Bit-Layers Text Representation for Efficient Text Processing* †

Domenico Cantone, Simone Faro, and Stefano Scafiti

Università di Catania, Viale A.Doria n.6, 95125 Catania, Italy
domenico.cantone@unict.it, faro@dmi.unict.it,
stefano.scafiti@studium.unict.it

Abstract. Textual data still remains the main format for storing information, justifying why text processing is among the most relevant topics in computer science. However, despite capability to store information is growing fast, the amount and complexity of textual data grows faster than storage capacities. In many cases, the problem is not due to the size or complexity of the text, but rather to the representations (or data structure) employed for carrying out the needed processing.
In this paper, we show the potentiality and the benefits of a straightforward text representation, referred to as the *Bit-Layers* text representation, which turns out to be particularly suitable for fast text searching, while still retaining the standard efficiency in the rest of text processing basic tasks. To show the advantages of the Bit-Layers representation, we also present a family of simple algorithms, tuned to it, for solving some classical and non-classical string-matching problems. Such algorithms turn out to be particularly suitable for implementation in modern hardware, and very fast in practice. Preliminary experimental results show that in some cases these algorithms are by far faster than their counterparts based on the standard text representation.

Keywords: Text processing · representation · experimental algorithms

1 Introduction

Text processing is one of the most relevant topics in computer science. It includes, among other problems, exact and approximate string matching, which are still among the most fundamental problems in computer science. Textual data remains indeed the main form for storing information, even though data are memorized in different ways. Thus, the need for faster and faster solutions to text processing problems. However, it turns out that the amount of available textual data grows faster than storage capacities and, as a consequence, the

* We gratefully acknowledge support from "Università degli Studi di Catania, Piano della Ricerca 2016/2018 Linea di intervento 2".

capability to perform text processing in the main memory of present-day computers is becoming more and more arduous. On the other hand, since processing texts in main memory is by far faster than on disks,[1] operating in main memory is crucial for carrying out efficient text processing. In many cases, the problem is not the size of the texts stored, but rather the data structures that must be built on the texts in order to efficiently carry out the processing [16].

The representation of a DNA sequence is representative of this problem: to encode a human genome, which is about 3.3 billion bases, slightly less than 800 MB are required, if one uses only 2 bits per character. On the other hand, standard text representation uses 8 bits per character, requiring a total of more than 2 GB, whereas the corresponding suffix tree requires at least 10 bytes per base, that is more than 30 GB, too large for many practical applications.

Several works appeared in recent years aiming at finding a tradeoff between the space requirements needed to encode huge texts and complex data structures and the time efficiency of the text searching algorithms designed to work with such representations. Among the most relevant approaches in this direction, we mention *compressed text searching*, in which processing takes place online directly on compressed data in order to speed-up searching with reduced extra space. This problem has been widely investigated in the last few years and, although efficient solutions exist for searching under standard compressions schemes,[2] they still require significant implementation efforts and turn out to be not very flexible, being designed for specific tasks.

A more suitable solution is *compact text processing*, which aims at storing information using *succint text representations*, allowing, at the same time, fast in-place text processing with no decompression.

The concept of succinct data representation and succinct data structures was originally introduced by Jacobson [15] to encode bit vectors, trees, and graphs, and has been recently brought back to the top [16] for the reasons discussed above.

Improving results in this direction, we discuss in this paper the advantages of using the *Bit-Layers* text representation, a succinct and quite basic string representation, which turns out to be particularly suitable for fast text searching, while retaining standard efficiency in the rest of basic text processing applications.

We also present a family of simple algorithms for solving some classical and non-classical string-matching problems, tuned to our proposed text representation, which turn out to be particularly suitable for implementation in modern hardware and very fast in practice. Preliminary experimental results show that in some cases these algorithms are by far faster than their counterparts based on the standard text representation.

[1] In present-day computers, accessing a text in main memory is about 10^5 times faster than on secondary memory.

[2] Best solutions for compressed string matching use less than 70% extra space for the text and are twice as fast in searching than standard online string matching algorithms.

2 Standard and Succinct Text Representations

In this section we briefly review the most relevant text representations that are generally used to code a string y in main memories with a word size of w bits. We shall assume that the block size w is fixed, so that all references to a string will only be to entire blocks of w bits. For simplicity, we refer to a w-bit block as a *byte*, though larger values than $w = 8$ could be supported as well.

Let y be a string of length $n \geqslant 0$ of characters from a finite alphabet Σ of size σ, such that $\ell \coloneqq \lceil \log(\sigma) \rceil \leqslant w$. In standard text representation, the string y is coded as an array S_y of n blocks, each of size w, that can be read and written at arbitrary positions, and where the i-th block of S_y contains the binary representation of the i-th character of y. We denote by $\rho(c)$ the binary representation of $c \in \Sigma$, so that $\rho(c)[j]$, with $0 \leqslant j < \ell$, is the j-th most significant bit in $\rho(c)$. As long as $\ell \leqslant w$, each character can be read and processed in a single operation, otherwise $\lceil \log(\sigma)/w \rceil$ operations are required.

Any array A of n blocks of size w can be regarded as a virtual bit array $\hat{A}$ of nw bits, where each bit can be processed at the cost of a single operation. Conversely, any bit string $\hat{B}$ of length m could be seen as an array B of $\lceil m/w \rceil$ blocks. Thus we have that $\hat{B}[i]$ is the j-th bit of $B[\lfloor i/w \rfloor]$, where $j \coloneqq i \mod w$. Since the length m of a binary string needs not necessarily be a multiple of w, the last block may be only partially defined. This approach turns out to be very simple and fast, as read and write operations of a given character can be done in constant time, by using a direct access to the position of the array where the character is stored or needs to be written. However, such a simple representation may waste a lot of space, since wn bits are used, where just ℓn would suffice.

A succinct representation of the string y has been discussed in [16]. It allocates an array C of $\lceil \ell n/w \rceil$ blocks, which is enough to encode n elements of ℓ bits. We regard these n bits stored in C as a virtual bit array $\hat{C}$ of ℓn bits, where each character $y[i]$ is stored at $\hat{C}[i\ell \mathinner{..} (i+1)\ell - 1]$. Also in this case any character $y[i]$ can be accessed in constant time, although it may require to access more then one word for a single access. Solutions tuned to this succinct text representation exists [17, 8], aiming at speeding up text searching, however it turns out that the gain is quite poor in practice, and is obtained only when ℓ divides w exactly.

For completeness, we mention other succinct text representations, based on variable-length encodings, which are related to our approach since they provide the enviable feature to give direct access to the text. We mention those based on sampling [11], the Elias-Fano-based representation [6, 7], Interpolative coding [18], Wavelet tree [12] and Direct Addressable Codes (DACs) [3]. Such text representations are specifically designed for succinctness and are not competitive for text processing tasks, if compared against standard text representations.

Notably, our proposed bit-layers representation is strongly related with Direct Addressable Codes (DACs) where, as in our approach, the bits of each encoding are stored in different bit sequences. However, DACs use variable-length encodings (where our representation encodes all characters of the alphabet with the

same number of bits, thus enhancing text processing performances) and maintains additional informations about the structures of the layers.

3 Bit-Layers Text Representation

Assume again that y is a string of length n over an alphabet Σ of size σ. As in the case of the succinct text representation described above, let us suppose that each character in Σ is represented by $\ell \coloneqq \lceil \log(\sigma) \rceil$ bits.

The bit-layers text representation codes the string y as an ordered collection of ℓ binary strings of length n, $\langle \hat{B}_0, \hat{B}_1, \ldots, \hat{B}_{\ell-1} \rangle$ (the reference to y has been omitted for conciseness), where the i-th binary string $\hat{B}_i$ is the sequence of the i-th bits of the characters in y, in the order in which they appear in y. We refer to the bit vectors $\hat{B}_0, \hat{B}_1, \ldots, \hat{B}_{\ell-1}$ induced by such representation as the *bit layers* of the encoding. More formally, letting $c \mapsto \rho(c)$ be the encoding map, for $c \in \Sigma$, then

$$\hat{B}_i \coloneqq \langle \rho(y[0])[i], \rho(y[1])[i], \ldots, \rho(y[n-1])[i] \rangle,$$

for $i = 0, \ldots, \ell - 1$. Thus, each layer $\hat{B}_i$ can be regarded as an array B_i of $\lceil n/w \rceil$ blocks of size w.

Example 1. Let $y = \texttt{abfefdgabaadefcc}$ be a string of 16 characters over the alphabet $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{e}, \texttt{f}, \texttt{g}\}$, with $\sigma = 7$, so that each character can be represented by $\lceil \log(\sigma) \rceil = 3$ bits. Assume therefore that $\rho(\texttt{a}) = 000$, $\rho(\texttt{b}) = 001$, $\rho(\texttt{c}) = 010$, $\rho(\texttt{d}) = 011$, $\rho(\texttt{e}) = 100$, $\rho(\texttt{f}) = 101$ and $\rho(\texttt{g}) = 110$.

According to our bit-layers text representation, the string y can be represented by the following sequence of 3 binary vectors, each one stored on $\lceil n/w \rceil = 2$ bytes:

$$\begin{array}{l|l} B_0 = 108.148 & \hat{B}_0 = 01101100.10010100 \\ B_1 = 006.019 & \hat{B}_1 = 00000110.00010011 \\ B_2 = 058.012 & \hat{B}_2 = 00111010.00001100 \end{array}$$

Since the bit $\hat{B}_i[j]$ is stored at the r-th most significant bit of $B_i[\lceil j/w \rceil]$, where $r = ((j-1) \bmod w) + 1$, the character $y[i]$ can readily be retrieved by the following formula: $y[i] = \langle \hat{B}_\ell[i], \hat{B}_{\ell-1}[i], .., \hat{B}_0[i] \rangle$.

Procedure `read` in Figure 1 accesses character $y[i]$ using the bit-layers text representation for coding y in $\Theta(\ell)$ time, which is worse than the constant time needed for reading a character under the standard or the succinct representations. However, our experimental results show that, in practical cases, the access speed to text's characters using the bit-layers text representations is as fast as using other representations (see Section 4).

The bit-layers representation arranges textual data in a two-dimensional structure so that text processing may proceed both *horizontally*, by reading bits (or bit blocks) along a binary vector in a given layer, and *vertically*, by moving from a given layer to the next one while reconstructing the encoding of a character (or of a group of characters).

```
readbit(B, i): returns (B[i] ≫ i) & 1
read(i):
    c ← 0
    for h ← 0 to ℓ − 1 do
        c ← c | readbit(B_h[i/w], i mod w) ≪ h
    return c
```

Fig. 1. Procedure read for retrieving the i-th character of a text coded using the bit-layers text representation. We assume that each character of the alphabet can be represented by $\ell = \lceil \log(\sigma) \rceil \leqslant w$ bits.

Thanks to its two-dimensional structure, the bit-layers text representation counts many favourable and interesting features.

First of all, it naturally allows *parallel computation* on textual data. Since a string is partitioned in ℓ independent binary vectors, it is straightforward to entrust the processing of each bit vector to a different processor, provided that the corresponding ℓ outputs are then blended.

The bit-layers representation is also well suited for *parallel accessing of multiple data*. A single computer word maintains, indeed, partial information about w contiguous characters. Although we can access only a fraction (namely $(1/\ell)$th) of a character encoding, such information can be processed in constant time, also by exploiting the intrinsic parallelism of bitwise operations, which allows one to cut down the horizontal processing by a factor up to w.

In addition, it allows also *adaptive accesses* to textual data. This means that processing may not always need to go all the way in depth along the layers of the representation, as in favourable cases accesses can stop already at the initial layers of the representation of a given character. For instance, assume we are interested in searching all occurrences of the character a in a given text y, where $\rho(\mathtt{a}) = 000$. If, while inspecting character $y[i]$ it is discovered that $\hat{B}_0[i] = 1$, then such position can immediately be marked as a mismatch. This feature may allow, under suitable conditions, to cut down the vertical data processing by a factor up to ℓ.

Finally, the bit-layers representation turns out to be also well suited for *cache-friendly accesses* to textual data. Indeed, the cache-hit rate is very crucial for good performances, as each cache miss results in fetching data from primary memory (or worse form secondary memory), which takes considerable time.[3] In comparison, reading data from the cache typically takes only a handful of cycles. According to the principle of spatial locality, if a particular vector location is referenced at a particular time, then it is expected that nearby positions will be referenced in the near future. Thus, in present-day computers, it is common to attempt to guess the size of the area around the current position for which it is worthwhile to prepare for faster accesses for subsequent references. Assume, for

[3] Data fetching takes hundreds of cycles from the primary memory and tens of billions of cycles from secondary memory.

instance, that such a size is of k positions, so that, after a cache miss, when a certain block of a given array is accessed, the next k blocks are also stored in cache memory. Then, under the standard text representation, we have at most k supplementary text accesses before the next cache miss, whereas in our bit-layers representation such number may grow by a factor up to ℓ, resulting, under suitable conditions, in faster accesses to text characters.

In the following section, we shall present some preliminary evidences that the bit-layers text representation is particularly appropriate for fast text processing. Specifically, we shall provide basic solutions to some standard and non-standard search problems, comparing the results obtained against those achieved under the standard representation.

4 Text Searching Using Bit-Layers Representation

Text processing, and especially text searching, can be included among the most significant topics in computer science. Many times, over the years, standard approaches to text searching have been influenced, or even transformed, by particularly innovative studies. Among them, we mention the Boyer-Moore algorithm [4, 13], the first practical approach to text searching, and the Bit-Parallelism [2], which respectively mainly focused on the search strategy and on the method to build or represent effective data structures to speed up the search process, even by carrying it out in parallel.

The bit-layers text representation presented in this paper aims at improving text searching by moving enhancements at the very bottom of the problem, namely at the encoding level, allowing a natural and more significant parallel computation, particularly suitable for hardware implementation. For these reasons, we believe that such representation is a valuable contribution to any application dealing with text searching.

Although the representation can inspire several technical and creative approaches to text searching, in this first paper we just show a few straightforward solutions tuned to the proposed representation. Specifically, we present some solutions based on the following two approaches: *horizontal word parallelism* and *vertical adaptive parallelism*.

In the *horizontal word parallelism*, textual data are read in chunks of w bits, from all the layers of the representation, and partial information is processed in parallel. In the *vertical adaptive parallelism* approach, we proceed much in the same way; however one does not always need to go all the way in depth along the layers of the representation.

In the following sections, we present solutions to some basic text processing tasks and we report the related experimental data that allow us to compare our proposed algorithms against those tuned for standard text representation.

In our experiments, all algorithms have been implemented in the C programming language, using 32 bits words (i.e. $w = 32$), and have been tested with the SMART research tool [10], which is available online at the URL `http:`

`//www.dmi.unict.it/~faro/smart/`.[4] All experiments have been executed locally on a MacBook Pro with 4 Cores, a 2 GHz Intel Core i7 processor, 16 GB RAM 1600 MHz DDR3, 256 KB of L2 Cache, and 6 MB of Cache L3. In addition, all algorithms have been compared in terms of their running times, excluding preprocessing times, on the following real data sets: a genome sequence, a protein sequence, and an English text. Each of the three sequences has a length of 15MB, and can be downloaded from the SMART tool. In the experimental evaluation, patterns of length m have been randomly extracted from the sequences, with m ranging over the set of values $\{2^i \mid 2 \leqslant i \leqslant 5\}$. For each experiment, the mean over the processing speed (expressed in billions of characters per second) of 500 runs has been reported.

4.1 Simple Text Scanning

Simple *text scanning* is a basic task in text processing. It consists in reading the characters of a given text one after the other in order to extract some specific features from the textual data. In this section, we focus on the following straightforward tasks: (a) ncomputing the absolute frequencies of all the characters occurring in a text, and (b) computing the absolute frequency of a specific character in a text.

Assume y is a text of length n over an alphabet Σ of size σ. The first task consists in counting the occurrences in y of each character of the alphabet. By using the standard representation, this problem can be solved by a simple iterative cycle that sweeps all characters of the text while increasing the corresponding counters. With the bit-layers representation, task (a) can be accomplished by scanning the text by way of a horizontal word parallelism. Initially, a table T of size 2^8 is precomputed, where, for each 8-bit vector $\hat{B}$, $T[\hat{B}]$ is an array of 8 characters (which can be regarded altogether as a 64-bit vector) such that, for $0 \leqslant k < 8$,

$$T[\hat{B}][k] \coloneqq \textbf{if } \hat{B}[k] = 0 \textbf{ then } 00000000 \textbf{ else } 00000001 \textbf{ endif}.$$

Then, during the scanning phase, for each $0 \leqslant j < \lceil n/8 \rceil$, a 64-bit vector $X \coloneqq \sum_{i=0}^{\ell-1}(T[B_i[j]] \ll i)$ is computed and, by regarding the vector X as an 8-character array, the counter of each character $X[k]$ is increased, for $0 \leqslant k < 8$.

Task (b) consists in counting the occurrences of a given input character $c \in \Sigma$ in the string y. By using the standard representation, this task can be carried through a simple iterative cycle that, while sweeping all characters of the text, whenever an occurrence of c is found it increases a counter by 1. With the bit-layers representation, task (b) can be accomplished through a scan of the text based on vertical adaptive parallelism. Initially, ℓ binary vectors P_i (each of size w), for $0 \leqslant i < \ell$, are precomputed, where

$$P_i \coloneqq \textbf{if } \rho(c)[i] = 1 \textbf{ then } 0^w \textbf{ else } 1^w \textbf{ endif}.$$

[4] The `C` codes of all tested algorithms are available at `http://www.dmi.unict.it/~faro/BLE/`

	Frequencies Count			Occurrence Count		
	genome	protein	english	genome	protein	english
Standard	0.782	0.880	0.880	1.766	1.552	1.572
Bit-Layers	**1.043**	**1.032**	**1.024**	**13.88**	**4.854**	**4.065**

Table 1. Speeds (in billions of characters per second) of the frequency and occurrence count tasks under the standard and the bit-layers representations. Best results have been reported in bold.

In the subsequent scanning phase, the bit-layers of the string y are read in blocks of w bits. Specifically, for $0 \leqslant j < \lceil n/w \rceil$, a sequence of ℓ bit vectors, $\langle X_i : 0 \leqslant i < \ell \rangle$ is computed, where $X_0 = B_0[j]$ xor P_0 and $X_i = X_{i-1}$ or $(B_i[j]$ xor $P_i)$, for $1 \leqslant i < \ell$, provided that all X_i's are nonnull. It can easily be checked that $y[jw+k] = c \iff X_{\ell-1}[k] = 1$ holds. Thus, if $X_{\ell-1} \neq 0$, the counter is increased by bitcount$(X_{\ell-1})$ units. Plainly, such procedure is adaptive, since, during each iteration, say the j-th one with $0 \leqslant j < \lceil n/w \rceil$, as soon as $X_i = 0$ for any $0 \leqslant i < \ell$, the iteration is aborted and the execution resumes with the $(j+1)$-st iteration.

Table 1 reports the experimental results relative to the above two tasks with both the standard and the bit-layers representations, where speeds are expressed in billions of characters per second. As for the frequency count task (a), the bit-layers representation allows for a slightly faster access to textual data than the standard representation, especially in the case of small alphabets, characterized by a modest number of layers. When the size of the alphabet increases, our approach unavoidably gets slower. Concerning the occurrence count task (b), the vertical adaptive parallel approach based on the bit-layers representation reaches by far the best results, as it is from 2.5 times (for English texts) up to 8 times (for genome sequences) faster than the standard approach.

4.2 Exact String Matching

The *exact string matching problem* is another basic task in text processing. It consists in finding all the (possibly overlapping) occurrences of an input pattern x of length m within a text y of length n, both strings over a common alphabet Σ of size σ. More formally, the problem aims at finding all positions j in $y[0\,..\,n-m]$ such that $y[j\,..\,j+m-1] = x$. A huge number of solutions has been devised since the 1980s [9] and, despite such a wide literature, still much work has been produced in the last few years, demonstrating that the need for efficient solutions is currently high.

We designed the following three basic solutions to the exact string matching problem, tuned to the bit-layers text representation.

Brute-force algorithm based on horizontal word parallelism (BF^*_{H}): For each position j in $y[0\,..\,n-m]$, the chunk $\hat{B}_i[j\,..\,j+m-1]$ from the i-th layer of y,

	GENOME				PROTEIN				ENGLISH			
m	4	8	16	32	4	8	16	32	4	8	16	32
BF	0.26	0.26	0.26	0.26	0.52	0.52	0.53	0.52	0.47	0.50	0.50	0.50
SA	0.55	0.54	0.55	0.55	0.50	0.50	0.50	0.52	0.50	0.50	0.50	0.50
HOR	0.27	0.35	0.40	0.40	0.45	0.68	0.90	1.05	0.43	0.65	0.86	1.05
WFR	0.49	0.91	1.29	1.71	**0.74**	1.03	1.28	1.64	**0.72**	1.03	1.30	1.65
BF^*_{H}	0.62	0.94	0.96	0.97	0.54	0.84	0.87	0.86	0.47	0.82	0.86	0.86
PFX*	0.51	1.55	1.64	1.65	0.45	1.48	1.46	1.44	0.42	1.34	1.45	1.46
SFX*	**0.79**	**1.70**	**3.10**	**5.20**	0.69	**3.35**	**2.64**	**4.50**	0.66	**1.66**	**2.85**	**4.67**

Table 2. Experimental results relative to the comparison of some string matching algorithms under the standard representation and the bit-layers representation. Speed is expressed in billions of characters per second and the best results are boldfaced.

for each $0 \leqslant i < \ell$, is compared with the corresponding layer of the pattern $\hat{D}_i[0\,..\,m-1]$. If $\hat{B}_i[j\,..\,j+m-1] = \hat{D}_i[0\,..\,m-1]$ for all $0 \leqslant i < \ell$, then a match is reported at position j. The algorithm BF^*_{H} works adaptively, since as soon as $\hat{B}_i[j\,..\,j+m-1] \neq \hat{D}_i[0\,..\,m-1]$ for some $0 \leqslant i < \ell$, the iteration for position j stops, and a new iteration starts from position $j+1$ in $y[0\,..\,n-m]$, if any.

Prefix-based algorithm based on vertical adaptive parallelism (PFX*): It is a prefix-based improvement of the algorithm BF^*_{H} described above. A prefix table $\pi\colon \{0,\ldots,2^k\} \to \{1,\ldots,k\}$, with $k = \min(m,8)$, is precomputed, where, for any given binary vector $\hat{B}$ of length k (which can be regarded as an integer in $\{0,\ldots,2^k-1\}$), we have:

$$\pi(\hat{B}) \coloneqq \min\{s : 1 < s < k \text{ and } \hat{B}[s\,..\,k-1] = \hat{B}[0\,..\,k-s-1]\} \cup \{k\}.$$

At the end of each iteration during the searching phase, the current window is shifted by $\pi(\hat{B}_0[j\,..\,j+k-1])$ positions to the right.

Suffix-based algorithm based on vertical adaptive parallelism (SFX*): It is another improvement (suffix-based) of the brute-force algorithm described earlier. A suffix table $\psi\colon \{0,\ldots,2^k\} \to \{1,\ldots,k\}$, with $k = \min(m,8)$, is precomputed, where, for any given a binary vector $\hat{B}$ of length k (which as before can be regarded as an integer in $\{0,\ldots,2^k-1\}$), we have:

$$\psi(\hat{B}) \coloneqq \min\{s : 1 \leqslant s < k \text{ and } \hat{D}[m-s-k\,..\,m-s-1] = \hat{B}[0\,..\,k-1]\} \cup \{k\}.$$

At the end of each iteration in the searching phase, the current window is shifted by $\psi(\hat{B}_0[j+m-k\,..\,j+m-1])$ positions to the right.

Table 2 shows the experimental results relative to the above algorithms (tuned to the bit-layers representation) against the following known solutions to the exact string matching problem (tuned to the standard representation),

which are considered among the fastest algorithms in practical cases [5], namely the brute-force algorithm (BF), the Boyer-Moore-Horspool algorithm (HOR) [13], the Shift-And algorithm (SA) based on bit-parallelism [2], and the Weak-Factor-Recognition algorithm (WFR) implemented with q-grams ($1 \leqslant q \leqslant 4$).

From Table 2, it turns out that algorithm BF^*_H is always faster (up to 3 times) than its counterpart BF and, in most cases, it is even faster than the Horspool algorithm (HOR). The suffix-based algorithm SFX^*, tuned to the bit-layers representation, is in almost all cases the fastest algorithm, even faster (up to 3 times) than the algorithm WFR, showing a sub-linear behaviour which rapidly improves as the length of the pattern increases.

4.3 String Matching with Mismatches

Approximate string matching has several variations. In this section we consider the δ-mismatches variation, where the task is to find all the occurrences of x with at most δ mismatches, where $0 \leqslant \delta < m$.[5]

Shift-Add [2] was the first practical algorithm for solving the δ-mismatches problem. It is based on bit-parallelism, where a vector of m states is used to represent the state of the search. A field of $m+1$ bits is used for representing each of the m states. When a mismatch is detected, the corresponding state is increased accordingly. Thus, a match is detected at a given position when the last state has a value less than or equal to δ. As in the case of other bit-parallel solutions, when $m(\log(m)+1)$ is greater than w, multiple words need to be involved in the computation.

In the context of bit-layers representation, we implemented the following algorithm:

Brute-force algorithm based on vertical adaptive parallelism (BF^*_V): Given as before a text y of length n and a pattern of length $m \leqslant n$, for each position j in $y[0\,..\,n-m]$, the chunk $\hat{B}_i[j\,..\,j+m-1]$ from the i-th layer of y, for $0 \leqslant i < \ell$, is compared with the corresponding layer $\hat{D}_i[0\,..\,m-1]$ of the pattern, by way of the xor bitwise operation. The result is a bit vector $\hat{R}_i$ of size m, such that $\hat{R}_i[k] = 1 \iff \hat{B}_i[k] \neq \hat{D}_i[k]$, for $0 \leqslant k < m$. Let $\hat{R}_{(0,i)} := \hat{R}_0 \,\mathsf{or}\, \hat{R}_1 \,\mathsf{or}\, \ldots \,\mathsf{or}\, \hat{R}_i$. If $\mathsf{bitcount}(\hat{R}_{(0,\ell-1)}) \leqslant \delta$, then an occurrence of the pattern is reported at position j. The algorithm BF^*_V works adaptively, since as soon as $\mathsf{bitcount}(\hat{R}_{(0,i)}) > \delta$, for some $i < \ell$, the iteration at position j stops, and the subsequent iteration starts from position $j+1$ in $y[0\,..\,n-m]$, if any.

Table 3 reports the experimental results relative to the comparison of three algorithms for the approximate string matching problem allowing for at most δ mismatches, with $\delta \in \{1, 3\}$. Specifically, we tested a brute-borce algorithm (BF) and the Shift-Add (SA) bit-parallel algorithm [2] (both tuned to the standard text representation) and our brute-force algorithm (BF^*_V) described above, tuned to our novel bit-layers representation.

[5] In this context, when $\delta = 0$ the approximate problem just reduces to the exact string matching problem.

		GENOME				PROTEIN				ENGLISH			
	m	4	8	16	32	4	8	16	32	4	8	16	32
$\delta = 1$	BF	0.13	0.12	0.12	0.12	0.19	0.19	0.18	0.18	0.19	0.18	0.19	0.18
	SA	**0.49**	**0.49**	0.22	0.14	**0.43**	**0.43**	0.19	0.12	**0.43**	**0.43**	0.18	0.13
	BF^*_V	0.16	0.32	**0.39**	**0.32**	0.13	0.28	**0.34**	**0.29**	0.12	0.27	**0.34**	**0.28**
$\delta = 3$	BF	0.19	0.08	0.08	0.08	0.17	0.11	0.11	0.11	0.17	0.11	0.11	0.11
	SA	**0.50**	**0.50**	0.22	0.14	**0.43**	**0.43**	0.19	0.12	**0.43**	**0.43**	0.19	0.12
	BF^*_V	0.17	0,15	**0.36**	**0.33**	0.05	0.12	**0.31**	**0.28**	0.05	0.12	**0.32**	**0.29**

Table 3. Experimental results relative to the comparison of three algorithms for the approximate string matching problem allowing for at most δ mismatches, with $\delta \in \{1, 3\}$. Speed is expressed in billions of characters per second. Best results are boldfaced.

For short patterns, our algorithm BF^*_V turns out to be slower than the Shift-Add bit-parallel algorithm, and it gets slower as the bound δ increases. This is due to the need, when the bound gets larger and larger, to go all the way in depth along the layers of the representation. However, thanks to its horizontal word parallelism, our solution shows a marked sub-linear behaviour, getting faster and faster (up to 2 times) than the Shift-Add algorithm in case of patterns of length greater than or equal to 16. In fact, as the pattern length increases, the Shift-Add algorithm gets slower and slower because of the need of involving more and more words in its computation, widening the performance gap with our algorithm BF^*_V.

5 Conclusions and Future Works

In this paper we introduced the *bit-layers text representation*, a novel succinct string representation in which textual data are arranged into a two-dimensional structure, which allows text processing to proceed both horizontally (by handling multiple data in constant time) and vertically (by moving in depth along the layers of the representation). The bit-layers text representation aims at improving text searching solutions. We believe that it may represent a valuable contribution to applications dealing with text searching. To substantiate this point, we showed how to solve some basic text processing tasks using the bit-layers representation, and presented the results of an experimental comparison of our solutions tailored to the bit-layers representation against those tuned for the standard text representation. Our solutions take particular advantage of the horizontal word parallelism and of the vertical adaptive parallelism, intrinsic in the two-dimensionality of the bit-layers representation, which could kick off several technical and creative approaches to text searching.

We plan to study other more effective solutions, tuned to the bit-layers representation, to further basic text processing problems and also to investigate other

non-standard searching problems amenable to fast solutions under our proposed text representation.

References

1. V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. Information Retrieval, 8, 151–166 (2005)
2. R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. Commun. ACM, 35(10):74-82, (1992)
3. N. R. Brisaboa, S. Ladra, G. Navarro. DACs: Bringing direct access to variable-length codes. Information Processing and Management 49, pp. 392-404 (2013)
4. R.S. Boyer, J.S. Moore. A fast string searching algorithm. Commun. ACM 20(10), 762-772 (1977)
5. D. Cantone, S. Faro, A. Pavone: Speeding Up String Matching by Weak Factor Recognition. Stringology 2017, pp. 42–50 (2017)
6. P. Elias. Efficient storage and retrieval by content and address of static files. Journal of the ACM, 21, 246-260 (1974)
7. R. Fano. On the number of bits required to implement an associative memory. Memo 61, Computer Structures Group, Project MAC, Massachusetts (1971)
8. S Faro, T Lecroq, An efficient matching algorithm for encoded DNA sequences and binary strings. In Proc. of the 20th Annual Symposium on Combinatorial Pattern Matching (CPM 2009). Lecture Notes In Computer Science, Vol. 5577, Springer-Verlag, pp.106-115 (2009)
9. S Faro, T Lecroq, The Exact Online String Matching Problem: a Review of the Most Recent Results, ACM Computing Surveys (CSUR) vol. 45 (2), pp. 13 (2013)
10. S. Faro, T. Lecroq, S. Borzì, S. Di Mauro, and A. Maggio. The String Matching Algorithms Research Tool. In *Proc. of Stringology*, pages 99–111, 2016.
11. P. Ferragina and R. Venturini. A simple storage scheme for strings achieving entropy bounds. In Proc. 18th symp. on discrete alg. (SODA), pp. 690-696 (2007)
12. R. Grossi, A. Gupta and J. Vitter. High-order entropy-compressed text indexes. In Proc. 14th symp. on discrete alg. (SODA), pp. 841-850 (2003)
13. R. N. Horspool, Practical fast searching in strings, Software: Practice & Experience 10 (6), pp. 501–506 (1980)
14. D. A. Huffman, A method for the construction of minimum-redundancy codes. Proceedings of the Institute of Radio Engineers (IRE), 40(9), pp. 1098–1101 (1952)
15. G. J. Jacobson. Succinct static data structures (Ph.D.). Pittsburgh, PA: Carnegie Mellon University (1988)
16. G. Navarro: Compact Data Structures: A Practical Approach. Cambridge University Press New York, NY, USA (2016)
17. H. Peltola, J. Tarhio, On String Matching in Chunked Texts. In Proc. of CIAA'07, Lecture Notes in Computer Science, vol. 4783, Springer Verlag, pp. 157-167 (2017)
18. J. Teuhola. Interpolative coding of integer sequences supporting log-time random access. Information Processing and Management, 47, pp. 742-761 (2011)
19. H. E. Williams and J. Zobel. Compressing integers for fast file access. Computer Journal, 42(3), pp. 193–201 (1999)
20. M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In Proc. 22nd international conference on data engineering (ICDE) (pp. 59). Washington, DC, USA: IEEE Computer Society (2006)