Fibonacci Based Compressed Suffix Array

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

[1] Dept. of Computer Science, Ariel University, Ariel 40700, Israel
benzakate@gmail.com, shapird@g.ariel.ac.il

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

Abstract. We propose Fibonacci based compressed suffix arrays, and show how repeated decompression can be avoided using our scheme. For a given file T of size n, the implementation requires $1.44\,n\,H_k + n + o(n)$ bits of space, where H_k is the k-th order empirical entropy of T, 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 previous suggestions.

1 Introduction

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. A *suffix array* (SA) for $T\$$, $\$ \notin \Sigma$, is an array $SA[1:n]$ of the indices of the 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 [17], 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. The *compressed suffix array* (CSA) introduced by Grossi and Vitter [9] is a text index that uses $2n\log\sigma$ bits in the worst case, and $O(m)$ processing time for searching a pattern of length m. Sadakane [19] extended the searching functionality to a self index, and proved that it uses search time $O(m\log n)$, and space $\epsilon^{-1}nH_0 + O(n\log\log\sigma)$ 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. [8] present an implementation of compressed suffix arrays that achieves asymptotic 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 [6] introduce the *FM-index*: a text index based on the Burrows-Wheeler Transform [4], which supports efficient pattern matching using a *Backwards Search*. The FM-index uses at most $5nH_k{+}o(n)$ bits for small alphabet size σ, and $O(m{+}\log^{1+\epsilon} n)$ searching time. We refer the reader to the book of Navarro [18] for a comprehensive review on compact data structures in general and compressed suffix arrays in particular.

Huo et al. [10] construct a space efficient CSA; Huo et al. [11] 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

Ekaterina Benza, Shmuel T. Klein, Dana Shapira: Fibonacci Based Compressed Suffix Array, pp. 3–11.
Proceedings of PSC 2018, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06484-9 © Czech Technical University in Prague, Czech Republic

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 [19] and the FM-index [6], 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 [10,11], and show how decompression can be avoided using our scheme. The implementation requires $1.44\, n\, H_k{+}n{+}o(n)$ bits of space, 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 [10,11].

The paper is organized as follows. Section 2 recalls the details of CSA and Section 3 presents its Fibonacci coding based variant, including the analysis of its space requirements and the summation process applied on the compressed form. Empirical results are given in Section 4.

2 Compressed Suffix Array

A suffix array (SA) for $T\$$, where T is a string over Σ and $\$ \notin \Sigma$, is an array $SA[1:n]$ of the indices of the suffixes of $T\$$, stored in lexicographical order. That is, if $SA[i]=j$ then the suffix starting at the j-th position of T, $T[j:n]$, is the i-th item in the lexicographically sorted list of all n 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 $\{1,\dots,n\}$, 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 different texts of length n over Σ. Thus a better lower bound is, in fact, $n \log \sigma$ bits. Grossi and Vitter [9] improve the space requirements of a suffix array by decomposing it based on the *neighbor function* defined as follows.

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

The inverse function $SA^{-1}[j]$ gives the position of $T[j:n]$ in the sorted list of the suffixes of T. The function Φ can also be rewritten as:

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

If $SA[i]$ refers to the suffix $T[j:n]$, then $\Phi[j]=i'$ is the position where $SA[i']=j+1$ refers to suffix $T[j+1:n]$. 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 $T[j:n] \prec T[j':n]$, 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 increasing arrays $\Phi_a = [1:n_a]$, for all $a \in \Sigma$, where n_a is the number of occurrences of the character a in the text. As an example, consider the text T = `mississippi$`. The text itself, the suffix array SA, its inverse function SA^{-1}, and Φ are given in the first rows in Figure 1. The last row partitions the Φ row into subintervals, denoted by Φ_i, Φ_m, Φ_p and Φ_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 giving the first character of the corresponding suffix, that is, holding $T[SA[i]]$ at position i.

	1	2	3	4	5	6	7	8	9	10	11	12
T	**m**	**i**	**s**	**s**	**i**	**s**	**s**	**i**	**p**	**p**	**i**	\$
SA	12	11	8	5	2	1	10	9	7	4	6	3
SA^{-1}	6	5	12	10	4	11	9	3	8	7	2	1
Φ	6	1	8	11	12	5	2	7	3	4	9	10
$T[SA]$	\$	i	i	i	i	m	p	p	s	s	s	s
Φ_a	$\Phi_\$$	Φ_i				Φ_m	Φ_p		Φ_s			

Figure 1. CSA example for T = `mississippi$`

The implementation for CSA used in [10,11] applies differential encoding. Instead of Φ itself, the values $\Delta\Phi[i] = \Phi[i] - \Phi[i-1]$ in each block are encoded, except for the first entry, which is assumed to be 0, thus need not be encoded. These differences are then encoded using *Elias'* methods [5]. Elias 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.

The Elias C_γ encoding of an integer x starts with a unary codeword of the number of bits in x followed by the standard binary codeword for x without its leading 1 bit. That is, $1 + \lfloor \log_2 x \rfloor$ is coded in unary, and $x - 2^{\lfloor \log_2 x \rfloor}$ is coded in binary for a total of $1 + 2\lfloor \log_2 x \rfloor$ bits. The Elias C_δ encoding uses the C_γ codeword for the number of bits in x, which requires $1 + 2\lfloor \log_2 \log_2 2x \rfloor$ bits, and again is followed by the binary codeword for x without the leading 1 bit, for a total of $1 + 2\lfloor \log_2 \log_2 2x \rfloor + \lfloor \log_2 x \rfloor$. A sample of Elias C_γ and C_δ codewords appears in Table 1 where blanks are inserted between the unary and the binary parts for clarity.

To provide faster access to the C_γ encoded sequence S of integers, which we denote as $C_\gamma(S)$, it is partitioned into so-called *super-blocks*, which in turn are subpartitioned into *blocks*, and three auxiliary tables SB, B and SAM are defined. For given values of 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 its corresponding super-block; and $\mathsf{SAM}[0 : \frac{n}{b} - 1]$ contains sampling values of Φ, so that the first value in each block is stored.

Each super-block refers to the encoding of $a = \log^3 n$ elements, and each block refers to the encoding of $b = \log^2 n$ 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. Figure 2 uses our running example illustrating the parsing of Φ into super-blocks of size $a = 8$ and into blocks of size $b = 4$. The differences are given in the row denoted by $\Delta\Phi$, and are encoded according to Elias'

C_γ. More precisely, the series $\Phi[ib+1]-\Phi[ib], \ldots, \Phi[(i+1)b-1]-\Phi[(i+1)b-2]$ is C_γ encoded, for all $0 \le i \le \frac{n}{b}$, and $\Phi[ib]=0$ is not encoded. In case the difference is negative, the value $\Phi[i]-\Phi[i-1]+n$ is used. The table also contains the encodings corresponding to the Fibonacci variants Fib_1 and Fib_2, presented in the next section, as well as the matching SB and B arrays.

<table>
<tr><th></th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th></tr>
<tr><td>SAM</td><td colspan="4">6</td><td colspan="4">12</td><td colspan="4">3</td></tr>
<tr><td>Φ</td><td>**6**</td><td>1</td><td>8</td><td>11</td><td>**12**</td><td>5</td><td>2</td><td>7</td><td>**3**</td><td>4</td><td>9</td><td>10</td></tr>
<tr><td>$\Delta\Phi$</td><td>0</td><td>8</td><td>7</td><td>3</td><td>0</td><td>6</td><td>10</td><td>5</td><td>0</td><td>1</td><td>5</td><td>1</td></tr>
<tr><td>$C_\gamma(\Delta\Phi)$</td><td colspan="4">0001000 00111 011</td><td colspan="4">00110 0001010 00101</td><td colspan="4">1 00101 1</td></tr>
<tr><td>SB</td><td colspan="8">0</td><td colspan="4">32</td></tr>
<tr><td>B</td><td colspan="4">0</td><td colspan="4">15</td><td colspan="4">0</td></tr>
<tr><td>$\mathsf{Fib}_1(\Delta\Phi)$</td><td colspan="4">000011 01011 0011</td><td colspan="4">10011 010011 00011</td><td colspan="4">11 00011 11</td></tr>
<tr><td>SB</td><td colspan="8">0</td><td colspan="4">31</td></tr>
<tr><td>B</td><td colspan="4">0</td><td colspan="4">15</td><td colspan="4">0</td></tr>
<tr><td>$\mathsf{Fib}_2(\Delta\Phi)$</td><td colspan="4">100101 101001 1001</td><td colspan="4">100001 1010001 10101</td><td colspan="4">1 10101 1</td></tr>
<tr><td>SB</td><td colspan="8">0</td><td colspan="4">34</td></tr>
<tr><td>B</td><td colspan="4">0</td><td colspan="4">16</td><td colspan="4">0</td></tr>
</table>

Figure 2. Super blocks and regular blocks parsing of Φ for $a=8$ and $b=4$ using C_γ, Fib_1 and Fib_2 codes.

The decoding function, denoted by $\mathcal{D}(\mathcal{E}, s, \ell)$, is given the encoded array $\mathcal{E}$ to be decoded, the starting position s within $\mathcal{E}$, and the number of codewords ℓ to be decoded. The values of Φ are then computed using:

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

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. As an example using Fib_2, $\Phi[11] = \mathsf{SAM}[11/4] + \mathcal{D}(\mathcal{E}, \mathsf{SB}[11/8] + \mathsf{B}[11/4], 11 \bmod 4) = \mathsf{SAM}[2] + \mathcal{D}(\mathcal{E}, \mathsf{SB}[1] + \mathsf{B}[2], 2) = 3 + \mathcal{D}(\mathcal{E}, 34 + 0, 2)$. Two consecutive values, 1 and 5, are decoded, and are added to 3, so that the final result 9 is returned.

3 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 [3] and (s,c)-dense codes [2]. This was also the motivation of using C_γ instead of the asymptotically better C_δ representation in the implementation of Huo et al. [10]. 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 the 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 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 [1], direct access [16], fast decoding and compressed search [13,15], compressed matching in dictionaries [14], faster modular exponentiation [12], 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 [7]. The adjacent 1s are then at the right instead of at the left end of each codeword, yielding the 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 [7], and found to be often preferable for the $\Delta\Phi$ encoding. 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 over the other codes is the encoding of the value 1 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 the most common value to be encoded.

3.1 Space Analysis

Recall that H_k denotes the k-th order empirical entropy. Huo et al. [10] 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 $k \leq c \log_a n - 1$ and any constant $c < 1$. Navarro [18] shows that if $\Delta\Phi$ is encoded using C_δ, the space for CSA is $nH_k + n + O(n)$.

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**	10 0000101
100	0000001 100100	00111 100100	**0010100001**	100100100001

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

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 to Fibonacci number F_r is $\frac{\phi^r}{\sqrt{5}}$, where $\phi = \frac{1+\sqrt{5}}{2} = 1.618$ is the golden ratio. From the fact that $F_{r-1} < x \le F_r$ we may extract that r is of the order of

$$\log_\phi x = (\log_\phi 2)\ \log_2 x = 1.4404 \log x.$$

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 [10] 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 $c < 1$.

3.2 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 $C[1:\ell]$. The sought result is obtained by summing $\sum_{j=1}^{\ell} C[j]F_j$ for Fib_1, or by summing $\sum_{j=1}^{\ell} C[j]F_j + i$ 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 $(i \mod b) \times \ell = 5 \times 4$ matrix:

$$\begin{pmatrix} 0\,1\,0\,0 \\ 0\,0\,1\,0 \\ 0\,0\,0\,1 \\ 1\,0\,0\,1 \\ 1\,0\,1\,0 \end{pmatrix}$$

4. Record the number of ones in each column in $C[1:4] = [2, 1, 2, 2]$.
5. $\sum_{j=1}^{\ell} C[j]F_j$ for i 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. Striping 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:

$$\begin{pmatrix} 1\,0\,0\,0 \\ 0\,1\,0\,0 \\ 1\,0\,1\,0 \\ 0\,0\,0\,1 \\ 0\,0\,1\,0 \end{pmatrix}$$

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

The processing time is thus proportional to the size of the compressed file, which is, asymptotically and empirically on our test files, smaller than the corresponding C_γ encodings used in [10], and does not require decoding tables.

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.

4 Experimental Results

We considered the same test files as [10], taken from the Pizza & Chili Corpus[1] as well as from the Canterbury Corpus[2]. We used the implementation of [10][3] and adapted it to encode the $\Delta\Phi$ values with Fib_1 and Fib_2, instead of Elias C_γ. We also report the space usage of Elias C_δ, as it is asymptotically the best of these four universal codes. The space usage for the super-blocks and blocks in our implementation is about the same as for the implementation of [10]. Tables 2 and 3 report the sizes in MBs of the encodings of $\Delta\Phi$ using these universal codes, Table 2 corresponding to files taken

[1] http://pizzachili. dcc.uchile.cl

[2] http://corpus.canterbury.ac.nz

[3] https://github.com/Hongweihuo-Lab/CSA

from the Canterbury Corpus, and Table 3 referring to files of size 100MB each taken from the Pizza & Chili corpus.

As can be seen, Fib_2 based CSA encoding performs the best on most files. Surprisingly, C_γ gives the best results for DNA and E.COLI, which are two files in the test files of [10] for which FM-index and Sadakane's CSA implementation produce better results than C_γ. Huo et al. explain this performance by the frequency of small values (1 and 2) in $\varDelta\varPhi$, which tends to be lower in these files than in the others. The other file for which Fib_2 does not produce the most efficient CSA is PROTEINS, for which it is outperformed by Fib_1.

Name	size (MB)	C_γ	C_δ	Fib_1	Fib_2
E.COLI	4.42	**1.923**	2.158	2.033	2.025
BIBLE	3.859	1.342	1.378	1.557	**1.320**
WORLD192	2.36	0.776	0.772	0.923	**0.747**
NEWS	0.36	0.178	0.175	0.183	**0.169**
BOOK1	0.73	0.348	0.358	0.361	**0.341**
PAPER1	0.05	0.024	0.024	0.025	**0.023**
KENNEDY	0.98	3.360	3.155	3.640	**3.049**

Table 2. Canterbury Corpus CSA using C_γ, C_δ, Fib_1 and Fib_2.

Name	C_γ	C_δ	Fib_1	Fib_2
DNA	**40.32**	44.99	43.79	42.24
DBLP.XML	22.90	23.15	32.12	**22.51**
SOURCES	31.92	31.69	38.48	**30.72**
ENGLISH	37.53	38.23	42.40	**36.79**
PROTEINS	65.51	64.89	**62.21**	62.72

Table 3. Pizza & Chili Corpus CSA using C_γ, C_δ, Fib_1 and Fib_2.

5 Conclusion

Huo et al. [10] 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.

However, the power of Fibonacci encoding has still not been fully exploited, especially the fact that adjacent 1's indicate the codewords' boundaries for both Fib_1 and Fib_2. For instance, this feature can replace the usage of the array B needed to indicate the beginning of each block of codewords relative to the start position of the corresponding super-block, yielding additional savings. This trade-off of time versus space will be addressed in future work.

Acknowledgement: We would like to thank Hongwei Huo for sharing the implementation of [10].

References

1. A. Apostolico and A. S. Fraenkel: *Robust transmission of unbounded strings using Fibonacci representations.* IEEE Trans. Information Theory, 33(2) 1987, pp. 238–245.
2. 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 String Processing and Information Retrieval, 10th International Symposium, SPIRE 2003, Manaus, Brazil, October 8-10, 2003, Proceedings, 2003, pp. 122–136.
3. N. R. Brisaboa, E. L. Iglesias, G. Navarro, and J. R. Paramá: *An efficient compression code for text databases*, in Advances in Information Retrieval, 25th European Conference on IR Research, ECIR 2003, Pisa, Italy, April 14-16, 2003, Proceedings, 2003, pp. 468–481.
4. M. Burrows and D. J. Wheeler: *A block sorting lossless data compression algorithm*, in Technical Report 124, Digital Equipment Corporation, 1994.
5. P. Elias: *Universal codeword sets and representations of the integers.* IEEE Trans. Information Theory, 21(2) 1975, pp. 194–203.
6. P. Ferragina and G. Manzini: *Indexing compressed text.* J. ACM, 52(4) 2005, pp. 552–581.
7. A. S. Fraenkel and S. T. Klein: *Robust universal complete codes for transmission and compression.* Discrete Applied Mathematics, 64(1) 1996, pp. 31–55.
8. R. Grossi, A. Gupta, and J. S. Vitter: *High-order entropy-compressed text indexes*, in Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '03, Philadelphia, PA, USA, 2003, Society for Industrial and Applied Mathematics, pp. 841–850.
9. R. Grossi and J. S. Vitter: *Compressed suffix arrays and suffix trees with applications to text indexing and string matching.* SIAM Journal on Computing, 35(2) 2005, pp. 378–407.
10. H. Huo, L. Chen, J. S. Vitter, and Y. Nekrich: *A practical implementation of compressed suffix arrays with applications to self-indexing*, in Data Compression Conference, DCC 2014, Snowbird, UT, USA, 26-28 March, 2014, 2014, pp. 292–301.
11. H. Huo, Z. Sun, S. Li, J. S. Vitter, X. Wang, Q. Yu, and J. Huan: *CS2A: A compressed suffix array-based method for short read alignment*, in 2016 Data Compression Conference, DCC 2016, Snowbird, UT, USA, March 30 - April 1, 2016, 2016, pp. 271–278.
12. S. T. Klein: *Should one always use repeated squaring for modular exponentiation?* Inf. Process. Letters, 106(6) 2008, pp. 232–237.
13. S. T. Klein and M. K. Ben-Nissan: *On the usefulness of Fibonacci compression codes.* Comput. J., 53(6) 2010, pp. 701–716.
14. S. T. Klein and D. Shapira: *Compressed pattern matching in JPEG images.* Int. J. Found. Comput. Sci., 17(6) 2006, pp. 1297–1306.
15. S. T. Klein and D. Shapira: *Compressed matching for feature vectors.* Theor. Comput. Sci., 638 2016, pp. 52–62.
16. S. T. Klein and D. Shapira: *Random access to Fibonacci encoded files.* Discrete Applied Mathematics, 212 2016, pp. 115–128.
17. U. Manber and G. Myers: *Suffix arrays: A new method for on-line string searches.* SIAM Journal on Computing, 22(5) 1993, pp. 935–948.
18. G. Navarro: *Compact Data Structures - A Practical Approach*, Cambridge University Press, 2016.
19. K. Sadakane: *New text indexing functionalities of the compressed suffix arrays.* J. Algorithms, 48(2) 2003, pp. 294–313.