Algorithmica (2020) 82:3390–3412
https://doi.org/10.1007/s00453-020-00732-4

Efficient Online String Matching Based on Characters Distance Text Sampling

Simone Faro[1] · **Francesco Pio Marino**[1] · **Arianna Pavone**[2]

Received: 23 April 2018 / Accepted: 3 June 2020 / Published online: 20 June 2020

Abstract

Searching for all occurrences of a pattern in a text is a fundamental problem in computer science with applications in many other fields, like natural language processing, information retrieval and computational biology. *Sampled string matching* is an efficient approach recently introduced in order to overcome the prohibitive space requirements of an index construction, on the one hand, and drastically reduce searching time for the online solutions, on the other hand. In this paper we present a new algorithm for the sampled string matching problem, based on a characters distance sampling approach. The main idea is to sample the distances between consecutive occurrences of a given *pivot* character and then to search online the sampled data for any occurrence of the sampled pattern, before verifying the original text. From a theoretical point of view we prove that, under suitable conditions, our solution can achieve both linear worst-case time complexity and optimal average-time complexity. From a practical point of view it turns out that our solution shows a sub-linear behaviour in practice and speeds up online searching by a factor of up to 9, using limited additional space whose amount goes from 11 to 2.8% of the text size, with a gain up to 50% if compared with previous solutions.

Keywords String matching · Text processing · Efficient searching · Text indexing

✉ Simone Faro
faro@dmi.unict.it

Arianna Pavone
apavone@unime.it

[1] Dipartimento di Matematica e Informatica, Università di Catania, viale A.Doria n.6, 95125 Catania, Italy

[2] Dipartimento di Scienze Cognitive, Università di Messina, via Concezione n.6, 98122 Messina, Italy

1 Introduction

String matching is a fundamental problem in computer science and in the wide domain of text processing. It consists in finding all the occurrences of a given pattern x, of length m, in a large text y, of length n, where both sequences are composed by characters drawn from an alphabet Σ of size σ. Although data are memorized in different ways, textual data remains the main form to store information. This is particularly evident in literature and in linguistics where data are in the form of huge corpus and dictionaries. But this applies as well to computer science where large amount of data are stored in linear files. And this is also the case, for instance, in molecular biology where biological molecules are often approximated as sequences of nucleotides or amino acids. Thus the need for more and more faster solutions to text searching problems.

Applications require two kinds of solutions: *online* and *offline* string matching. Solutions based on the first approach assume that the text is not preprocessed and thus they need to scan the text *online*, when searching. Their worst case time complexity is $\Theta(n)$, and was achieved for the first time by the well known Knuth-Morris-Pratt (KMP) algorithm [16], while the optimal average time complexity of the problem is $\Theta(n \log_\sigma m/m)$ [22], achieved for example by the Backward-Dawg-Matching (BDM) algorithm [7]. Many string matching algorithms have been also developed to obtain sub-linear performance in practice [8]. Among them the Boyer-Moore-Horspool algorithm [3, 13] deserves a special mention, since it has been particularly successful and has inspired much work.

Memory requirements of this class of algorithms are very low and generally limited to a precomputed table of size $O(m\sigma)$ or $O(\sigma^2)$ [8]. However their performance may stay poor in many practical cases, especially when used for processing huge input texts and short patterns.[1]

Solutions based on the second approach tries to drastically speed up searching by preprocessing the text and building a data structure that allows searching in time proportional to the length of the pattern. For this reason such kind of problem is known as *indexed searching*. Among the most efficient solutions to such problem we mention those based on suffix trees [2], which find all occurrences in $O(m + occ)$ -worst case time, those based on suffix arrays [18], which solve the problem in $O(m + \log n + occ)$ [18], where occ is the number of occurrences of x in y, and those based on the FM-index [10] (Full-text index in Minute space), which is a compressed full-text substring index based on the Burrows-Wheeler transform allowing compression of the input text while still permitting fast substring queries. However, despite their optimal time performance,[2] space requirements of full-index data structures, as suffix-trees and suffix-arrays, are from 4 to 20 times the size of the text,

[1] Search speed of an online string matching algorithm may depend on the length of the pattern. Typical search speed of a fast solution, on a modern laptop computer, goes from 1 GB/s (in the case of short patterns) to 5 GB/s (in the case of very long patterns) [5].

[2] Search speed of a fast offline solution do not depend on the length of the text and is typically under 1 ms per query.

while the size of a compressed index, as the FM-Index, is typically less than the size of the text, but its construction may require almost the same space as that required by a full-index. Such space requirement is too large for many practical applications.

A different solution to the problem is to compress the input text and search online directly the compressed data in order to speed-up the searching process using reduced extra space. Such problem, known in literature as *compressed string matching*, has been widely investigated in the last few years. Although efficient solutions exist for searching on standard compression schemes, as Ziv-Lempel [20] and Huffman [4], the best practical behaviour are achieved by ad-hoc schemes designed for allowing fast searching [11, 15, 17, 19, 21]. These latter solutions use less than 70% of text size extra space (achieving a compression rate over 30%) and are twice as fast in searching as standard online string matching algorithms. A drawback of such solutions is that most of them still require significant implementation efforts and an high time for each reported occurrence.

A more suitable solution to the problem is *sampled string matching*, recently introduced by Claude et al. [6], which consists in the construction of a succint sampled version of the text and in the application of any online string matching algorithm directly on the sampled sequence. The drawback of this approach is that any occurrence reported in the sampled-text may require to be verified in the original text. However a sampled-text approach may have a lot of good features: it may be easy to implement, may require little extra space and may allow fast searching. Additionally it may allow fast updates of the data structure. Specifically the solution of Claude et al. is based on an alphabet reduction. Their algorithm has an extra space requirement which is only 14% of text size and is up to 5 times faster than standard online string matching on English text. Thus it turns out to be one of the most effective and flexible solution for this kind of searching problems.

1.1 Our Contribution and Organization of the Paper

In this paper we present a new approach to the sampled string matching problem based on alphabet reduction and characters distance sampling. Instead of sampling characters of the text belonging to a restricted alphabet, we divide the text in blocks of size k and sample positions of such characters inside the blocks. The sampled data is then used to filter candidate occurrences of the pattern, before verifying the whole match in the original text.

Our new approach is simple to implement and guarantees approximately the same performance as the solution proposed by Claude et al. in practice. However it is faster in the case of short patterns and, more interesting, may require only 5% of additional extra space.

We prove also that if the underlying algorithm used for searching the sampled text for the sampled pattern, achieves optimal worst-case and average-case time complexities, then our new solution attains the same optimal complexities, at least for patterns with a length of (at most) few hundreds of characters.

The paper is organized as follows. Firstly, we present in Sect. 2 the efficient sampling solution proposed by Claude et al. Then, in Sect. 3, we introduce our new

sampling approach, discuss its good features and present its practical behaviour. In Sect. 4 we compare the two approaches and present some experimental results. Finally, in Sect. 5 we draw our conclusions and discuss some further improvements.

2 Sampled String Matching

The task of the *sampled string matching* problem is to find all occurrences of a given pattern x, of length m, in a given text y, of length n, assuming that a fast and succint preprocessing of the text is allowed in order to build a data-structure, which is used to speed-up the searching phase. For its features we call such data structure a *partial-index* of the text.

In order to be of any practical and theoretical interest a partial-index of the text should:

(1) *be succint*: since it must be maintained together with the original text, it should require few additional spaces to be constructed;
(2) *be fast to build*: it should be constructed using few computational resources, also in terms of time. This should allow the data structure to be easily built online when a set of queries is required;
(3) *allow fast search*: it should drastically increase the searching time of the underlying string matching algorithm. This is one of the main features required by this kind of solutions;
(4) *allow fast update*: it should be possible to easily and quickly update the data structure if modifications have been applied on the original text. A desirable update procedure should be at least as fast as the modification procedure on the original text.

In this section we briefly describe the efficient text-sampling approach proposed by Claude et al. [6]. To the best of our knowledge it is the most effective and flexible solution known in literature for such a problem. We will refer to this solution as the Occurrence-Text-Sampling algorithm (OTS).

2.1 The Occurrence Text Sampling Algorithm

Let y be the input text, of length n, and let x be the input pattern, of length m, both over an alphabet Σ of size σ. The main idea of their sampling approach is to select a subset of the alphabet, $\hat{\Sigma} \subset \Sigma$ (the sampled alphabet), and then to construct a partial-index as the subsequence of the text (the sampled text) $\hat{y}$, of length $\hat{n}$, containing all (and only) the characters of the sampled alphabet $\hat{\Sigma}$. More formally $\hat{y}[i] \in \hat{\Sigma}$, for all $1 \leq i \leq \hat{n}$.

During the searching phase of the algorithm a sampled version of the input pattern, $\hat{x}$, of length $\hat{m}$, is constructed and searched in the sampled text. Since $\hat{y}$ contains partial information, for each candidate position i returned by the search procedure on the sampled text, the algorithm has to verify the corresponding occurrence

of x in the original text. For this reason a table ρ is maintained in order to map, at regular intervals, positions of the sampled text to their corresponding positions in the original text. The position mapping ρ has size $\lfloor \hat{n}/q \rfloor$, where q is the *interval factor*, and is such that $\rho[i] = j$ if character $y[j]$ corresponds to character $\hat{y}[q \times i]$. The value of $\rho[0]$ is set to 0. In their paper, on the basis of an accurate experimentation, the authors suggest to use values of q in the set $\{8, 16, 32\}$

Then, if the candidate occurrence position j is stored in the mapping table, i.e if $\rho[i] = j$ for some $1 \leq i \leq \lfloor \hat{n}/q \rfloor$, the algorithm directly checks the corresponding position in y for the whole occurrence of x. Otherwise, if the sampled pattern is found in a position r of $\hat{y}$, which is not mapped in ρ, the algorithm has to check the substring of the original text which goes from position $\rho[r/q] + (r \mod q) - \alpha + 1$ to position $\rho[r/q + 1] - (q - (r \mod q)) - \alpha + 1$, where α is the first position in x such that $x[\alpha] \in \hat{\Sigma}$.

Notice that, if the input pattern does not contain characters of the sampled alphabet, i.e. id $\bar{m} = 0$, the algorithm merely reduces to search for x in the original text y.

Example 1 Suppose $y =$ "abaacabdaacabcc" is a text of length 15 over the alphabet $\Sigma =$ {a,b,c,d}. Let $\hat{\Sigma} =$ {b,c,d} be the sampled alphabet, by omitting character "a". Thus the sampled text is $\hat{y} =$ "bcbdcbcc". If we map every $q = 2$ positions in the sampled text, the position mapping ρ is $\langle 5, 8, 12, 14 \rangle$. To search for the pattern $x =$ "acab" the algorithm constructs the sampled pattern $\hat{x} =$ "cb" and search for it in the sampled text, finding two occurrences at position 2 and 5, respectively. We note that $\hat{y}[2]$ is mapped and thus it suffices to verify for an occurrence starting at position 4, finding a match. However position $\hat{y}[5]$ is not mapped, thus we have to search in the substring $y[\rho(2) + 3 - 1 \cdots \rho(3)]$, finding no matches.

The above algorithm works well with most of the known pattern matching algorithms. However, since the sampled patterns tend to be short, the authors implemented the search phase using the Horspool algorithm, which has been found to be fast in such setting.

The real challenge in their algorithm is how to choose the best alphabet subset to sample. Based on some analytical results, supported by an experimental evaluation, they showed that it suffices in practice to sample the least frequent characters up to some limit.[3] Under this assumption their algorithm has an extra space requirement which is only 14% of text size and is up to 5 times faster than standard online string matching on English texts.

For the sake of completeness it has to be noticed that in [6] the authors also consider indexing the sampled text. Specifically they build a suffix array indexing the sampled positions of the text, and get a sampled suffix array. This approach is similar to the sparse suffix array [14] as both index a subset of the suffixes, but the different sampling properties induce rather different search algorithms and performance

[3] According to their theoretical evaluation and their experimental results it turns out that, when searching on an English text, the best performance are obtained when the least 13 characters are removed from the original alphabet.

characteristics. More recently Grabowsky and Raniszewski [12] proposed a more convenient indexing suffix sampling approach, with only a minimum pattern length as a requirement.

3 A New Algorithm Based on Characters Distance Sampling

In this section we present a new efficient approach to the sampled string matching problem, introducing a new method for the construction of the partial-index, which turns out to require limited additional space, still maintaining the same performance of the algorithm recently introduced by Claude et al. [6]. In the next subsections we illustrate in details our idea and describe the algorithms for the construction of the sampled text and for the searching phase.

3.1 Characters Distance Sampling

As above, let y be the input text, of length n, and let x be the input pattern, of length m, both over the alphabet Σ of length σ. We assume that all strings can be treated as vectors starting at position 1. Thus we refer to $x[i]$ as the i-th character of the string x, for $1 \leq i \leq m$, where m is the length of x.

We first define a sampled alphabet $\bar{\Sigma} \subset \Sigma$, which we call the set of *pivot characters*. The set $\bar{\Sigma}$ could be very small, and in many practical cases could be reduced to a single character. If $|\bar{\Sigma}| = 1$, the unique character of $\bar{\Sigma}$ is called the *pivot* character. For simplicity we will assume in what follows that $|\bar{\Sigma}| = 1$, which can be trivially generalized to the case where $|\bar{\Sigma}| > 1$.

The text sampling approach used in our solution is based on the following definition of *bounded position sampling*.

Definition 1 (The Bounded Position Sampling) Let y a text of length n, let $C \subseteq \Sigma$ be the set of pivot characters and let n_c be the number of occurrences of any character of C in the input text y. First we define the *position function*, $\delta : \{1, \ldots, n_c\} \rightarrow \{1, \ldots, n\}$, where $\delta(i)$ is the position of the i-th occurrence of any character of C in y. Formally we have

$$\begin{array}{lll} (i) & 1 \leq \delta(i) < \delta(i+1) \leq n & \text{for each } 1 \leq i \leq n_c - 1 \\ (i) & y[\delta(i)] \in C & \text{for each } 1 \leq i \leq n_c \\ (i) & y[\delta(i) + 1 \ldots \delta(i+1) - 1] \in C & \text{for each } 0 \leq i \leq n_c \end{array}$$

where, in (iii), we assume that $\delta(0) = 0$ and $\delta(n_c + 1) = n + 1$.

Assume now that k is a given threshold constant. We define the *k-bounded position function*, $\delta_k : \{1, \ldots, n_c\} \rightarrow \{0, \ldots, k-1\}$, where $\delta_k(i)$ is the position (modulus k) of the i-th occurrence of any character of c in y. Formally we have

$$\delta_k(i) = [\delta(i) \mod k], \text{ for each } i = 1, \ldots, n_c$$

The *k-bounded-position sampled version* of y, indicated by $\dot{y}$, is a numeric sequence, of length n_c defined as

$$\dot{y} = \langle \delta_k(1), \delta_k(2), \ldots, \delta_k(n_c) \rangle. \tag{1}$$

Plainly we have $0 \leq \dot{y}[i] < k$, for each $1 \leq i \leq n_c$.

Example 2 Suppose $y =$ "abaacbcabdada" is a text of length 13, over the alphabet $\Sigma =$ {a,b,c,d}. Let $C =$ {"a", "c"} be the set of pivot characters and let $k = 5$ be the threshold value. Thus the position sampled version of y is $\dot{y} = \langle 1, 3, 4, 0, 2, 3, 1, 3 \rangle$. Specifically the first occurrence of a character in C is at position 1 ($y[1] =$ a), its second occurrence is at position 3 ($y[3] =$ a). However its 5-th occurrence is at position 7, thus $\dot{y}[5] = [\delta(7) \mod 5] = [7 \mod 5] = 2$.

Example 3 Suppose $y =$ "abaacbcabdada" is a text of length 13, over the alphabet $\Sigma =$ {a,b,c,d}. Let "a" be the pivot character and let $k = 5$ be the threshold value. Thus the position sampled version of y is $\dot{y} = \langle 1, 3, 4, 3, 1, 3 \rangle$. Specifically the first occurrence of character "a" is at position 1, its second occurrence is at position 3. However its 4-th occurrence is at position 8, thus $\dot{y}[4] = [\delta(4) \mod 5] = [8 \mod 5] = 3$.

Definition 2 (The Characters Distance Sampling) Let $c \in \Sigma$ be the pivot character, let $n_c \leq n$ be the number of occurrences of the pivot character in the text y and let δ be the position function of y. We define the *characters distance function* $\Delta(i) = \delta(i+1) - \delta(i)$, for $1 \leq i \leq n_c - 1$, as the distance between two consecutive occurrences of the character c in y.

The *characters-distance sampled version* of the text y is a numeric sequence, indicated by $\bar{y}$, of length $n_c - 1$ defined as

$$\bar{y} = \langle \Delta(1), \Delta(2), \ldots, \Delta(n_c - 1) \rangle. \tag{2}$$

Plainly we have

$$\sum_{i=1}^{n_c - 1} \Delta(i) \leq n - 1.$$

Example 4 As in Example 3, let $y =$ "abaacbcabdada" be a text of length 13, over the alphabet $\Sigma =$ {a,b,c,d}. Let "a" be the pivot character. Thus the character distance sampled version of y is $\bar{y} = \langle 2, 1, 4, 3, 2 \rangle$. Specifically $\bar{y}[1] = \Delta(1) = \delta(1) - \delta(0) = 3 - 1 = 2$, while $\bar{y}[3] = \Delta(4) = \delta(4) - \delta(3) = 8 - 4 = 4$, and so on.

In order to be able to retrieve the original i-th position $\delta(i)$, of the pivot character, from the i-th element of the k-bounded position sampled text $\dot{y}$, we also maintain a *block-mapping table* τ which stores the indexes of the last positions of the pivot character in each k-block of the original text, for a given input block size k.

Specifically, we assume that the text y is divided in $\lceil n/k \rceil$ blocks of length k, with the last block containing $(n \mod k)$ characters. Then $\tau[i] = j$ if the j-th occurrence

of the pivot character in y is also its last occurrence in the i-th block. If the i-th block of y does not contain any occurrence of the pivot character than $\tau[i]$ is set to be equal to the last position of the pivot character in one of the previous blocks.

More formally we have, for $1 \leq i \leq \lceil n/k \rceil$

$$\tau[i] = \max\left(\{j : \delta(j) \leq ik\} \cup \{0\}\right) \tag{3}$$

Thus it is trivial to prove that $\tau[i] = j$ if and only if $\delta(j) \leq (ik)$ and $\delta(j+1) > (ik)$. In addition the values in the block mapping τ are stored in a non decreasing order. Formally

$$\tau[i] \leq \tau[i+1],\ \forall\, 0 \leq i \leq \lceil n/k \rceil \tag{4}$$

Example 5 Let $y =$ "caacbddcbcabbacdcadcab" be a text of length 22, over the alphabet $\Sigma =$ {a,b,c,d}. Let "a" be the pivot character and let $k = 5$ be the block size. The k-bounded position sampled version of y is $\dot{y} = \langle 2, 3, 1, 4, 3, 1\rangle$. The text is divided in $\lceil 22/5 \rceil = 5$ blocks, where the last block contains only 2 characters. Thus the mapping table τ contains exactly 5 entries, and specifically $\tau[1] = 2$, since the last occurrence of the pivot character in the first block corresponds to its second occurrence in y. Similarly, $\tau[3] = 4$, $\tau[4] = 5$ and $\tau[5] = 6$. Observe however that, since the second block does not contain any occurrence of the pivot character we have $\tau[2] = \tau[1] = 2$.

The following Lemma 1 defines how to compute the index of the block in which the j-th occurrence of the pivot character is located.

Lemma 1 *Let y be a text of length n, let $c \in \Sigma$ be the pivot characters and assume c occurs n_c times in y. In addition let $\dot{y}$ be the k-bounded-position sampled versions of y, and let τ be corresponding mapping table. Then the j-th occurrence of the pivot character in y occurs in the i-th block of y if $\tau[i] \geq j$ and $\tau[i-1] < j$.*

Proof Assume that the j-th occurrence of the pivot character in y occurs in the i-th block and let $\delta(h_1)$ be the position of the last occurrence of the pivot character in the $(i-1)$-th block. Since the character of position $\delta(j)$ occurs in the i-th block we have $j > h_1$. By definition we have $\tau[i-1] = h_1 < j$.

Moreover let $\delta(h_2)$ be the position of last occurrence of the pivot character int the i-th block. We plainly have $\delta(h_2) \geq \delta(j)$, $j < ik$ and $h_2 < ik$. Thus by equation (3) we have $\tau[i] \geq j$. Proving the lemma. □

The following Corollary 1 defines the relation for computing the original position $\delta(j)$ from $\dot{y}[j]$ and the mapping table τ. It trivially follows from Lemma 1 and equation (4).

Corollary 1 *Let y be a text of length n, let $c \in \Sigma$ be the pivot character and assume c occurs n_c times in y. In addition let $\dot{y}$ be the k-bounded-position sampled versions of y, and let τ be corresponding mapping table. Let $b = \min\{i : \tau[i] \geq j\}$, then we have*

```
GET-POSITION(τ, b, ẏ, i)
1.  while τ[b] < i do
2.      b ← b + 1
3.  p ← (b − 1) × k + ẏ[i]
4.  return (p, b)
```

```
COMPUTE-CHARACTER-DISTANCE-SAMPLING(ẏ, τ)
1.  ȳ ← ⟨⟩
2.  n_c ← len(ẏ)
3.  b ← 1
4.  (δ_1, b) ← GET-POSITION(τ, b, ẏ, 1)
5.  for i ← 2 to n_c do
6.      (δ_i, b) ← GET-POSITION(τ, b, ẏ, i)
7.      ȳ[i − 1] ← δ(i) − δ(i − 1)
8.  return ȳ
```

Fig. 1 (On the left) The pseudocode of procedure GET-POSITION which computes the index of the block corresponding the i-th occurrence of the pivot character in y. (On the right) The pseudocode of procedure COMPUTE-CHARACTER-DISTANCE-SAMPLING which computes, on the flight from $\dot{y}$, the Characters Distance sampled version of y

$$\delta(j) = (\tau[b] - 1)k + \dot{y}[j].$$

□

Based on the relation defined by Corollary 1 the pseudocode shown in Fig. 1 (on the left) describes a procedure for computing the correct position $\delta(i)$ in y, from the i-th element of the k-bounded position sampled text $\dot{y}$. In the pseudocode of GET-POSITION$(\tau, b, \dot{y}, i)$ we assume that the parameter b is less or equal the actual block of the i-th occurrence of the pivot character, i.e. $bk \leq \delta(i)$. It returns a couple (p, b), where $p = \delta(i)$ and b is such that $(b-1)k < \delta(i) \leq bk$.

The following lemma introduces an efficient way for computing $\bar{y}$ from $\dot{y}$ and τ.

Lemma 2 *Let y be a text of length n, let $c \in \Sigma$ be the pivot character and assume c occurs n_c times in y. In addition let $\dot{y}$ and $\bar{y}$ be the k-bounded-position and character-distance sampled versions of y, respectively. If $b = \min\{j : \tau[j] \geq i+1\}$ and $a = \min\{j : \tau[j] \geq i\}$, then the following relation holds*

$$\bar{y}[i] = \dot{y}[i+1] + (\tau[a] - \tau[b])k - \dot{y}[i] \tag{5}$$

Proof Le $b = \min\{j : \tau[j] \geq i+1\}$ and $a = \min\{j : \tau[j] \geq i\}$. By corollary 1 we have

$$\begin{aligned} \bar{y}[i] = \Delta(i) = & \quad \text{by Definition 2} \\ \delta(i+1) - \delta(i) = & \\ (\tau[b] - 1)k + \dot{y}[i+1] - (\tau[a] - 1)k + \dot{y}[i] = & \quad \text{by Corollary 1} \\ \dot{y}[i+1] + (\tau[a] - \tau[b])k - \dot{y}[i] & \end{aligned}$$

proving the Lemma. □

Based on the formula introduced by Lemma 2 the pseudocode shown in Fig. 1 (on the right) describes a procedure for computing on the flight from $\dot{y}$ the character distance sampled version of the text y. It is easy to prove that the time complexity of this procedure is given by $O(n_c + n/k)$, where a time $O(n_c)$ is required for scanning the sequence $\dot{y}$, while a time $O(n/k)$ is required for scanning the mapping table τ. If

we suppose equiprobability and independence of characters, the number of expected occurrences of the pivot character is $\mathbf{E}(n_c) = n/\sigma$. Thus under the assumption[4] that $k \geq \sigma$ the overall average time complexity of this procedure is $O(n_c)$. Its worst-case time complexity is plainly $O(n)$.

Under the specific assumption that $\Delta(i) < k$, for each $1 \leq i \leq n_c - 1$, the sequence $\bar{y}$ can be computed in a more efficient way. It is defined by the following Corollary 2 which trivially follows from Lemma 2.

Corollary 2 *Let y be a text of length n, let $c \in \Sigma$ be the pivot character and assume c occurs n_c times in y. In addition let $\dot{y}$ and $\bar{y}$ be the k-bounded-position and character-distance sampled versions of y, respectively. Then, if we assume that $\Delta(i) < k$, for each $1 \leq i < n_c$, the following relation holds*

$$\bar{y}[i] = \begin{cases} \dot{y}[i+1] - \dot{y}[i] & \text{if } \dot{y}[i+1] > \dot{y}[i] \\ \dot{y}[i+1] + k - \dot{y}[i] & \text{otherwise} \end{cases} \tag{6}$$

Proof Assume that the i-th occurrence of the pivot character is in the j-th block. Since $\Delta(j) < k$ by definition, the $(i+1)$-th occurrence of the pivot character is in block j or (at most) in block $(j+1)$.

Plainly, if $\dot{y}[i+1] \leq \dot{y}[i]$, the two pivot characters occurs into different (consecutive) blocks. Thus $\Delta(i) = \dot{y}[i+1] + k - \dot{y}[i]$.

Conversely, assume now that $\dot{y}[i+1] > \dot{y}[i]$. The $(i+1)$-th occurrence of the pivot character cannot be in block $j+1$, because is such a case we should have $\Delta(i) = \dot{y}[i+1] + k - \dot{y}[i] > k$. Thus the $(i+1)$-th occurrence of the pivot character is in block j and $\Delta(i) = \dot{y}[i+1] - \dot{y}[i] > k$. □

We are now ready to describe the preprocessing and the searching phase of our new proposed algorithm.

3.2 The Preprocessing Phase

In this section we describe the preprocessing phase of the algorithm. Figure 2 shows the presudocode for computing the two sampled versions of the text. Assuming that the maximum distance between two consecutive occurrences of the pivot character is bounded by k, by Definition 1 and by Definition 2, the sequences $\dot{y}$ and $\bar{y}$ both require $(n_c)\log(k)$ bits to be maintained. In practical cases we can chose $k = 256$ using a single byte to store each value of the sampled sequences and maintaining the assumption $\Delta(i) < k$ more feasible. This will allow us to store the sampled text using only n_c bytes.

Figure 3 reports the maximum and average distances between two consecutive occurrences, computed for the most frequent characters in a natural language text. Observe that the first 6 most frequent characters follow the constraint on the maximum distance.

[4] In practical cases we can implement our solution with a block size $k = 256$, which allows to represent the elements of the sequence $\dot{y}$ using a single byte. In such a case the assumption $k \geq \sigma$ is plausible for any practical application.

```
COMPUTE-DISTANCE-SAMPLING(y, n, Σ̄)
1.  ȳ ← ⟨⟩
2.  j ← 0
3.  p ← 0
4.  for i ← 1 to n do
5.    if y[i] ∈ Σ̄ then
6.      j ← j + 1
7.      ȳ[j] ← i − p
8.      p ← i
9.  return (ȳ, j)
```

```
COMPUTE-POSITION-SAMPLING(y, n, Σ̄, k)
1.  ẏ ← ⟨⟩
2.  τ ← a table of ⌈n/k⌉ entries
3.  for i ← 1 to ⌈n/k⌉ do τ[i] ← 0
4.  j ← 0
5.  for i ← 1 to n do
6.    if τ[⌈i/k⌉] = 0 then
7.      τ[⌈i/k⌉] ← τ[⌈i/k⌉ − 1]
8.    if y[i] ∈ Σ̄ then
9.      j ← j + 1
10.     ẏ[j] ← i mod k
11.     τ[⌈i/k⌉] ← j
12. return (ẏ,j,τ)
```

Fig. 2 (On the left) The pseudocode of procedure COMPUTE-DISTANCE-SAMPLING for the construction of the *character distance sampling* version of a text y. (On the right) The pseudocode of procedure COMPUTE-POSITION-SAMPLING for the construction of the *character position sampling* version of a text y

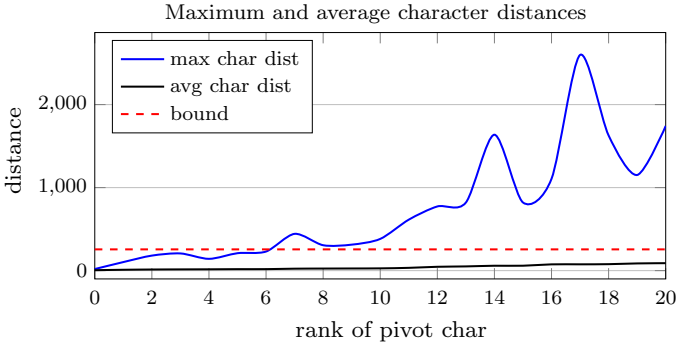

Fig. 3 Maximum and average distances between two consecutive occurrences, computed for the most frequent characters in a natural language text. On the x axis characters are ordered on the base of their rank value, in a non decreasing order. The red line represents the bound $k = 256$

As a consequence the choice of the pivot character directly influences the additional memory used for storing the sampled text (the larger is the rank of the pivot character the shorter is the resulting sampled text) and the performance of the searching phase, as we will see later in Sect. 4.

Let y be an input text of length n over the alphabet Σ, of size σ, and let r be a constant input parameter. Specifically r is the rank of the pivot character to be selected for building the sampled text. The rank r must be chosen in order to have $\Delta(i) < k$, for each $1 \leq i \leq n_c$.

The first step of the preprocessing phase of the algorithm consists in counting the frequencies of each character $c \in \Sigma$ and in computing their corresponding ranks.

Subsequently the algorithm builds and store the k-bounded position sampled text $\dot{y}$. This step requires $O(r \log(n) + n)$-time (since only the first r characters need to be ordered) and $O(n_c)$-space, where n_c is the number of occurrences of the pivot character in y.

```
VERIFY(x, m, y, s)
1.  i ← 0
2.  while i ≤ m and x[i] = y[s + i] do
3.      i ← i + 1
4.  if i = m then
5.      report occurrence at s
```

```
RESTORINGVERIFY(x, m, y, s, k, D(x), q_0)
1.  q ← q_0
2.  c ← 0
3.  for i ← s to s + k − 1 do
4.      q ← δ(q, y[i])
5.      if (q = m) then
6.          report occurrence at i − m + 1
```

Fig. 4 The pseudocode of procedure VERIFY (on the left) for testing the occurrence of a pattern x, of length m, in a text y, of length n, starting at position s; and the pseudocode of procedure RESTORINGVERIFY (on the right) based on the string matching automaton of x and used to remember those positions of the text which have been already processed during a previous verification

Each element of the mapping table τ can be stored using $\log(n)$ bit. Therefore the overall space complexity of the algorithm is $O(n_c + n\log(n)/k)$.

3.3 The Searching Phase

Let x be an input pattern of length m and let $c \in \Sigma$ be the pivot character. Let m_c be the number of occurrences of the pivot character in x. The searching phase can be then divided in three different subroutines, depending on the value of m_c. All searching procedures work using a filtering approach. They takes advantage of the partial-index precomputed during the preprocessing phase in order to quickly locate any candidate substring of the text which may include an occurrence of the pattern.

If such candidate substring has length m the algorithm simply performs a character-by-character comparison between the pattern and the substring. Figure 4 shows the pseudocode of two procedures which can be used by the algorithm for testing the occurrence of a pattern x, of length m, in a text y, of length n, starting at position s:

- procedure VERIFY (on the left) plainly compares the substring of the text $y[s \cdots s+m-1]$ and the pattern x, character by character, proceeding from left to right until a mismatch if found or the two strings turn out to be equal. In this case the procedure reports an occurrence at position s of the text. Its complexity is plainly $O(m)$ in the worst case.
- procedure RESTORINGVERIFY (on the right) is designed to remember those positions of the text which have been already processed during a previous verification. Specifically it makes use on the string matching automaton [1] (SMA for short) $\mathcal{D}(x)$ for a given pattern x of length m. Specifically the SMA $\mathcal{D}(x)$ is a quintuple $\mathcal{D}(x) = \{Q, q_0, F, \Sigma, \delta\}$ where $Q = \{q_0, q_1, \ldots, q_m\}$ is the set of states; q_0 is the initial state; $F = \{q_m\}$ is the set of accepting states and $\delta : Q \times \Sigma \to Q$ is the transition function defined as

$$\delta(q_i, c) = \begin{cases} q_{i+1} & \text{if } c = x[i+1] \\ q_{|k|}, \text{where } k \text{ is the longest border of } x[0 \cdots i-1]c & \text{otherwise} \end{cases}$$

  for $0 \leqslant i \leqslant m$ and $c \in \Sigma$. The language recognized by $\mathcal{D}(x)$ is $\mathcal{L}(\mathcal{D}(x)) = \Sigma^* x$. The procedure simply scans the text character by character, from left to right,

```
SEARCH-0(x, ẏ, y)
Δ     We assume c does not occur in x
1.    m ← len(x)
2.    n_c ← len(ẏ)
3.    b ← 1
4.    δ_0 ← 0
5.    for i ≤ 2 to n_c do
6.        (δ_i, b) ← GET-POSITION(τ, b, ẏ, i)
7.        if (δ_i − δ_{i−1} > m) then
8.            search for x in y[δ_{i−1} + 1..δ_i − 1]
9.    if (n + 1 − δ_{n_c} > m) then
10.       search for x in y[δ_{n_c} + 1..n]
```

Fig. 5 The pseudocode of procedure SEARCH-0 for the sampled string matching problem, when the pivot character does not occur in the input pattern x

performing transitions on the automaton $\mathcal{D}(x)$. If the final state is reached after the character $y[i]$ is read, then a match is reported at position $(i - m + 1)$.

When the candidate substring has length greater than m, then a searching procedure is called, based on any exact online string matching algorithm. As we will discuss later, if we suppose that the underlying algorithm has a linear worst case time complexity, as in the case of the KMP algorithm [16] or the Automaton Matcher [1], then our solution achieves the same complexity in the worst case. Similarly if we suppose to implement the searching procedure using an optimal average string matching algorithm, like the BDM algorithm [13], the resulting solution achieves an optimal $O(n \log_\sigma m/m)$ average time complexity.

In what follows we describe in details the three different searching procedures which are applied when $m_c = 0$, $m_c = 1$ and $m_c > 1$, respectively.

3.3.1 Case 1: $m_c = 0$

Consider firstly the case where the pattern does not contain any occurrence of the pivot character c, so that m_c is equal to zero. Under this assumption the algorithm searches for the pattern x in all substrings of the original text which do not contain the pivot character. Specifically such substrings are identified in the original text by the intervals $[\delta(i) + 1 \cdots \delta(i + 1) - 1]$, for each $0 \leq i \leq n_c$, assuming $\delta(0) = 0$ and $\delta(n_c + 1) = n + 1$.

Figure 5 shows the pseudocode of the algorithm which searches for all occurrences of a pattern x, when the pivot character c does not occur in it. Specifically, for each $1 \leq i \leq n_c + 1$, the algorithm checks if the interval $\delta(i) - \delta(i - 1)$ is greater than m (lines 5-7). In such a case the algorithm searches for x in the substring of the text $y[\delta(i - 1) + 1 \cdots \delta(i) - 1]$ (line 8) using any standard string matching algorithm. Otherwise the substring is skipped, since no occurrence of the pattern could be found at such position.

The following theorem proves that procedure SEARCH-0 achieves optimal worst-case and average-case time complexity.

Theorem 1 *Le x and y be two strings of size m and n, respectively, over an alphabet Σ of size $\sigma > 1$. Let $c \in \Sigma$ be the pivot character and let $\dot{y}$ be the k-bounded*

position sampled version of the text y. Under the assumption of equiprobability and independence of characters in Σ, *the worst-case and average time complexity of the* SEARCH-0 *algorithm are* $O(n)$ *and* $O(n \log_\sigma m/m)$, *respectively.*

Proof In order to evaluate the worst-case time complexity of procedure SEARCH-0, we can notice that each substring of the text is scanned at least once in line 10, with no overlap. Thus if we use a linear algorithm to perform the standard search then it is trivial to prove that the whole searching procedure requires

$$T^0_{\mathsf{wst}}(n) = O(m) + \sum_{i=1}^{n_c-1} O(\Delta(i)) + O\left(\frac{n}{k}\right) = O(n).$$

Assuming that the underlying algorithm has an $O(n \log m/m))$ average time complexity, on a text of length n and a pattern of length m, we can express the expected average time complexity as

$$T^0_{\mathsf{avg}}(n) = \sum_{i=1}^{n_c-1} O\left(\frac{\Delta(i) \log_\sigma m}{m}\right) + O\left(\frac{n}{k}\right) = O\left(\frac{n \log_\sigma m}{m}\right).$$

for enough great values $k \geq m/\log_\sigma m$ and $k \geq \sigma$. □

Observe that, in the case of a natural language text, where generally $\sigma \geq 100$, a block size $k = 256$ is enough to reach the optimal average time complexity for any (short enough) pattern of length $m < 256$.

3.3.2 Case 2: $m_c = 1$

Suppose now the case where the pattern x contains a single occurrence of the pivot character c, so that the length of the sampled version of the pattern is still equal to zero. The algorithm efficiently takes advantage of the information precomputed in $\dot{y}$ using the positions of the pivot character c in y as an anchor to locate all candidate occurrences of x.

Figure 6 shows the pseudocode of the algorithm which searches for all occurrences of a pattern x, when the pivot character c occurs only once in it. Specifically, let α be the unique position in x which contains the pivot character (line 3), i.e. we assume that $x[\alpha] = c$ and $x[1 \cdots \alpha - 1]$ does not contain c. Then, for each $0 \leq i \leq n_c - 1$, the algorithm checks if the interval $\delta(i-1) - \delta(i-2)$ is greater than $\alpha - 1$ and if the interval $\delta(i) - \delta(i-1)$ is greater than $m - \alpha$ (lines 7-9). In such a case the algorithm merely checks if the substring of the text $y[\delta(i) - \alpha + 1 \cdots \delta(i) + m - \alpha]$ is equal to the pattern (line 10). Otherwise the substring is skipped. As before we assume that $\delta(0) = 0$ (line 5) and $\delta(n_c + 1) = n + 1$ (line 11). The last alignment of the pattern in the text is verified separately at the end of the main cycle (lines 11-12).

The following theorem proves that procedure SEARCH-1 achieves optimal worst-case and average-case time complexity.

```
SEARCH-1(x, ẏ, y)
Δ     We assume c occurs once in x
1.    m ← len(x)
2.    n_c ← len(ẏ)
3.    α ← min{i : x[i] = c}
4.    b ← 1
5.    δ_0 ← 0
6.    (δ_1, b) ← GET-POSITION(τ, b, ẏ, 1)
7.    for i ≤ 2 to n_c do
8.        (δ_i, b) ← GET-POSITION(τ, b, ẏ, i)
9.        if (δ_{i−1} − δ_{i−2} > α − 1 and δ_i − δ_{i−1} > m − α) then
10.           VERIFY(x, m, y, δ_{i−1} − α + 1)
11.   if (δ_{n_c} − δ_{n_c−1} > α − 1 and n + 1 − δ_{n_c} > m − α) then
12.       VERIFY(x, m, y, δ_{n_c} − α + 1)
```

Fig. 6 The pseudocode of procedure SEARCH-1 for the sampled string matching problem, when the pivot character occurs once in the input pattern x

Theorem 2 *Le x and y be two strings of size m and n, respectively, over an alphabet Σ of size $\sigma > 1$. Let $c \in \Sigma$ be the pivot character and let $\dot{y}$ be the k-bounded position sampled version of the text y. Under the assumption of equiprobability and independence of characters in Σ, the worst-case and average time complexity of the SEARCH-1 algorithm are $O(n)$ and $O(n \log_\sigma m/m)$, respectively.*

Proof In order to evaluate the worst-case time complexity of the algorithm notice that each character could be involved in, at most, two consecutive checks in line 10. Specifically any text position in the interval $[\delta(i-1)+1 \cdots \delta(i)-1]$ could be involved in the verification of the substrings $y[\delta(i-1)-\alpha+1 \cdots \delta(i-1)+m-\alpha]$ and $y[\delta(i)-\alpha+1 \cdots \delta(i)+m-\alpha]$. Thus the overall worst case time complexity of the searching phase is $T^1_{\mathsf{wst}}(n) = O(n)$.

In order to evaluate the average-case time complexity of the algorithm notice that the expected number of occurrences in y of the pivot character is given by $\mathbf{E}(n_c) = n/\sigma$. Moreover, for any candidate occurrence of x in y, the number $\mathbf{E}(\text{insp})$ of expected character inspections performed by procedure VERIFY, when called on a pattern of length m, is given by

$$\mathbf{E}(\text{insp}) = 1 + \sum_{i=1}^{m-1} \left(\frac{1}{\sigma}\right)^i \leq \frac{\sigma}{\sigma - 1}$$

Thus the average time complexity of the algorithm ican be expressed by

$$T^1_{\mathsf{avg}}(n) = \mathbf{E}(n_c) \cdot \mathbf{E}(\text{insp}) = O\Big(\frac{n}{\sigma}\Big) \cdot O\Big(\frac{\sigma}{\sigma-1}\Big) = O\Big(\frac{n}{\sigma-1}\Big)$$

```
SEARCH-2+(x, ẏ, y)
Δ    We assume c occurs more than once in x
1.   m ← len(x)
2.   ṅ ← len(ẏ)
3.   α ← min{i : x[i] = c}
4.   b ← 1
5.   (x̄, m̄) ← COMPUTE-DISTANCE-SAMPLING(x, m, {c})
6.   search for x̄ in ȳ :
7.        for each i such that x̄ = ȳ[i..i + m̄ − 1] do
8.             (δ_i, b) ← GET-POSITION(τ, b, i)
9.             VERIFY(x, m, y, δ_i − α)
```

Fig. 7 The pseudocode of procedure SEARCH-2 for the sampled string matching problem, when the pivot character occurs more than once in the input pattern x

obtaining the optimal average time complexity $O(n \log_\sigma m/m))$ for great enough alphabets of size $\sigma > (m/\log_\sigma m) + 1$, and for $k \geq \sigma$. □

3.3.3 Case 3: $m_c \geq 2$

If the number of occurrences of the pivot character in x are more than 2 (i.e. if $m_c \geq 2$ and $\bar{m} \geq 1$) then the algorithm uses the sampled text $\bar{y}$ to compute on the fly the character-distance sampled version of y and use it to searching for any occurrence of $\bar{x}$. This is used as a filtering phase for locating in y any candidate occurrence of x.

Figure 7 shows the pseudocode of the algorithm which searches for all occurrences of a pattern x, when the pivot character c occurs more than once in it. First the character distance sampled version of the pattern $\bar{x}$ is computed (line 5). Then the algorithm searches for $\bar{x}$ in the $\bar{y}$ using any exact online string matching algorithm (line 6). Notice that $\bar{y}$ can be efficiently retrieved online from the sampled text $\dot{y}$, using relation given in (6).

For each occurrence position j of $\bar{x}$ in $\bar{y}$ an additional procedure must be run to check if such occurrence corresponds to a match of the whole pattern x in y (lines 7-9). For this purpose the algorithm checks if the substring of the text $y[\delta(j) - \alpha \cdots \delta(j) - \alpha + m - 1]$ is equal to x, where α, as before, is the first position in x where the pivot character occurs. Notice that the value of $\delta(j)$ can be obtained in constant time from $\dot{y}[j]$ (line 8).

The following theorem proves that procedure SEARCH-2 achieves optimal worst-case and average-case time complexity.

Theorem 3 *Le x and y be two strings of size m and n, respectively, over an alphabet Σ of size $\sigma > 1$. Let $c \in \Sigma$ be the pivot character and let $\dot{y}$ be the k-bounded position sampled version of the text y. Under the assumption of equiprobability and independence of characters in Σ, the worst-case and average time complexity of the SEARCH-2 algorithm are $O(n)$ and $O(n \log_\sigma m/m)$, respectively.*

Proof In order to evaluate the worst-case time complexity of the algorithm in this last case notice that, if we use a linear algorithm to search $\bar{y}$ for $\bar{x}$, the overall time complexity of the searching phase is $O(n_c + n_x m + n/k)$, where n_x is the number of occurrences of $\bar{x}$ in $\bar{y}$. In the worst case it translates in $O(n_c m)$ worst case time complexity.

However we can implement the algorithm by substituting procedure VERIFY with procedure RESTORINGVERIFY (Fig. 4) in order to allow the algorithm to remember all positions of the text which have been already processed. In this case it is required to maintain, at the end of each iteration of the FOR loop of line 7, the position r of the last character which has been processed and the last state q_r reached by the automaton during the current verification. Thus, if a next verification is required, starting at position $\delta_i - \alpha$, the algorithm performs a call to

$$\text{RESTORINGVERIFY}(x, m, y, s, \delta_i - \alpha + m - s, \mathcal{D}(x), q_r)$$

assuming $s = \max(\delta_i - \alpha, r + 1)$ and where we remember that $\mathcal{D}(x)$ is the string matching automaton of the pattern x which can be constructed during the preprocessing phase of the algorithm. This allows the algorithm to run in overall $T^{2+}_{\text{wst}}(n) = O(n)$ worst-case time complexity.

In order to evaluate the average-case time complexity of the algorithm notice that time required for searching $\bar{x}$ in $\bar{y}$ is $O(n_c \log m_c / m_c) + n/k)$. Moreover, observe that the number of verification is bounded by the expected number of occurrences of the pivot character in y, thus, following the same line of Theorem 2, the overall average time complexity of the verifications phase is $O(n/(\sigma - 1))$. Thus the average time complexity of the algorithm can be expressed by

$$T^{2+}_{\text{avg}}(n) = O\left(\frac{n_c \log m_c}{m_c}\right) + O\left(\frac{n}{k}\right) + O\left(\frac{n}{\sigma - 1}\right)$$

obtaining the optimal average time complexity $O(n \log_\sigma m/m)$ for great enough values of k and σ, such that $k \geq \sigma \geq (m/\log_\sigma m) + 1$. □

4 Experimental Results

In this section we report the results of an extensive evaluation of the new presented algorithm in comparison with the OTS algorithm by Claude et al. [6] for the sampled string matching problem. The algorithms have been implemented in C, and have been tested using a variant of the SMART tool [9], properly tuned for testing string matching algorithms based on a text-sampling approach, and 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.[5] Although all the

[5] The SMART tool is available online for download at http://www.dmi.unict.it/~faro/smart/ or at https://github.com/smart-tool/smart.

evaluated algorithms could be implemented in conjunction with any online string matching algorithm, for the underlying searching procedure, for a fair comparison in our tests we realise implementations in conjunction with the Horspool algorithm, following the same line of the experimental evaluation conducted in [6].

Comparisons have been performed in terms of preprocessing times, searching times and space consumption. For our tests, we used the English text of size 5MB provided by the research tool Smart, available online for download.[6]

In the experimental evaluation, patterns of length m were randomly extracted from the text (thus the number of reported occurrences is always greater than 0), with m ranging over the set of values $\{2^i \mid 2 \leq i \leq 8\}$. In all cases, the mean over the running times (expressed in ms) of 1000 runs has been reported.

4.1 Space Requirements

In the context of text-sampling string-matching space requirement is one of the most significant parameter to take into account. It indicates how much additional space, with regard to the size of the text, is required by a given solution to solve the problem.

Conversely to other kind of algorithms, like compressed-matching algorithms, which are allowed to maintain the input strings in a compressed form (with a detriment in processing time), algorithms in text-sampling matching require to store the whole text together with the additional sampled-text which is used to speed-up the searching phase. In this context they are much more similar to online string-matching solutions and, although they have the additional good property to allow a direct access to the input text, to be of any practical interest they should require as little extra space as possible.

Figure 8 shows the space consumption of the text-sampling algorithms described in this paper. Specifically, memory space required by the new algorithm is plotted on variations of the pivot character, while memory space required by the Ots algorithm is plotted on variations of the size of the set of removed characters. Both values range from 2 to 20.

Space consumption of the Ots algorithm ranges from 4 MB (80% of text size), when only 2 characters are removed and $q = 8$, to 740 KB (14.8% of text size), in the case of 20 removed characters and $q = 32$. As expected, the function which describes memory requirements follows a decreasing trend while the value of σ^* increases.

A similar trend can be observed for our new algorithm, whose space consumption decreases while the rank of the pivot character increases. However, in this case, space requirement ranges from 557 KB (11% of text size), if we select the most frequent character as a pivot, to 142 KB (2.8% of text size), when the pivot is the character with rank position 20. Thus the benefit in space consumption

[6] Specifically, the text buffer is the concatenation of two different texts: The King James version of the bible (3.9 MB) and The CIA world fact book (2.4 MB). The first 5MB of the resulting text buffer have been used in our experimental results.

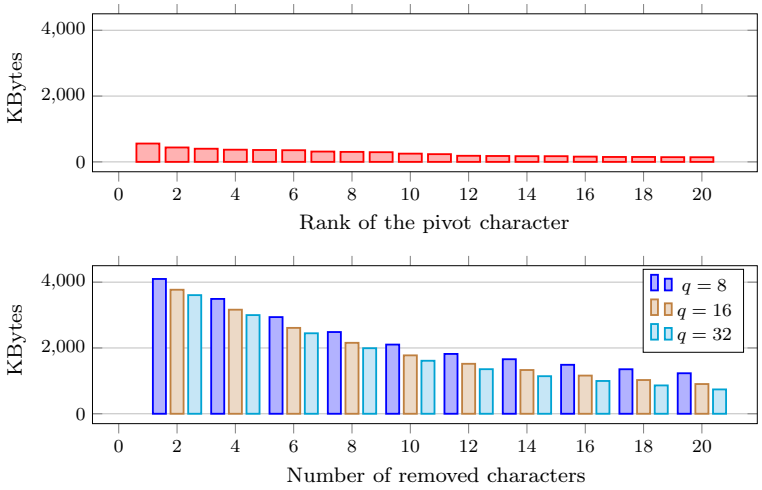

Fig. 8 Space consumption of the text sampled algorithms. All values are in KB. On the top: memory space required by the new algorithm, for different pivot characters with rank ranging from 1 to 20. On the bottom: memory space required by the OTS algorithm, for different sets of removed characters, with size ranging from 2 to 20, and for different values of the parameter q

obtained by the new algorithm ranges (at least) from 24 to 80%, even with the best performance of the OTS algorithm.

Observe also that in the case of short texts (with a size of few Mega Bytes), the space consumption of our algorithm is easily comparable with space requirements of a standard online string matching algorithm. For instance, if we suppose to search for a pattern of 30 characters on a text of 3 MB, the Boyer-Moore-Horspool algorithm [7] requires only 256 Bytes (less than 0.1% of the text size), the BDM algorithm [13] requires approximately 8 KB (0.2% of text size), while more efficient solutions as the Berry-Ravidran and the WFR algorithms [5] require approximately 65 KB (2.1% of text size). However, the space requirements of an online string matching algorithm does not depend on the size of the text, thus when the value of n increases the gap may become considerable.

4.2 Preprocessing and Searching Times

In this section we compare the two approach for the text-sampling string matching problem, in terms of preprocessing and searching times. With the term *preprocessing time* we refer to the time needed to construct the sampled text which will be used during the searching phase. We do not take into account in this measurements the preprocessing time needed by the online exact string matching algorithm used during the searching phase. We will refer to *searching time* as the time needed to perform searching of the pattern on both sampled and original texts, including any preprocessing of the underlying searching algorithm.

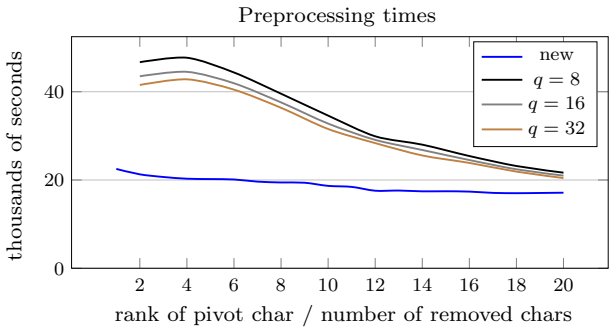

Fig. 9 Preprocessing times of the text sampled algorithms. Running times are expressed in thousands of sec. The x axis represents the rank of the pivot character in the case of the new algorithm, while represents the number of removed characters in the case of previous algorithms

Fugure 9 shows the preprocessing times of our algorithm together with those of the Ots algorithms (in the latter case we show preprocessing time for the three different values of $q \in \{8, 16, 32\}$). It turns out that the new algorithm has always a faster preprocessing time, with a speed-up which goes from 50, to 15%, depending on the pivot element and on the number of removed characters. This is mainly correlated with size of the data structure constructed during the preprocessing phase and discussed in the previous section.

Figures 10 and 11 show the searching times of the text sampling algorithms in the case of short patterns ($2 \leq m \leq 16$) and long patterns ($32 \leq m \leq 256$), respectively. The dashed red line represents the running time of the original Horspool algorithm. Running times (in the y axis) are represented in thousands of seconds. The x axis represents the rank of the pivot character in the case of the new algorithm, while represents the number of removed characters in the case of the Ots algorithms.

In the case of short patterns best results are obtained by the new sampled approach selecting a small-rank pivot character. If compared with the original Horspool algorithm, the speed up obtained by the new approach goes from 32% (for $m = 2$) to 64% (for m = 16), while the gain obtained in comparison with the Ots algorithm decreases as the length of the pattern increases and going from 13%, for $m = 2$, to 7.7%, in the case of $m = 16$.

In the case of long patterns the difference between the running times of the two algorithms is negligible. However, if compared with the original Horspool algorithm, the speed up is much more evident and goes from 66% (for $m = 32$) to 91% (for $m = 256$).

5 Conclusions and Future Works

We presented a new approach to the sampled string matching problem based on alphabet reduction and characters distance sampling. In our solution we divide the text in blocks of size k and sample positions of such characters inside the blocks. During the searching phase the sampled data is used to filter candidate occurrences

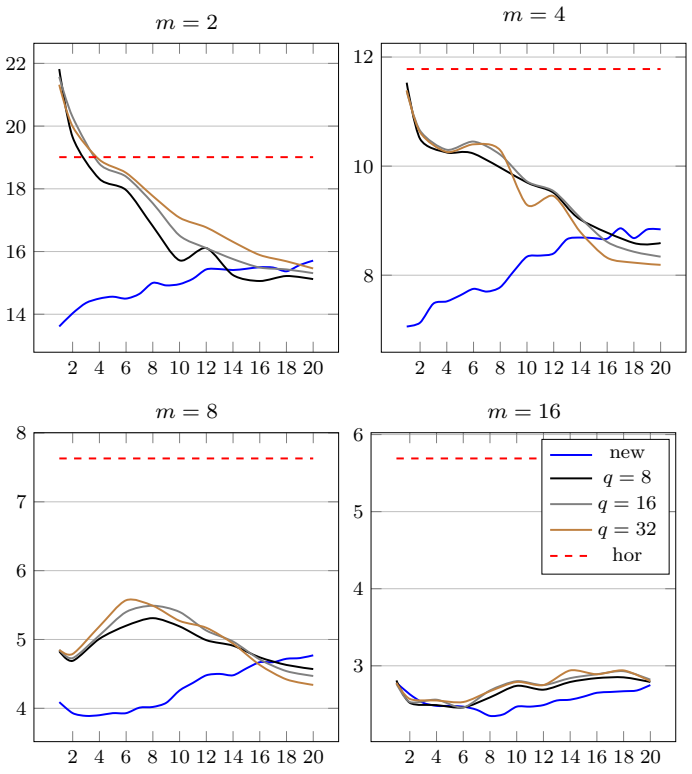

Fig. 10 Running times of the text sampling algorithms in the case of small patterns ($2 \leq m \leq 16$). The dashed red line represent the running time of the original Horspool algorithm. Running times (in the y axis) are represented in thousands of seconds. The x axis represents the rank of the pivot character in the case of the new algorithm, while represents the number of removed characters in the case of previous algorithms

of the pattern. All the occurrences which are found during this first step are then verified for a whole match in the original text.

Our algorithm is faster than previous solutions in the case of short patterns and may require only 5% of additional extra space. Despite this good performance we also proved that, when the underlying algorithm used for searching the sampled text for the sampled pattern achieves optimal worst-case and average-case time complexities, then also our algorithm attains the same optimal complexities, at least for patterns with a length of (at most) few hundreds of characters.

We applied our solution only to the case of a natural language text with a rather wide alphabet since the current approach doesn't work efficiently for small alphabets. It turns out indeed that the number of false positives located during the filtering phase increases as the size of the alphabet decreases. Thus it should be interesting to find a non-trivial strategy to extend this kind of solution also to the case of sequences over a small alphabet, like genome or protein sequences. We also wonder whether indexed solutions, as those based on the suffix tree, the suffix array and the

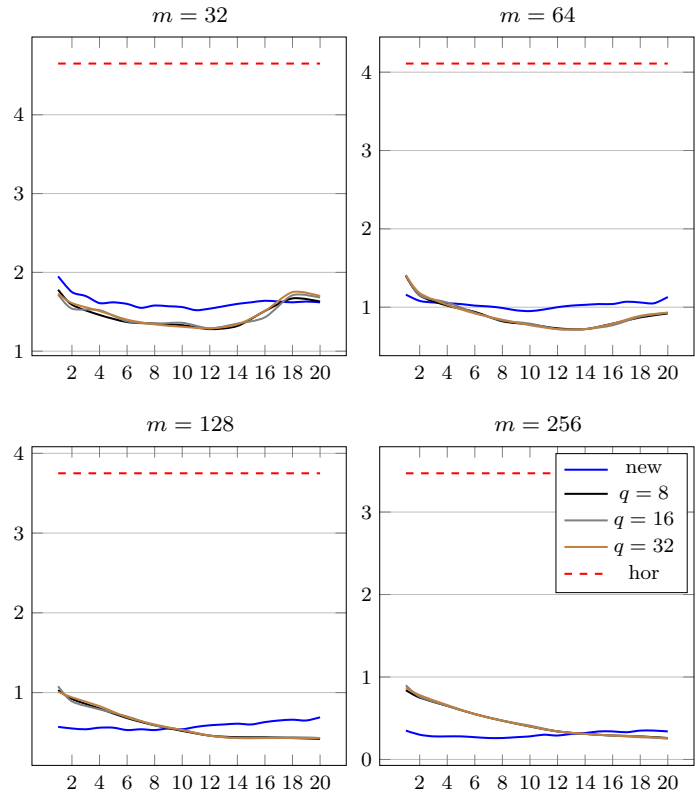

Fig. 11 Running times of the text sampling algorithms in the case of long patterns ($32 \leq m \leq 256$). The dashed red line represent the running time of the original Horspool algorithm. Running times (in the y axis) are represented in thousands of seconds. The x axis represents the rank of the pivot character in the case of the new algorithm, while represents the number of removed characters in the case of the Ots algorithms

FM-index of the text, should take advantage of the new text sampling approach. We intend to go in such directions in our future works.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley, London (1974)
2. Apostolico, A.: The myriad virtues of suffix trees. In: Apostolico, A., Galil, Z. (eds.) Combinatorial Algorithms on Words. NATO Advanced Science Institutes, Series F, vol. 12, pp. 85–96. Springer, Berlin (1985)
3. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Commun. ACM **20**(10), 762–772 (1977)
4. Cantone, D., Faro, S., Giaquinta, E.: Adapting Boyer-Moore-like algorithms for searching Huffman encoded texts. Int. J. Found. Comput. Sci. **23**(2), 343–356 (2012)
5. Cantone, D., Faro, S., Pavone, A.: Speeding up string matching by weak factor recognition. Stringology **2017**, 42–50 (2017)

6. Claude, F., Navarro, G., Peltola, H., Salmela, L., Tarhio, J.: String matching with alphabet sampling. J. Discrete Algorithms **11**, 37–50 (2012)
7. Crochemore, M., Czumaj, A., Gasieniec, L., Jarominek, S., Lecroq, T., Plandowski, W., Rytter, W.: Speeding up two string-matching algorithms. Algorithmica **12**(4), 247–267 (1994)
8. Faro, S., Lecroq, T.: The exact online string matching problem: a review of the most recent results. ACM Comput. Surv. **45**(2), 13 (2013)
9. Faro, S., Lecroq, T., Borzì, S., Di Mauro, S., Maggio, A.: The String Matching Algorithms Research Tool. In Procedings of Stringology, pp. 99–111, (2016)
10. Ferragina, P., Manzini, G.: Indexing compressed text. J. ACM **52**(4), 552–581 (2005)
11. Fredriksson, K., Grabowski, S.: A general compression algorithm that supports fast searching. Inf. Process. Lett. **100**(6), 226–232 (2006)
12. Grabowski, S., Raniszewski, M.: Sampling the suffix array with minimizers. In: Porceedings of String Processing and Information Retrieval (SPIRE 2015), Lecture Notes in Computer Science, vol 9309, Springer, pp. 287–298 (2015)
13. Horspool, R.N.: Practical fast searching in strings. Softw. Pract. Exp. **10**(6), 501–506 (1980)
14. Karkkainen, J., Ukkonen, E.: Sparse suffix trees. In: Proceedings of 2nd Annual International Conference on Computing and Combinatorics (COCOON), LNCS 1090, pp. 219–230 (1996)
15. Klein, S.T., Shapira, D.: A new compression method for compressed matching. In: Data Compression Conference, IEEE. pp. 400–409 (2000)
16. Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comput. **6**(2), 323–350 (1977)
17. Manber: A text compression scheme that allows fast searching directly in the compressed file. ACM Trans. Inf. Syst. **15**(2), 124–136 (1997)
18. Manber, U., Myers, G.: Suffix arrays: a new method for online string searches. SIAM J. Comput. **22**(5), 935–948 (1993)
19. Moura, E., Navarro, G., Ziviani, N., Baeza-Yates, R.: Fast and flexible word searching on compressed text. ACM Trans. Inf. Syst. **18**(2), 113–139 (2000)
20. Navarro, G., Tarhio, J.: LZgrep: a Boyer-Moore string matching tool for Ziv-Lempel compressed text. Softw. Pract. Exp. **35**, 1107–1130 (2005)
21. Shibata, Y., Kida, T., Fukamachi, S., Takeda, M., Shinohara, A., Shinohara, T., Arikawa, S.: Speeding Up Pattern Matching by Text Compression. In: CIAC 306–315 (2000)
22. Yao, A.C.: The complexity of pattern matching for a random string. SIAM J. Comput. **8**(3), 368–387 (1979)