file for which Fib_2 does not produce the most efficient CSA is Proteins, for which it is outperformed by Fib_1.

	C_γ	Fib_1	Fib_2
Canterbury			
E.coli	0.095	0.103	0.102
Bible	0.081	0.087	0.086
world192	0.078	0.082	0.080
news	0.108	0.122	0.119
book1	0.108	0.105	0.105
paper1	0.200	0.340	0.320
Pizza & Chili			
dna	0.273	0.280	0.279
proteins	0.258	0.265	0.264
xml	0.191	0.193	0.191
sources	0.160	0.164	0.163
english	0.253	0.256	0.255

Table 6: *Construction time of the CSA in seconds per MB.*

The sizes presented in Tables 4 and 5 also show that the overhead is approximately 25–42% for the Canterbury Corpus, and about 24–69% for the larger files in the Pizza & Chili Corpus.

For the processing times all experiments were conducted on a machine running 64 bit Windows 10 with an Intel Core i7-8550U @ 1.80–4.0 GHz processor, with 8GB of main memory and 8 MB Cache. For the Fibonacci codes we considered both the variant that replaces the C_γ encoding of [18, 19] by one of the Fibonacci encoding alternatives, and the one that also performs compressed addition, as presented in Section 3.2; the latter are denoted by Fib_1-ca and Fib_2-ca.

Table 6 presents the construction time of each CSA alternative. Each test was run ten times and the results, given in seconds per MB, were averaged. As can be seen, in most cases the construction for C_γ is slightly faster than for the Fibonacci variants. Note that the construction is done only once.

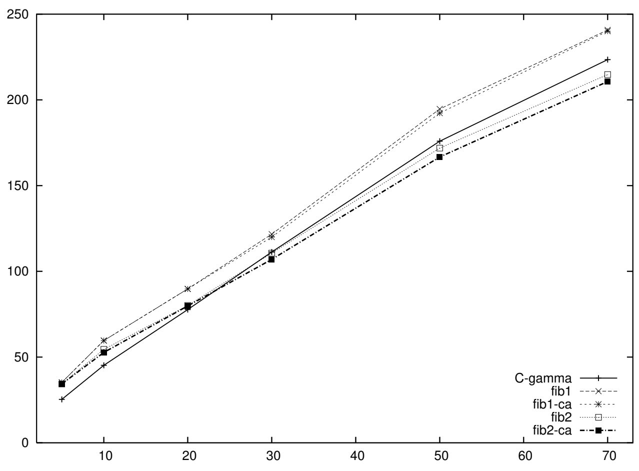

Figure 1: *Counting query of our CSA compared with C_γ (in 10^{-4} sec).*

For the locating and counting queries we considered patterns of different lengths appearing in the text, by randomly selecting their starting positions. As above, each test was repeated ten times for each of the input patterns and the results were averaged.

The counting query processing times are given in units of 10^{-4} seconds, and are presented in Figure 1. The original implementation of Huo et al. was 5-8 times slower than the times reported here, as we improved it for a fair comparison. In all cases, the counting queries on the Fibonacci variants are faster than for C_γ. For Fib_2-CA, which is the best alternative in all cases, the processing is 3.7 to 9.5 times faster than for C_γ. For the locating queries, the processing times of the Fibonacci variants are comparable to those of C_γ in all cases.

7. CONCLUSION

Huo et al. [18] present experiments showing that their CSA implementation is empirically better than the FM-index and Sadakane's CSA implementations on most tested files. We suggest here a Fibonacci based CSA, which generally achieves even better compression performance on the same data-sets, without hurting the processing times.

Acknowledgement: We would like to thank Hongwei Huo for sharing the implementation of [18].

REFERENCES

[1] Benza, E., Klein, S. T., and Shapira, D. (2018) Fibonacci based compressed suffix array. *Prague Stringology Conference 2018, Prague, Czech Republic, August 27-28, 2018*, pp. 3–11.

[2] Navarro, G. (2016) *Compact Data Structures - A Practical Approach.* Cambridge University Press, Cambridge UK.

[3] Jacobson, G. (1989) Space-efficient static trees and graphs. *Proceedings of the 30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October–1 November 1989*, pp. 549–554. IEEE Computer Society, Washington DC.

[4] Raman, R., Raman, V., and Satti, S. R. (2007) Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, **3**, 43.

[5] Navarro, G. and Providel, E. (2012) Fast, small, simple rank/select on bitmaps. *Experimental Algorithms - Proceedings of the 11th International Symposium, SEA 2012, Bordeaux, France, June 7-9*, Lecture Notes in Computer Science, **7276**, pp. 295–306. Springer Verlag, Berlin.

[6] Grossi, R., Gupta, A., and Vitter, J. S. (2003) High-order entropy-compressed text indexes. *Proceedings of the 14th ACM-SIAM Symposium on Discrete Algorithms, Baltimore, Maryland, January 12–14*, pp. 841–850. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

[7] Brisaboa, N. R., Fariña, A., Ladra, S., and Navarro, G. (2012) Implicit indexing of natural language text by reorganizing bytecodes. *Inf. Retr.*, **15**, 527–557.

[8] Brisaboa, N. R., Ladra, S., and Navarro, G. (2013) DACs: Bringing direct access to variable-length codes. *Inf. Process. Manage.*, **49**, 392–404.

[9] Külekci, M. O. (2014) Enhanced variable-length codes: Improved compression with efficient random access. *Proceeding of the Data Compression Conference, DCC 2014, Snowbird, UT, USA, 26–28 March*, pp. 362–371. IEEE Computer Society, Los Alamitos, CA.

[10] Klein, S. T. and Shapira, D. (2016) Compressed matching for feature vectors. *Theor. Comput. Sci.*, **638**, 52–62.

[11] Baruch, G., Klein, S. T., and Shapira, D. (2018) Accelerated partial decoding in wavelet trees. To appear in *Discrete Applied Mathematics*, **258**, https://doi.org/10.1016/j.dam.2018.07.016.

[12] Baruch, G., Klein, S. T., and Shapira, D. (2017) A space efficient direct access data structure. *J. Discrete Algorithms*, **43**, 26–37.

[13] Grossi, R. and Vitter, J. S. (2005) Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, **35**, 378–407.

[14] Manber, U. and Myers, G. (1993) Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, **22**, 935–948.

[15] Sadakane, K. (2003) New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, **48**, 294–313.

[16] Ferragina, P. and Manzini, G. (2005) Indexing compressed text. *J. ACM*, **52**, 552–581.

[17] Burrows, M. and Wheeler, D. J. (1994) A block sorting lossless data compression algorithm. *SRC Technical Report* **124**. Digital Equipment Corporation, Palo Alto, CA.

[18] Huo, H., Chen, L., Vitter, J. S., and Nekrich, Y. (2014) A practical implementation of compressed suffix arrays with applications to self-indexing. *Proceeding of the Data Compression Conference, DCC 2014, Snowbird, UT, USA, 26–28 March*, pp. 292–301. IEEE Computer Society, Los Alamitos, CA.

[19] Huo, H., Sun, Z., Li, S., Vitter, J. S., Wang, X., Yu, Q., and Huan, J. (2016) CS2A: A compressed suffix array-based method for short read alignment. *Proceeding of the Data Compression Conference, DCC 2016, Snowbird, UT, USA, March 30–April 1*, pp. 271–278. IEEE Computer Society, Los Alamitos, CA.

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

[21] Brisaboa, N. R., Iglesias, E. L., Navarro, G., and Paramá, J. R. (2003) An efficient compression code for text databases. *Advances in Information Retrieval, Proceedings of the 25th European Conference on IR Research, ECIR 2003, Pisa, Italy, April 14-16*, Lecture Notes in Computer Science, **2633**, pp. 468–481. Springer Verlag, Berlin.

[22] Brisaboa, N. R., Fariña, A., Navarro, G., and Esteller, M. F. (2003) (s, c)-dense coding: An optimized compression code for natural language text databases. *Proc. 10th Symposium on String Processing and Information Retrieval, SPIRE 2003, Manaus, Brazil, October 8-10*, Lecture Notes in Computer Science, **2857**, pp. 122–136. Springer Verlag, Berlin.

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

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

[25] Klein, S. T. and Ben-Nissan, M. K. (2010) On the usefulness of Fibonacci compression codes. *Comput. J.*, **53**, 701–716.

[26] Klein, S. T. and Shapira, D. (2006) Compressed pattern matching in JPEG images. *Int. J. Found. Comput. Sci.*, **17**, 1297–1306.

[27] Klein, S. T. and Shapira, D. (2019) Context sensitive rewriting codes for flash memory. *Comput. J.*, **62**, 20–29.

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

[29] Ottaviano, G. and Venturini, R. (2014) Partitioned Elias-Fano indexes. *The 37th Conference on Research and Development in Information Retrieval, SIGIR'14, Gold Coast, QLD, Australia, July 06 – 11*, pp. 273–282. ACM, NY, USA.

[30] Gog, S., Moffat, A., and Petri, M. (2017) CSA++: fast pattern search for large alphabets. *Proc. 19th Workshop on Algorithm Engineering and Experiments, ALENEX 2017, Barcelona, Spain, January 17-18*, pp. 73–82. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.