A New Directly Accessible Compression Scheme*

Domenico Cantone, Simone Faro

Department of Mathematics and Computer Science
University of Catania, Viale A. Doria n.6, 95125, Catania, Italy

Abstract

We present a new variable-length computation-friendly encoding scheme, named SFDC (Succinct Format with Direct aCcessibility), that supports direct and fast accessibility to any element of the compressed sequence and achieves compression ratios often higher than those offered by other solutions existing in the literature. The SFDC scheme provides a flexible and simple representation geared towards either practical efficiency or compression ratios, as required. For a text of length n over an alphabet of size σ and a fixed parameter λ, the access time of our proposed encoding is proportional to the length of the character's code-word, plus an expected $\mathcal{O}((F_{\sigma-\lambda+3} - 3)/F_{\sigma+1})$ overhead, where F_j is the j-th Fibonacci number. In the overall, it uses $N + \mathcal{O}\big(n \cdot \big(\lambda - (F_{\sigma+3} - 3)/F_{\sigma+1}\big)\big) = N + \mathcal{O}(n)$ bits, where N is the length of the encoded string.

Keywords

Data compression, Huffman encoding, text representation, direct accessibility, data structure design

1. Introduction

The problem of *text compression* involves modifying the representation of any given text in plain format (referred to as *plain text*) so that the output format requires less space (as little as possible) for its storage and, consequently, less time for its transmission. It is understood that compression schemes must guarantee that the plain text can be reconstructed exactly from its compressed representation. Compressed representations allow one also to speed up algorithmic computations, as they can make better use of the memory hierarchy available in modern PCs, reducing disk access time. In addition, they find applications in the fields of compressed data structures [2, 3] that permit the manipulation of data directly in their encoded form, heavily used in bioinformatics and computer science. Thus, we contend that compression schemes with direct accessibility to the elements of the encoded sequence are of fundamental importance.

We tacitly assume that, in an uncompressed text y of length n over an alphabet Σ of size σ, every symbol is represented by $\lceil \log \sigma \rceil$ bits, for a total of $n\lceil \log \sigma \rceil$ bits.[1] On the other hand, using a more efficient variable-length encoding, such as the Huffman's optimal compression scheme [4], each character $c \in \Sigma$ can be represented with a code $\rho(c)$, whose (variable) length

ICTCS'24: Italian Conference on Theoretical Computer Science, September 11–13, 2024, Torino, Italy

* An extended version of this work is available in a public repository [1]. The extended version encompasses some initial experimental findings indicating that the performance of our approach is, to some extent, comparable to that of DACs and Wavelet Trees, which are considered among the most efficient schemes.

domenico.cantone@unict.it (D. Cantone); simone.faro@unict.it (S. Faro)
0000-0002-1825-0097 (D. Cantone); simone.faro@unict.it (S. Faro)

© 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

[1] Throughout the paper, all logarithms are intended in base 2, unless otherwise stated.

CEUR-WS.org/Vol-3811/paper100.pdf

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

Method	Reference	Overall Space	Access to $y[i]$
SS	-	$N + \lceil n/h \rceil \lceil \log(N) \rceil$	$\mathcal{O}(h\rho_{max})$
DS	[6]	$N + n(\log\log N + \log\log\sigma)$	$\mathcal{O}(\lvert\rho(y[i])\rvert)$
IC	[8, 9]	$N + \mathcal{O}(n\log(N)/\log(n))$	$\mathcal{O}(\log n)$
WT	[10]	$N + o(N)$	$\mathcal{O}(\lvert\rho(y[i])\rvert)$
DACs	[11, 12]	$\mathcal{O}((N\log\log N)/(\sqrt{N_0/n}\log N) + \log\sigma)$	$\mathcal{O}(N/(n(\sqrt{N_0/n})))$
SFDC	this paper	$N + \mathcal{O}(n)$	$\mathcal{O}(\lvert\rho(y[i])\rvert)$ (expected)

Table 1
Some of the main compression schemes based on variable-length codes that allow for direct access to characters in the encoded sequence

depends on the absolute frequency $f(c)$ of c in the text y, allowing the text to be represented with $N = \sum_{c\in\Sigma} f(c) \cdot |\rho(c)| \leqslant n\lceil\log\sigma\rceil$ bits.

Specifically, the Huffman algorithm computes an optimal *prefix code* relative to given frequencies of the alphabet characters, so that decoding becomes particularly simple as it can take advantage of ordered binary trees, whose leaves are labeled with the alphabet characters and whose edges are labeled with 0 (left edges) or 1 (right edges) in such a way that the code-word of an alphabet character is the word labeling the branch from the root to the leaf labeled by the same character. Prefix code trees, as computed by the Huffman algorithm, are called *Huffman trees*. These are not unique, by any means. The usually preferred tree for a given set of frequencies, out of the various possible Huffman trees, is the one induced by *canonical Huffman codes* [5]. Such trees have the property that, the sequence of the leaves depths, scanned from left to right, is non-decreasing.

One of the biggest problems with variable-length codes, like Huffman codes, is the impossibility of directly accessing the i-th code-word in the encoded string, for any $0 \leqslant i < n$, since its position in the encoded text depends on the sum of the lengths of the encodings of the characters that precede it.

This is why, over the years, various encoding schemes have been proposed that are able to complement variable-length codes with direct access to the characters of the encoded sequence. Dense Sampling [6], Elias-Fano codes [7], Interpolative coding [8, 9], Wavelet Trees [10], and DACs [11, 12] are just some of the best known and most efficient solutions to the direct access problem. However, the possibility of directly accessing the encoding of any character of the text comes at a cost, in terms of additional space used for the encoding, ranging from $\mathcal{O}(n\log N)$ for Dense Sampling to $\mathcal{O}(N)$ for the case of Wavelet Trees. Table 1 summarises the main direct access variable length codes, indicating the number of bits required for the encoding and the access time to any character of the text.

In this paper, we present a new variable-length encoding format for integer sequences that support direct and fast access to any element of the compressed sequence. The proposed format, named SFDC (Succinct Format with Direct aCcessibility), is based on variable-length codes obtained from existing compression methods. Just for presentation purposes, in this paper we

show how to construct our SFDC in the case of Huffman codes. Despite its apparent simplicity, which can be regarded as a value in itself, many interesting (and surprising) qualities emerge from our proposed encoding scheme. To the extent that we show in this paper, the SFDC encoding is relevant for the following reasons:

- it allows direct access to text characters in (expected) constant time, through a conceptually simple and easy to implement model;
- it achieves compression ratios that, under suitable conditions, are superior to those offered by other solutions in the literature;
- it offers a flexible representation that can be adapted to the type of application at hand, where one can prefer, according to her needs, aspects relating to efficiency versus those relating to space consumption;
- it is designed to naturally allow parallel accessing of multiple data, parallel-computation, and adaptive-access on textual data, allowing its direct application to various text processing tasks with the capability of improving performance up to a factor equal to the word length.

Unlike other direct-access coding schemes, SFDC is based on a model that offers a constant access time in the expected case, when character frequencies have a low variability along the text. We can prove that, in the overall, our compression scheme uses a number of bits equal to $N + \mathcal{O}\big(n\left(\lambda - (F_{\sigma+3} - 3)/F_{\sigma+1}\right)\big) = N + \mathcal{O}(n)$, where F_j is the j-th Fibonacci number and N is the length of the encoded string. In addition, our encoding allows accessing each character of the text in time proportional to the length of its encoding, plus an expected $\mathcal{O}((F_{\sigma-\lambda+3} - 3)/F_{\sigma+1})$ time overhead.[2]

2. A New Succinct Format with Direct Accessibility

In this section, we present in detail our proposed SFDC representation and the related encoding and decoding procedures. Again, let y be a text of length n, over an alphabet Σ of size σ, and let $f\colon \Sigma \to \{0, \dots, n\}$ be its corresponding frequency function. By running the Huffman algorithm on y, we generate a set of optimal prefix binary codes for the characters of Σ, represented by the map $\rho\colon \Sigma \to \{0,1\}^+$. For any $c \in \Sigma$, we denote by $|\rho(c)|$ the length of the binary code $\rho(c)$. Without loss of generality, we assume that the alphabet $\Sigma = \{c_0, c_1, \dots, c_{\sigma-1}\}$ is arranged as an ordered set in non-decreasing order of frequencies, so that $f(c_i) \leqslant f(c_{i+1})$ holds for $0 \leqslant i < \sigma - 1$. Thus, c_0 and $c_{\sigma-1}$ are the least and the most frequent characters in y, respectively. For instance, based on the codes shown in Fig. 2, the string `Compression`, of length 11, is encoded by the binary sequence

$$1100011010 \cdot 1100111 \cdot 101 \cdot 0110101 \cdot 11101 \cdot 01 \cdot 001 \cdot 001 \cdot 11010 \cdot 1100111 \cdot 010\,,$$

where the individual character codes have been separated by grey dots to enhance readability.

Let $\max(\rho) \coloneqq \max\{|\rho(c)| : c \in \Sigma\}$ be the length of the longest code of any character in Σ, and assume that λ is any fixed integer constant such that $2 \leqslant \lambda \leqslant \max(\rho)$. The SFDC codes

[2] From an experimental point of view, it is shown in [1] that in practical cases our scheme is particularly efficient and, in some respects, is comparable with the performance of DACs [11, 12] and Wavelet Trees [10], which are among the most efficient schemes in the literature.

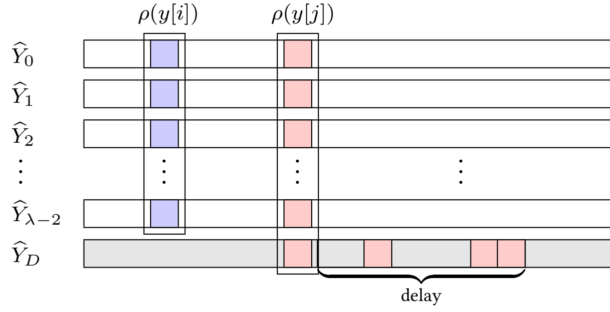

Figure 1: Examples of the reorganization of the bits of two character's code: $\rho(y[i])$ has length $\lambda - 1$ and fits within the λ layers of the representation; $\rho(y[j])$ has length $\lambda + 3$ and its 3 pending bits are arranged along the dynamic layer.

any string y of length n as an ordered collection of λ binary strings representing $\lambda - 1$ fixed layers and an additional dynamic layer. The number λ of layers is particularly relevant for the encoding performance. We will refer to it as the *size* of the SFDC representation.

The first $\lambda - 1$ binary strings, denoted $\widehat{Y}_0, \widehat{Y}_1, \ldots, \widehat{Y}_{\lambda-2}$, have length n. Specifically, the i-th binary string $\widehat{Y}_i$ is the sequence of the i-th bits (if present, 0 otherwise) of the encodings of the characters in y, in the order in which they appear in y. More formally, for $0 \leqslant i \leqslant \lambda - 2$, we have

$$\widehat{Y}_i := \big\langle \rho(y[0])[i],\ \rho(y[1])[i],\ \ldots,\ \rho(y[n-1])[i] \big\rangle,$$

where we agree that $\rho(y[j])[i] = 0$ when $i \geqslant |\rho(y[j])|$, for $0 \leqslant j < n$. Thus, each binary string $\widehat{Y}_i$ can be regarded as an array Y_i of $\lceil n/w \rceil$ binary blocks of size w. We refer to the bit vectors $\widehat{Y}_0, \widehat{Y}_1, \ldots, \widehat{Y}_{\lambda-2}$ as the *fixed layers* of the encoding, and to the bits stored in them as the *fixed bits*. The last layer of the encoding is a binary string $\widehat{Y}_D := \widehat{Y}_{\lambda-1}$, of length $n_D \geqslant n$, which gathers all the bits at positions $\lambda - 1, \lambda, \ldots$ of the encodings whose length exceeds $\lambda - 1$. We refer to such an additional layer as the *dynamic layer*, and to the bits in it as the *pending bits*.

Pending bits are arranged within the dynamic layer proceeding from left to right and following a last-in first-out (LIFO) scheme. Specifically, such a scheme obeys the following three rules: (a) If the encoding $\rho(y[i])$ has more than one pending bit, then each bit $\rho(y[i])[k+1]$ is stored on the dynamic layer $\widehat{Y}_D$ at some position on the right of that at which the bit $\rho(y[i])[k]$ is stored, for $k = \lambda - 1, \lambda, \ldots, |\rho(y[i])| - 1$. (b) For $0 \leqslant i < j < n$, each pending bit of $\rho(y[i])$ (if any) is stored in the dynamic layer either in a position p such that $i \leqslant p < j$ or to the right of all positions at which the pending bits (if any) of $\rho(y[j])$ have been stored. (c) Every pending bit is stored in the leftmost available position in the dynamic layer, complying with rules (a) and (b).

Fig. 2 shows the layout of the pending bits, arranged according to the LIFO scheme just illustrated. The fixed layers have a white background, while the dynamic layer has a grey one. In the middle, it is shown a relaxed 2-dimensional representation of the pending bits (i.e., the bits past position $\lambda - 2$) in the dynamic layer. On the right, the pending bits are arranged linearly along the dynamic layer, according to a last-in first-out approach. Idle bits are indicated with the symbol -. In the following sections, we will present the encoding and decoding procedures of the SFDC compression scheme.

char	code	length
s	001	3
e	01	2
n	010	3
p	0110101	7
m	101	3
C	1100011010	10
o	1100111	4
i	11010	5
r	11101	5

	0	1	2	3	4	5	6	7	8	9	10
	C	o	m	p	r	e	s	s	i	o	n
$\widehat{Y}_0$	1	1	1	0	1	0	0	0	1	1	0
$\widehat{Y}_1$	1	1	0	1	1	1	0	0	1	1	1
$\widehat{Y}_2$	0	0	1	1	1	-	1	1	0	0	0
$\widehat{Y}_3$	0	0	-	0	0	-	-	-	1	0	-
$\widehat{Y}_4$	0	1	-	1	1	-	-	-	0	1	-
$\widehat{Y}_D$	1	1		0						1	
	1	1		1						1	
	0										
	1										
	0										

	0	1	2	3	4	5	6	7	8	9	10
	C	o	m	p	r	e	s	s	i	o	n
$\widehat{Y}_0$	1	1	1	0	1	0	0	0	1	1	0
$\widehat{Y}_1$	1	1	0	1	1	1	0	0	1	1	1
$\widehat{Y}_2$	0	0	1	1	1	-	1	1	0	0	0
$\widehat{Y}_3$	0	0	-	0	0	-	-	-	1	0	-
$\widehat{Y}_4$	0	1	-	1	1	-	-	-	0	1	-
$\widehat{Y}_D$	1	1	1	0	1	1	0	1	0	1	1

Figure 2: The SFDC representation of the string `Compression` with $\lambda = 6$ layers (5 fixed layers and an additional dynamic layer)

2.1. The Encoding Procedure

The encoding procedure (see Figure 3, on the left) takes as input a text y of length n, the mapping function ρ generated by the Huffman compression algorithm, and the size λ of the SFDC representation. We recall that the mapping function ρ associates each character $c \in \Sigma$ with a variable-length binary code, and that the set of codes $\{\rho(c) \mid c \in \Sigma\}$ is prefix-free.

While iterating over all n characters of the text, the encoding procedure uses a stack $\mathcal{S}$ to implement the LIFO strategy for the arrangement of pending bits within the dynamic layer $\widehat{Y}_D$. At the i-th iteration, the algorithm takes care of inserting the first bits of the encoding $\rho(y[i])$ in the fixed layers, up to a maximum of $\lambda - 1$ bits. If $|\rho(y[i])| < \lambda - 1$, exactly $\lambda - |\rho(y[i])| - 1$ bits in the fixed layers will remain idle. Conversely, when the length of the encoding of the character $y[i]$ exceeds the value $\lambda - 1$, the remaining (pending) bits are pushed in reverse order into the stack $\mathcal{S}$. At the end of the i-th iteration, the first bit extracted from the stack is stored in the dynamic layer. Should the stack be empty at the i-th iteration, the i-th bit of the dynamic layer will remain idle.

At the end of the procedure, all the pending bits still in the stack, if any, are placed at the end of the dynamic layer.

Assuming that each of the layers of length n used in the representation can be initialised at 0^n in constant time (or at most in $\lceil n/w \rceil$ steps), the execution time of the encoding algorithm for a text y of length n is trivially equal to the sum of the encoding lengths of each individual character. Thus the encoding of a text of length n is performed in $\mathcal{O}(N)$-time, where N is the length of the encoded text. No additional time is required by the algorithm.

Concerning the number of bits used by the SFDC representation, it can be proved that our compression scheme uses, in the overall, a number of bits equal to $N + \mathcal{O}\big(n\left(\lambda - (F_{\sigma+3} - 3)/F_{\sigma+1}\right)\big) = N + \mathcal{O}(n)$, where F_j is the j-th Fibonacci number.

2.2. The Decoding Procedure

Although character decoding from a text with a SFDC representation takes place via a direct access to the position where the encoding begins, it does not necessarily take constant time. In fact, during the decoding of a character $y[i]$, it may be necessary to address some *additional work* related to a certain number of additional characters that have to be decoded in order to complete the full decoding of $y[i]$. In particular, if the last bit of the encoding of $y[i]$ is placed at position j in the dynamic layer, where $j \geqslant i$, then the procedure is forced to decode all the characters from position i to position j of the text, $y[j]$ included. We refer to such additional work, namely the $j - i$ additional characters that must be decoded in order to complete the decoding of character $y[i]$, as *decoding delay*. When no character other than $y[i]$ needs to be decoded, we have $j = i$ and the decoding delay is 0.

Despite the presence of decoding delays, when the whole text, or just a window of it, needs to be decoded, the additional work is not lost: it may indeed happen that a character at position j is decoded before a character at position i, with $i < j$. This happens when the pending bits of $\rho(y[i])$ are stored after position j.

We assume that the decoding procedure is invoked to decode the window $y[i\,..\,j]$ of the text, with $i \leqslant j$. The procedure (see Figure 3, on the right) takes as input the SFDC representation Y of the text y of length n, the starting position i and the ending position j of the window to be decoded, the root of the Huffman tree, and the size λ of the SFDC representation. Note that in order to decode the single character at position i, the procedure needs to be invoked with $j = i$. Likewise, to decode the entire text, the procedure must be invoked with $i = 0$ and $j = n - 1$.

Decoding of a character $y[i]$ is done by scanning a descending path on the Huffman tree, from the root to a leaf. Specifically, while scanning the bits of the encoding $\rho(y[i])$, we descend into the left or right subtree, according to whether we find a bit equal to 0 or to 1, respectively. We assume that each node x in the tree has a Boolean value associated with it, x.leaf, which is set to True when the node x is a leaf, False otherwise. We further assume that x.left (resp., x.right) allows one to access the left (resp., right) child of x. If the node x is a leaf, then it contains a parameter, x.symbol, that allows one to access the character associated with the encoding.

Again, we maintain a stack to extract the pending bits in the dynamic layer, storing the nodes in the Huffman tree associated with the characters in the text for which decoding has started but is not yet complete. Within the stack, the node x used for decoding the character $y[p]$ is coupled with the position p itself, so that the symbol can be positioned in constant time, once decoded. Thus, the elements of the stack are of the form $\langle x, p \rangle$. In our description we denote by x_p the node within the Huffman tree used for decoding the character $y[p]$. Since we use a LIFO strategy for bit arrangements in the dynamic layer, when two nodes x_i and x_j, with $1 \leqslant i < j \leqslant n$, are contained in the stack, the node x_j is closer than x_i to the top of the stack.

Each iteration of the procedure explores vertically the k-th bit of each layer, proceeding from the first layer $\widehat{Y}_0$ towards the last layer $\widehat{Y}_D$, in an attempt to decode the k-th character of the text, for $k \geqslant i$. The iterations stop when the entire window, $y[i\,..\,j]$, has been fully decoded. This condition occurs when the character $y[j]$ has been decoded (i.e., $y[j] \neq$Nil) and the stack is empty, so that all characters preceding position j in the window have already been decoded.

Each iteration is divided into two parts: the first part, which is executed only when $k < n$, explores the Huffman tree, starting from the root and proceeding towards the leaves, in the

```
readbit(B, i):
1.     return (B[i] ≫ i) & 1

writebit(B, i, b):
1.     B ← B | (b ≪ (w − i))

encode(y, n, ρ, λ):
1.     S ← new Stack
2.     i ← 0
3.     while i < n do
4.         h ← 0
5.         while (h < min(|ρ(y[i])|, λ − 1) do
6.             writebit(Y_h, i, ρ(y[i])[h])
7.             h ← h + 1
8.         for j ← |ρ(y[i])| − 1 downto h do
9.             S.push(ρ(y[i])[j])
10.        if (S ≠ ∅) then
11.            writebit(Y_D, i, S.pop())
12.        i ← i + 1
13.    while (S ≠ ∅) do
14.        writebit(Y_D, i, S.pop())
15.        i ← i + 1
16.    return Y
```

```
decode(Y, n, i, j, root, λ):
1.     S ← new Stack
2.     y[j] ← Nil
3.     k ← i
4.     while (y[j] ≠ Nil) do
5.         if (k < n) then
6.             x ← root
7.             h ← 0
8.             while (h < λ − 1 and not x.leaf) do
9.                 if (readbit(Y_h[k/w], k mod w))
10.                then x ← x.right
11.                else x ← x.left
12.                if (x.leaf) then y[p] ← x.symbol
13.                else S.push(⟨x, k⟩)
14.                h ← h + 1
15.        if (S ≠ ∅) then
16.            ⟨x, p⟩ ← S.pop()
17.            if (readbit(Y_D[k/w], k mod w)) then
18.                x ← x.right
19.            else x ← x.left
20.            if (x.leaf) then y[p] ← x.symbol
21.            else S.push(⟨x, k⟩)
22.        k ← k + 1
23.    return y[i..j]
```

Figure 3: Procedures encode and decode for encoding and decoding a text y of length n

hope of directly decoding the k-th character (this happens when $|\rho(y[k])| < \lambda$). If a leaf is reached at this stage, the text encoding is transcribed; otherwise, the node x_k is kept on hold and pushed onto the stack. The second part of the loop, if necessary (i.e., when $\mathcal{S} \neq \emptyset$), scans the bit in the dynamic layer Y_D and updates the node x_p at the top of the stack accordingly. If the node x_p reaches the position of a leaf, the corresponding symbol is transcribed to position $y[p]$, and the node is deleted from the stack.

The time complexity required to decode a single character $y[i]$ is $\mathcal{O}(|\rho(y[i])| + d)$, while decoding a text window $y[i..j]$ requires $\mathcal{O}(\sum_{k=i}^{j} |\rho(y[k]|) + d)$, where d is the decoding delay.

The decoding delay is a key feature for the use of SFDC in practical applications, since the expected access time to text characters is connected to it. This value depends on the distribution of the characters within the text and on the size λ of the representation. Ideally, the SFDC representation should use the minimum number of layers that ensures the expected value of the decoding delay to fall below a certain user-defined bound.

Since in general neither the frequency of the characters within the text nor their distribution is known in advance, it is not possible to define a priori the number of layers that is most suitable for the application at hand. However, one can compute the expected decoding delay value by simulating the construction of the SFDC representation on the text for a given number of layers. Should this value be above the bound, one can repeat the computation with a higher number of layers, until a value of λ is reached that guarantees an acceptable delay. Such a computation can be done by means of a procedure (see Figure 4) that simulates the construction

```
compute-delay(y, n, ρ, λ):
1.    S ← new Stack
2.    d ← i ← 0
3.    while i < n do
4.       h ← 0
5.       while (h < min(|ρ(y[i])|, λ − 1) do h ← h + 1
6.       if (h < |ρ(y[i])|) then S.push(⟨i, h⟩)
7.       if (S ≠ ∅) then
8.          ⟨p, h⟩ ← S.pop()
9.          if (h + 1 < |ρ(y[p])|) then S.push(⟨p, h + 1⟩)
10.         else d ← d + i − p
11.      i ← i + 1
12.   while (S ≠ ∅) do
13.      ⟨p, h⟩ ← S.pop()
14.      while (h < |ρ(y[i])|) do
15.         h ← h + 1
16.         i ← i + 1
17.      d ← d + i − p − 1
18.   return d/n
```

Figure 4: Procedure compute-delay for computing the average decoding delay of the SFDC of a text y of length n using λ layers. The procedure takes as input also the mapping ρ and the number λ of layers.

of the representation SFDC for the text y, without actually producing it.

Again, assuming that the λ layers of the SFDC can be initialised at 0^n in constant time, the complexity of such procedure is equal to $\mathcal{O}(N)$, where we recall that $N = \sum_{i=0}^{n-1} |\rho(y[i])|$.

Concerning complexity issues, it can be shown that, in the worst case, the summation $\sum_{i=0}^{\sigma-\lambda-1} f^r(c_i) \cdot (|\rho(c_i)| - \lambda)$, where $f^r(c)$ denotes the relative frequency of each character $c \in \Sigma$, provides an estimate of the expected delay of our SFDC encoding with λ layers (where $4 \leqslant \lambda \leqslant \sigma - 1$). Additionally, it can be proved that

$$\sum_{i=0}^{\sigma-\lambda-1} f^r(c_i) \cdot (|\rho(c_i)| - \lambda) = \frac{F_{\sigma-\lambda+3} - 3}{F_{\sigma+1}},$$

where F_j is the j-th Fibonacci number. As a consequence, the SFDC encoding allows accessing each character of the text in time proportional to the length of its encoding, plus an expected $\mathcal{O}((F_{\sigma-\lambda+3} - 3)/F_{\sigma+1})$ time overhead, which is less than 1.0 when $\sigma \geqslant 4$.

3. Complexity Issues

In this section, we present a theoretical analysis of the performance of our SFDC encoding, in terms of expected values of decoding delay and average number of idle bits. The first value is related to the average access time, while the second one gives us an estimate of the additional space used by the encoding with respect to the minimum number of bits required to compress the sequence. In our analysis, we make some simplifying assumptions that do not significantly impact the overall results. When necessary, in order to support the reasonableness of our

arguments, we will compare theoretical results with experimentally obtained data, reproducing the same conditions as in the theoretical analysis.

Before entering into the details of our analysis, we state two useful identities related to Fibonacci numbers, which hold for every $n \geqslant 0$, that can be proved by induction on n:

$$\sum_{i=0}^{n} F_i = F_{n+2} - 1\,, \tag{1}$$

$$\sum_{i=0}^{n} iF_i = nF_{n+2} - F_{n+3} + 2\,. \tag{2}$$

For convenience, we define a slight modification of the Fibonacci sequence, by putting

$$f_i := \begin{cases} 1 & \text{if } i = 0 \\ F_i & \text{otherwise.} \end{cases}$$

Hence, by (1), we have $\sum_{i=0}^{n} f_i = F_{n+2}$, for every $n \in \mathbb{N}$.

Let $\Sigma := \{c_0, c_1, \ldots, c_{\sigma-1}\}$ be an alphabet of size σ, and assume that the characters in Σ are ordered in non-decreasing order with respect to their frequency in a given text y (of length n), namely, $f(c_i) \leqslant f(c_{i+1})$ holds, for $0 \leqslant i < \sigma - 1$, where $f \colon \Sigma \to \mathbb{N}^+$ is the frequency function.

In order to keep the number of bits used by our encoding scheme as low as possible, it is useful to choose the number λ of layers as close as possible to the expected value

$$e_\rho := \frac{1}{n} \cdot \sum_{i=0}^{n-1} |\rho(y[i])|$$

of the length of the Huffman encodings of the characters in the text y. In particular, if we set $\lambda = \lceil e_\rho \rceil$, it is guaranteed that the number of idle bits, equal to $n(\lambda - e_\rho)$, is the minimum possible. As already observed, should the decoding delay prove to be too high in practical applications, the value of λ can be increased.

In terms of decoding delay, the optimal case for the application of the SFDC method is when all the characters of the alphabet have the same frequency. In this circumstance, in fact, the character Huffman encodings have length between $\lfloor \log \sigma \rfloor$ and $\lceil \log \sigma \rceil$.[3] Hence, in this case it suffices to use a number of layers equal to $\lambda = \lceil \log \sigma \rceil$ to obtain a guaranteed direct access to each character of the text, hence a decoding delay equal to 0.

On the other side of the spectrum, the worst case occurs when the Huffman tree, in its canonical configuration, is completely unbalanced. In this case, in fact, the difference between the expected value e_ρ and the maximum length of the Huffman encodings, namely $\big(\max_{c \in \Sigma} |\rho(c)|\big) - e_\rho$, is as large as possible, thus causing the maximum possible increase of the average decoding delay. In addition, we observe that the presence of several characters in the text whose encoding length is less than e_ρ does not affect positively the expected value of

[3]More precisely there are $2\sigma - 2^{\lfloor \log \sigma \rfloor + 1}$ characters with encodings of length $\lceil \log \sigma \rceil + 1$ and $2^{\lceil \log \sigma \rceil + 1} - \sigma$ characters with encodings of length $\lceil \log \sigma \rceil$.

the delay, since pending bits are placed exclusively in the dynamic layer. Rather, the presence of such *short-code* characters increases the number of idle bits of the encoding.

We call such a completely unbalanced tree a *degenerate tree.* Among all the possible texts whose character frequency induces a degenerate Huffman coding tree, the ones that represent the worst case for our encoding scheme are those in which the frequency differences $f(c_i) - f(c_{i-1})$, for $i = 1, 2, \ldots, \sigma - 1$ are minimal. It is not hard to verify that such a condition occurs when the frequencies follow a Fibonacci-like distribution of the following form

$$f(c_i) \coloneqq \begin{cases} 1 & \text{if } i = 0 \\ F_i & \text{if } 1 \leqslant i \leqslant \sigma - 1\,. \end{cases} \tag{3}$$

(or any multiple of it).

Thus, let us assume y to be a random text over Σ with the frequency function (3) (so that, by (1), $|y| = F_{\sigma+1}$), and let $f^r : \Sigma \to [0, 1]$ be the relative frequency function of the text y, where $f^r(c_i) = f(c_i)/F_{\sigma+1}$, for $i = 0, 1, \ldots, \sigma - 1$.

Assume also that $\rho\colon \Sigma \to \{0, 1\}^+$ is the canonical degenerate Huffman encoding of Σ relative to the frequency function (3), with

$$|\rho(c_i)| = \begin{cases} \sigma - 1 & \text{if } i = 0 \\ \sigma - i & \text{if } 1 \leqslant i \leqslant \sigma - 1. \end{cases}$$

Lemma 1. *The expected encoding length $E[|\rho(y[i])|]$ by ρ of the characters in y is equal to $\frac{F_{\sigma+3}-3}{F_{\sigma+1}}$.*

Proof 1. *Using (1) and (2), we have:*

$$\begin{aligned}
E[|\rho(y[i])|] &= \sum_{i=0}^{\sigma-1} f^r(c_i) \cdot |\rho(c_i)| = \sum_{i=0}^{\sigma-1} \frac{f(c_i)}{F_{\sigma+1}} \cdot |\rho(c_i)| \\
&= \frac{1}{F_{\sigma+1}} \cdot \left(\sum_{i=1}^{\sigma-1} (\sigma - i) F_i + \sigma - 1 \right) = \frac{1}{F_{\sigma+1}} \cdot \left(\sigma \cdot \sum_{i=0}^{\sigma-1} f_i - \sum_{i=1}^{\sigma-1} i \cdot F_i - 1 \right) \\
&= \frac{1}{F_{\sigma+1}} \cdot (\sigma F_{\sigma+1} - (\sigma - 1) F_{\sigma+1} + F_{\sigma+2} - 3) \\
&= \frac{1}{F_{\sigma+1}} \cdot (F_{\sigma+1} + F_{\sigma+2} - 3) = \frac{F_{\sigma+3} - 3}{F_{\sigma+1}}.
\end{aligned}$$

As a consequence of the above, the expected number of idle bits per character in a SFDC encoding using λ layers can be estimated by the following expression $\lambda - (F_{\sigma+3} - 3)/F_{\sigma+1}$.

Table 2 shows the expected number of idle bits for each text element, in a SFDC encoding of a text over an alphabet with Fibonacci frequencies, comparing theoretical values against values experimentally computed. To obtain the experimental values, we used artificially generated texts of 100 MB whose character frequency results in a Fibonacci frequency.

Idle Bits (Theoretical)			
$\lambda \setminus \sigma$	10	20	30
5	2.42	2.38	2.38
6	3.42	3.38	3.38
7	4.42	4.38	4.38
8	5.42	5.38	5.38

Idle Bits (Experimental)			
$\lambda \setminus \sigma$	10	20	30
5	2.29	2.26	2.27
6	3.29	3.27	3.27
7	4.30	4.29	4.29
8	5.30	5.31	5.31

Table 2
Expected number of idle bits for each text element, in a SFDC encoding of a text over an alphabet with Fibonacci frequencies. Values are shown for $5 \leqslant \lambda \leqslant 8$ and $\sigma \in \{10, 20, 30\}$.

As the size of the alphabet σ increases, it is easy to verify that the function quickly converges to the value $\lambda - 2.618$. Indeed, by recalling that $\lim_{\sigma\to\infty} F_\sigma/\phi^\sigma = 1$, where $\phi = \frac{1}{2}(1+\sqrt{5})$ is the golden ratio, by Lemma 1 we have

$$\lim_{\sigma\to\infty} E[|\rho(y[i])|] = \lim_{\sigma\to\infty} \frac{F_{\sigma+3}-3}{F_{\sigma+1}} = \phi^2 = \frac{1}{2}(3+\sqrt{5}) \approx 2.618\,.$$

Consequently, if we assume that the value λ represents a constant implementation-related parameter, the total space used by SFDC for encoding a sequence of n characters is equal to $N + \mathcal{O}\left(n\left(\lambda - \frac{F_{\sigma+3}-3}{F_{\sigma+1}}\right)\right) = N + \mathcal{O}(n)$, where $N \coloneqq \sum_{i=0}^{\sigma-1} f(c_i)\cdot|\rho(c_i)|$.

3.1. Expected Decoding Delay

On the basis of what was shown above, we now estimate the expected value of the decoding delay, i.e., the number of additional characters that need to be decoded in order to obtain the full encoding of a character of the text.

Assume that the i-th character of the text is $y[i] = c_k$, where $0 \leqslant k < \sigma$ and $0 \leqslant i < n$. We want to estimate the position $j \geqslant i$ at which the last pending bit of the encoding $\rho(y[i])$ is placed, so that the delay of character $y[i]$ is $j - i$.

Let us assume that the number of layers of the representation is equal to λ. If the length of the encoding of $y[i]$ falls within the number λ of layers, namely $|\rho(y[i])| \leqslant \lambda$, then the character can be decoded in a single cycle and the delay is 0. On the other hand, if $|\rho(y[i])| > \lambda$, then $|\rho(y[i])| - \lambda$ bits remain to be read, all arranged within the dynamic layer. Since the length of the encoding of any text character can be approximated in the average case to the value 2.618, if we assume to use a fixed number of layers $\lambda \geqslant 4$, we expect that on average the pending bits of $\rho(y[i])$ will be placed in the $|\rho(y[i])| - \lambda$ positions past position i. The average delay in this case can be estimated by $|\rho(y[i])| - \lambda$.

Thus, for an alphabet of σ characters with frequency function (3), the expected delay of our Fibonacci encoding with λ layers (where $4 \leqslant \lambda \leqslant \sigma - 1$) can be estimated by the summation: $\sum_{i=0}^{\sigma-\lambda-1} f^r(c_i)\cdot(|\rho(c_i)| - \lambda)$.

We claim that

$$\sum_{i=0}^{\sigma-\lambda-1} f^r(c_i)\cdot(|\rho(c_i)| - \lambda) = \frac{F_{\sigma-\lambda+3}-3}{F_{\sigma+1}}. \tag{4}$$

AVERAGE DELAY (THEORETICAL)					
$\lambda \setminus \sigma$	10	15	20	25	30
5	0.20	0.23	0.24	0.24	0.24
6	0.11	0.14	0.15	0.15	0.15
7	0.06	0.09	0.09	0.09	0.09
8	0.02	0.05	0.06	0.06	0.06

AVERAGE DELAY (EXPERIMENTAL)					
$\lambda \setminus \sigma$	10	15	20	25	30
5	0.31	0.37	0.37	0.39	0.39
6	0.15	0.20	0.18	0.17	0.21
7	0.07	0.11	0.11	0.11	0.11
8	0.02	0.06	0.06	0.07	0.06

Table 3
The average delay in decoding a single character in a text over an alphabet with Fibonacci frequencies using SFDC. Theoretical values (on the left) compared with experimental values (on the right). Values are tabulated for $5 \leqslant \lambda \leqslant 8$ and $\sigma \in \{10, 15, 20, 25, 30\}$.

Indeed,

$$\begin{aligned}
\sum_{i=0}^{\sigma-\lambda-1} f^r(c_i) \cdot (|\rho(c_i)| - \lambda) &= \frac{1}{F_{\sigma+1}} \cdot \sum_{i=0}^{\sigma-\lambda-1} f_i \cdot (|\rho(c_i)| - \lambda) \\
&= \frac{1}{F_{\sigma+1}} \cdot \left(\sum_{i=1}^{\sigma-\lambda-1} F_i \cdot (\sigma - \lambda - i) + \sigma - \lambda - 1 \right) \\
&= \frac{1}{F_{\sigma+1}} \cdot \left((\sigma - \lambda) \cdot \sum_{i=1}^{\sigma-\lambda-1} F_i - \sum_{i=1}^{\sigma-\lambda-1} iF_i + \sigma - \lambda - 1 \right) \\
&= \frac{(\sigma-\lambda)(F_{\sigma-\lambda+1} - 1) - (\sigma - \lambda - 1)F_{\sigma-\lambda+1} + F_{\sigma-\lambda+2} - 2 + \sigma - \lambda - 1}{F_{\sigma+1}} \\
&= \frac{F_{\sigma-\lambda+1} + F_{\sigma-\lambda+2} - 3}{F_{\sigma+1}} = \frac{F_{\sigma-\lambda+3} - 3}{F_{\sigma+1}}.
\end{aligned}$$

Table 3 reports the delay in decoding a single character in a text over an alphabet with Fibonacci frequencies using SFDC, comparing values resulting from the identity (4) against values computed experimentally. To obtain the experimental values, we generated texts of 100 MB whose character frequency is the Fibonacci frequency (3). For each such text y, the average delay value was experimentally obtained by accessing all the characters in random order, computing the delays, and dividing the sum by the total number of characters in y.

4. Conclusions and future works

In this paper, we have introduced a data reorganisation technique that, when applied to variable-length codes, allows easy, direct, and fast access to any character of the encoded text, in time proportional to the length of its code-word, plus an additional overhead that is constant in many practical cases. Besides being extremely simple to be translated into a computer program and efficient in terms of space and time, our method has the surprising feature of being also computation-friendly. We expect, in fact, that it will turn out to be particularly suitable for applications in text processing. Our future work will be devoted to demonstrating such a claim.

References

[1] D. Cantone, S. Faro, The many qualities of a new directly accessible compression scheme, CoRR abs/2303.18063 (2023). URL: https://doi.org/10.48550/arXiv.2303.18063. doi:10.48550/ARXIV.2303.18063. arXiv:2303.18063.

[2] A. Moffat, A. Turpin, Compression and Coding Algorithms, volume 669 of *The international series in engineering and computer science*, Kluwer, 2002. URL: http://www.cs.mu.oz.au/caca/.

[3] D. Salomon, Variable-length Codes for Data Compression, Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 2007.

[4] D. A. Huffman, A method for the construction of minimum-redundancy codes, Proceedings of the Institute of Radio Engineers 40 (1952) 1098–1101.

[5] E. S. Schwartz, B. Kallick, Generating a canonical prefix encoding, Commun. ACM 7 (1964) 166–169. URL: https://doi.org/10.1145/363958.363991. doi:10.1145/363958.363991.

[6] P. Ferragina, R. Venturini, A simple storage scheme for strings achieving entropy bounds, Theor. Comput. Sci. 372 (2007) 115–121. URL: https://doi.org/10.1016/j.tcs.2006.12.012. doi:10.1016/j.tcs.2006.12.012.

[7] P. Elias, Efficient storage and retrieval by content and address of static files, J. ACM 21 (1974) 246–260. URL: https://doi.org/10.1145/321812.321820. doi:10.1145/321812.321820.

[8] A. Moffat, L. Stuiver, Binary interpolative coding for effective index compression, Inf. Retr. 3 (2000) 25–47. URL: https://doi.org/10.1023/A:1013002601898. doi:10.1023/A:1013002601898.

[9] J. Teuhola, Interpolative coding of integer sequences supporting log-time random access, Inf. Process. Manag. 47 (2011) 742–761. URL: https://doi.org/10.1016/j.ipm.2010.11.006. doi:10.1016/j.ipm.2010.11.006.

[10] R. Grossi, A. Gupta, J. S. Vitter, High-order entropy-compressed text indexes, in: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, ACM/SIAM, 2003, pp. 841–850. URL: http://dl.acm.org/citation.cfm?id=644108.644250.

[11] N. Brisaboa, S. Ladra, G. Navarro, Directly addressable variable-length codes, in: SPIRE 2009, volume 5721 of *LNCS*, Springer, 2009, pp. 122–130. URL: https://doi.org/10.1007/978-3-642-03784-9_12. doi:10.1007/978-3-642-03784-9_12.

[12] N. R. Brisaboa, S. Ladra, G. Navarro, Dacs: Bringing direct access to variable-length codes, Inf. Process. Manag. 49 (2013) 392–404. URL: https://doi.org/10.1016/j.ipm.2012.08.003. doi:10.1016/j.ipm.2012.08.003.