This manuscript is the AAM version of the DOI 10.1016/j.tcs.2022.12.034, published in Theoretical Computer Science Volume 946, 10 February 2023, 113684

Improved Characters Distance Sampling for Online and Offline Text Searching[1]

Simone Faro[a], Francesco Pio Marino[a], Arianna Pavone[b]

[a]*Department of Mathematics and Computer Science, University of Catania, Viale A.Doria n.6, Catania, 95125, Italy*
[b]*Department of Cognitive Science, University of Messina, Via Concezione n.6, Messina, 98121, Italy*

Abstract

Sampled string matching is a very effective technique to reduce the search time for a pattern within a text at the cost of a small amount of additional memory, used for storing a partial index of the text. This approach has recently received some interest and has been applied to improve both online and offline string matching solutions, improving standard solutions by more than 50%. However, this improvement is currently only achievable in the case of texts on large-sized alphabets, and remains small (or absent) in the case of small-sized alphabets. In this article we propose an extension of the approach to text-sampling, known as Character Distance Sampling, to the case of small alphabets, obtaining an improvement of up to 98% compared to standard solutions in the case of online string matching. We also extend this approach to the case of offline string matching, introducing a sampled version of the suffix array, obtaining performances up to 5 times higher than the search obtained on the standard suffix array. Differently from what has been done by previous solutions, our idea is not based on the reduction of the number of indexed suffixes, but on the construction of the index directly on the sampled text.

Keywords: Text processing, experimental algorithms, string matching

[1]This paper is based on preliminary results appeared in Proceedings of the 22nd Italian Conference on Theoretical Computer Science (ICTCS 2021) [13] and in Proceedings of the 25nd Prague Stringology Conference (PSC 2021) [11]

1. Introduction

Exact 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 in a large text y where characters of both sequences are drawn from an alphabet Σ.

It is a fundamental problem in computer science with applications in many other fields, like natural language processing and information retrieval. It is also a critical problem in computational molecular biology and plays a very important role in biological sequences analysis, mainly due to the constantly growing amount of molecular data extracted from living organisms. For this reason sequence matching techniques play a very important role in various applications in computational biology for data analysis. In addition, as the size of data increases, the space required to store this data and the data structures useful for solving the problem is also constantly increasing, which is why it is necessary to adopt new efficient approaches that can drastically reduce the space used while preserving the effectiveness of the search.

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 input sequence *online*, when searching. Differently, solutions based on the second approach try 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*. *Sampled string matching* is a technique that has recently received interest and which is halfway between the two just described solutions, allowing both to be improved. Its goal is to significantly cut down the space requirements of indexed matching, on the one hand, and drastically reduce searching time for the online solutions, on the other hand. However sampled string matching allows so far to search efficiently only in the case of natural language texts or, in general, when searching on input sequences over large alphabets, while its performances degrade when the size of the underlying alphabets decreases.

In this paper, we present an extension of a previous approach [12], called Character Distance Sampling (CDS), in two separate directions.

- We extend CDS to small alphabets obtaining a more feasible solution in the case of biological data like genome or protein sequences and, in general, in the case of small alphabets. Our proposed approach

makes use of condensed characters in order to enlarge the size of the underlying alphabet and, as a result, speed up the searching process and reduce the space consumption of the resulting sampled text.

- We adapt the proposed sampling approach also in the case of the offline search by developing a sampled variant of the suffix array based on the CDS approach. Differently from what has been done by previous solutions, our idea is not based on the reduction of the number of indexed suffixes, but on the construction of the index directly on the sampled text.

From our experimental results it turns out that the use of condensed alphabets leads to reduce the space consumption up to 80% and to speed up the online searching process up to 98%, significantly improving the results obtained by the previous text sampling approach. We also conducted experimental tests for offline search, obtaining, also in this case, significant improvements. Specifically, the new approach presented in this paper allows a reduction of search times up to 5 times, compared to the times obtained using the standard data structure.

The paper is organized as follows. First we briefly review previous solutions related with our work making appropriate references to the most relevant literature. In Section 3 we briefly review the Characters Distance Sampling approach introduced by Faro *et al.* [12] and extend it to condensed alphabets. Then, in Section 4, we show how to apply our sampling approach to the online string matching case, while its application to the case of offline string matching is presented in Section 5. Finally, in Section 6, we present experimental results and draw our conclusions in Section 7.

2. Related Results

Formally, the *exact string matching* problem consists in finding all the occurrences of a given pattern x, of length m, in a large text y, of length n, where characters of both sequences are drawn from an alphabet Σ of size σ.

Online string matching solutions assume that the text is not preprocessed and thus they need to scan the input sequence *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 [24], while the optimal average time complexity of the problem is $\Theta(n \log_\sigma(m)/m)$ [38], achieved for example by the Backward-Dawg-Matching (BDM) algorithm [6].

Many string matching solutions have been also developed in order to obtain sub-linear performance in practice [9]. Among them the Boyer-Moore-Horspool (BMH) algorithm [2, 20] deserves a special mention, since it 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 $O(\sigma^2)$ [9]. However their searching time is always proportional to the length of the text and thus their performances may stay poor in many practical cases, especially for huge texts and short patterns.[2]

Differently, solutions based on *indexed searching* try 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. The literature in this research area is truly extensive and citing all the solutions proposed over the years would go beyond the scope of this paper. However, among the most efficient solutions to such problem we mention those based on suffix trees [9], which find all occurrences in $O(m+occ)$-worst case time, those based on suffix arrays [2], which solve the problem in $O(m + \log n + occ)$ [2], where occ is the number of occurrences of x in y, and those based on the FM-index [15] (Full-text index in Minute space), which is a compressed full-text substring index based on the Burrows-Wheeler Transform (BWT) allowing compression of the input text while still permitting fast substring queries. However, despite their optimal time performance[3], space requirements of full-index data structures, as suffix-trees and suffix-arrays, are from 4 to 20 times the size of the text.

While the size of a compressed indexes, as the FM-Index [15], is typically less than the size of the text, it turns out that their space requirement is too large for many practical applications.

An alternative solution to full indexes 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 compressions schemes, as Ziv-Lempel [33] and Huffman [3], the best practical be-

[2]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) [4].

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

haviour are achieved by ad-hoc schemes designed for allowing fast searching [29, 7, 23, 36, 16]. 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 a high time for each reported occurrence.

One of the most interesting solutions to the problem are *compact data structures*. Such structures are equipped with native tools for handling text directly in its compressed form [31]. In general, however, they are not able to compress text by orders of magnitude, offering only complex functionality in the space required by raw data.

When working on repetitive texts the BWT, featuring long runs of equal consecutive symbols, has enormous potential in terms of compression [32] and compact data structures benefit from this feature. These include, for example, a complete index for pan-genomics read alignment using prefix-free parsing [26]. A relevant compressibility measure for a repetitive text is indeed the number r of runs in their BWT. Based on this measure, the Run-Length FM-index [28] is able to efficiently count the number of occurrences of a pattern using $O(r)$ space and in log-logarithmic time per pattern symbol. Although it can be also extended [17, 18] in order to be able to locate the positions of such occurrences without using additional space, such data structures are designed to be efficient only in the case of very repetitive texts.

2.1. Sampled String Matching

An alternative solution to the problem is *sampled string matching*, introduced in 1991 by Vishkin [37], which consists in the construction of a succinct sampled version of the text (which must be maintained together with the original text) and in the application of any online string matching algorithm directly on the sampled sequence.

Although any candidate occurrence of the pattern may be found more efficiently, the drawback of this approach is that any occurrence reported in the sampled-text requires to be verified in the original text. Apart from this point a sampled-text approach may have a lot of good features: it may be easy to implement if compared with other succint matching approaches, it may require very small extra space and may allow fast searching. Additionally it may also allow fast updates of the data structure.

Apart the theoretical result of Vishkin, the first practical solution to sampled string matching has been introduced by Claude *et al.* [5] and is based on an alphabet reduction. In this paper we refer to this algorithm as Occurrence Text Sampling (OTS). Specifically, if we 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 the OTS 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 $1 \leq i \leq \hat{n}$. However, since $\hat{y}$ contains partial information, a table ρ is maintained in order to map, at regular intervals, positions of the sampled text to their corresponding positions in y.

It turns out that the OTS approach leads to solutions which are to be up to 5 times faster than standard online string matching and 2 times faster than standard offline string matching on English texts. Such results are obtained with an extra space requirement which is only 14% of text size.[4]

More recently Faro *et al.* presented a more effective sampling approach based on *character distance sampling* (CDS) [12, 11], obtaining in practice a speed up by a factor of up to 9 on English texts, using limited additional space whose amount goes from 11% to 2.8% of the text size, with a gain in searching time up to 50% if compared against the OTS approach. We will describe in more detail the ideas on which the CDS approach is based in the next section, in which we will extend their application to search for texts on smaller alphabets using a technique based on condensed alphabets. In fact, it should be emphasized that both OTS and CDS approaches to exact string matching prove to work efficiently only in the case of natural language texts or, in general, when searching on input sequences over large alphabets, while their performances degrade when the size of the underlying alphabets decreases.

[4]We also notice that some partial improvements have been thereafter presented by Grabowsky and Raniszewski [19]. They proposed a more convenient indexing suffix sampling approach, with only a minimum pattern length as a requirement. Their experiments show that the resulting solution achieves competitive time-space tradeoffs on most standard benchmark data.

3. Character Distance Sampling and Condensed Alphabets

In this section we briefly present the sampling approach known as Character Distance Sampling (CDS) and extend it to the case of condensed alphabets.

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. 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 size of x.

We elect a set $C \subseteq \Sigma$ to be the *set of pivot characters*. Given this set of pivot characters we sample the text y by taking into account the distances between consecutive positions of any pivot characters $c \in C$ in y. More formally our sampling approach is based on the following definition of *position sampling* of a text.

Definition 1 (Position Sampling)**.** *Let y be 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 $c \in C$ in the input text y.*

First we define the position function*, $\delta : \{1, .., n_C\} \rightarrow \{1, .., n\}$, where $\delta(i)$ is the position of the i-th occurrence of any character of C in y. Formally we have*

(i)	$1 \leq \delta(i) < \delta(i+1) \leq n$	*for each* $1 \leq i \leq n_C - 1$
(ii)	$y[\delta(i)] \in C$	*for each* $1 \leq i \leq n_C$
(iii)	$y[\delta(i)+1..\delta(i+1)-1]$ *contains no* $c \in C$	*for each* $0 \leq i \leq n_C$

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

Then the position sampled version of y, indicated by $\dot{y}$, is a numeric sequence, of length n_C, defined as

$$\dot{y} = \langle \delta(1), \delta(2), .., \delta(n_C) \rangle. \tag{1}$$

Example 1. *Suppose $y =$ "agaacgcagtata" is a* DNA *sequence of length 13, over the alphabet $\Sigma =$ {a,c,g,t}. Let $C =$ {a} be the set of pivot characters. Thus the position sampled version of y is $\dot{y} = \langle 1, 3, 4, 8, 11, 13 \rangle$. Specifically the first occurrence of character $a \in C$ is at position 1 ($y[1] = a$), its second occurrence is at position 3 ($y[3] = a$), and so on.*

Example 2. *As in Example 1, assume $y =$ "agaacgcagtata" is a* DNA *sequence of length 13, over the alphabet $\Sigma =$ {a,c,g,t}. Let $C =$ {a,c} be the*

set of pivot characters (now C contains two characters). Thus the position sampled version of y is $\dot{y} = \langle 1, 3, 4, 5, 7, 8, 11, 13\rangle$. Note that in this example we simply added the two positions, 5 and 7, where the character c occurs.

Definition 2 (Characters Distance Sampling)**.** *Let $C \subseteq \Sigma$ be the set of pivot characters, let $n_C \leq n$ be the number of occurrences of any 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 any pivot character in y.*

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

$$\begin{aligned} \bar{y} &= \langle \Delta(1), \Delta(2), .., \Delta(n_C - 1)\rangle \\ &= \langle \delta(2) - \delta(1), .., \delta(n_C) - \delta(n_C - 1)\rangle \end{aligned} \tag{2}$$

Plainly we have

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

Example 3. *Let $y =$ "agaacgcagtata" be a text of length 13, over the alphabet $\Sigma = \{a,c,g,t\}$. Let $C = \{a\}$ be the set of pivot characters. Thus the character distance sampling version of y is $\bar{y} = \langle 2, 1, 4, 3, 2\rangle$. Specifically $\bar{y}[1] = \Delta(1) = \delta(2) - \delta(1) = 3 - 1 = 2$, while $\bar{y}[3] = \Delta(3) = \delta(4) - \delta(3) = 8 - 4 = 4$, and so on.*

Definition 3 (Rank of a character)**.** *Let x be a pattern of length m, and let $c \in \Sigma$. We define $\phi : \Sigma \to \{0..m\}$ as the function which associates any character of the text with the number of its occurrences in x. The rank of the character c is the position of c in the alphabet Σ, if we assume that all characters are sorted by their $\phi(c)$ values in non increasing order. If two characters of the alphabet have the same number of occurrences, then their relative order is irrelevant. To avoid confusion, we assume that they are arranged in lexicographic order. More formally the rank of c is given by the cardinality of the set $\{k \in \Sigma \mid \phi(k) > \phi(c)$ or $(\phi(k) = \phi(c)$ and $k > c)\} + 1$*

Example 4. *Let again $y =$ "agaacgcagtata" be a text of length 13, over the alphabet $\Sigma = \{a,c,g,t\}$. The values associated by the function ϕ to the four characters of the alphabet are $\phi(a) = 6$, $\phi(c) = 2$, $\phi(g) = 3$ and $\phi(t) = 2$, respectively. Thus the character a has rank 1, while t has rank 4. The characters c and g have rank 2 and 3, respectively.*

It is important to notice that the description of the CDS approach presented in this paper is slightly simplified compared to that introduced in [12]. Specifically, in the original approach described in [12], use was made of the *k-bounded position function*, $\delta_k : \{1, .., n_c\} \rightarrow \{0, .., k-1\}$, where k is a given threshold constant and $\delta_k(i)$ is the position (modulus k), i.e.$\delta_k(i) = [\delta(i) \mod k]$, for each $i = 1, ..., n_c$. Then then *k-bounded-position sampled version* of y, indicated by $\dot{y}$, is a numeric sequence, of length n_c defined as $\dot{y} = \langle \delta_k(1), \delta_k(2), .., \delta_k(n_c) \rangle$. Plainly we have $0 \leq \dot{y}[i] < k$, for each $1 \leq i \leq n_c$.

Although this allows to store each element of the sampled version of the text using only $\log(k)$ bits, 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 need to 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. Then, if 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 we have $\tau[i] = \max(\{j : \delta(j) \leq ik\} \cup \{0\})$, for $1 \leq i \leq \lceil n/k \rceil$. 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, since the values in the block mapping τ are stored in a non decreasing order, i.e. $\tau[i] \leq \tau[i+1]$, $\forall\ 0 \leq i \leq \lceil n/k \rceil$, it follows that $\delta(j) = (\tau[b] - 1)k + \dot{y}[j]$, where $b = \min\{i : \tau[i] \geq j\}$.

In practical cases, the choice of $k = 256$ would allow the sampled text to be stored using n_C bytes, plus the additional space of $4n/k$ bytes to maintain the block-mapping table.

In this paper, we simplified the representation of the CDS approach by dropping the τ table and keeping only the sampled text by storing each position with 4 bytes. On the one hand, this makes the approach more efficient from a practical point of view by avoiding the computation of the $\delta(i)$ value at each step of the algorithm. On the other hand, this increases the space required for storing the partial index. However, this disadvantage is largely mitigated by the extension of the approach to condensed alphabets, presented in the next section.

3.1. Extension to Condensed Alphabets

Let y be an input string, of length n, over an alphabet Σ of size σ. Given a constant parameter q, with $1 \leq q < n$, we define the condensed alphabet $\Sigma_y^{(q)}$, related to y, as

$$\{c \in \Sigma^q \mid c = y[i..i+q-1] \text{ for some } 1 \leq i \leq n-q+1\}$$

Roughly speaking $\Sigma_y^{(q)}$ is the set of all different substrings of length q (or q-grams) appearing in y. We define the q-condensed version of y as follows.

Definition 4 (q-Condensed Sequence)**.** *Let y be a text of length n over an alphabet Σ of size σ and let $\Sigma_y^{(q)}$ be the condensed alphabet, related to y, for a given constant parameter q. We define the q-condensed version of the sequence y as the sequence, of length $n-q+1$, of all consecutive (and overlapping) substrings of length q appearing in y. More formally*

$$y^{(q)} = \langle y[1..q], y[2..q+1], y[3..q+2], .., y[n-q+1..n] \rangle$$

Example 5. *Assume $y =$ "agtagcgcagt" is a* DNA *sequence of length 11, over the alphabet $\Sigma = \{a,c,g,t\}$. Then we have*

$$\begin{aligned} y^{(2)} &= \langle ag, gt, ta, ag, gc, cg, gc, ca, ag, gt \rangle \\ y^{(3)} &= \langle agt, gta, tag, agc, gcg, cgc, gca, cag, agt \rangle \\ y^{(4)} &= \langle agta, gtag, tagc, agcg, gcgc, cgca, gcag, cagt \rangle \end{aligned}$$

Definition 5 (q-Characters Distance Sampling)**.** *Let $C \subseteq \Sigma_y^{(q)}$ be the set of pivot characters, let $n_C \leq n$ be the number of occurrences of any pivot character in the text $y^{(q)}$ and let δ be the position function of $y^{(q)}$. We define the* q-characters distance function *$\Delta^{(q)}$ as the distance between two consecutive occurrences of any pivot character in $y^{(q)}$, where $\Delta^{(q)}(i)$, for $1 \leq i \leq n_C - 1$, is the distance between the $(i+1)$-th and the i-th occurrence of any occurrences of any pivot character in $y^{(q)}$.*

Then the q-characters-distance sampled version *of the sequence y is a numeric sequence of length $n_C - 1$, indicated by $\bar{y}^{(q)}$ and defined as*

$$\bar{y}^{(q)} = \langle \Delta^{(q)}(1), \Delta^{(q)}(2), .., \Delta^{(q)}(n_C - 1) \rangle. \tag{3}$$

Example 6. *As in the previous Example 5 assume $y =$ "agtagcgcagtagta" is a* DNA *sequence of length 15, over the alphabet $\Sigma = \{a,c,g,t\}$. If we suppose $q = 2$ and $C = \{$"ag"$\}$ is the set of pivot characters, then we have*

$$\begin{aligned} \dot{y}^{(2)} &= \langle 1, 4, 9, 12 \rangle \\ \bar{y}^{(2)} &= \langle 3, 5, 3 \rangle. \end{aligned}$$

Similarly, if we suppose $q = 3$ and $C = \{$ "agt" $\}$ is the set of pivot characters, then we have

$$\begin{aligned} \dot{y}^{(3)} &= \langle 1, 9, 12 \rangle \\ \bar{y}^{(3)} &= \langle 8, 3 \rangle. \end{aligned}$$

In this paper we do not go into the way for a correct selection of the set of pivot characters, and even we leave the details of an analysis about what is the best subset to be chosen. However in our experimental evaluation (see Section 6) we will show how it is enough to put a single character in the set of pivot characters. Such a character is selected on the basis of its *rank value*, where we remember that the rank of a character c corresponds to its position in the alphabet Σ when we assume that all characters are sorted by their frequencies inside the text (see Definition 3).

As we will note later, even the choice of the most frequent character of the alphabet (be it a single character or a q-gram) is enough to obtain a good efficacy, both in terms of search time and in terms of space used for storing the partial index. In the following two sections we will show how to apply sampling techniques in the two main scenarios, that of online searching and that of offline searching.

4. Online Sampled String Matching

In this section we show how the sampled text-based approach can be adopted to solve the online string matching problem. Specifically, in a first phase, we briefly present how the OTS approach is used in this scenario. Next we present the solution for online string matching based on the CDS approach, also based on a condensed alphabet. Such algorithms make use of an auxiliary string matching algorithm, used for searching the sampled pattern on the sampled text, and they work well with most of the known string matching algorithms. However, since the sampled patterns tend to be short, we assume that the search phase is implemented using the Horspool algorithm, which has been found to be fast in such setting. Such assumption is the same as that adopted in the paper by Claude *et al.* [5]

4.1. Online Searching Using the OTS Approach

Claude *et al.* [5] presented a very efficient algorithm for online string matching based on their OTS approach. Specifically, 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. In addition let $\hat{\Sigma} \subset \Sigma$ be the sampled alphabet, and let $\hat{y}$ be the sampled text of length $\hat{n}$, containing all (and only) the characters of the sampled alphabet.

The OTS algorithm constructs a sampled version of the input pattern, $\hat{x}$, of length $\hat{m}$ during the searching phase. Such pattern is then 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 it uses information maintained in the table ρ to map positions of the sampled text to their corresponding positions in the original text. The position mapping ρ has size $\lfloor \hat{n}/h \rfloor$, where h is the *interval factor*, and is such that $\rho[i] = j$ if character $y[j]$ corresponds to character $\hat{y}[h \times i]$. More formally we have, for $1 \leq i \leq \lfloor \hat{n}/h \rfloor$

$$\rho[i] = j, \text{ if } y[j] \text{ is the } (h \times i)\text{-th occurrence in } y \text{ of any character of } \hat{\Sigma}$$

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 h 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 $1 \leq i \leq \lfloor \hat{n}/h \rfloor$, 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 $\rho[r/h] + (r \mod h) - \alpha + 1$ to position $\rho[r/h + 1] - (h - (r \mod h)) - \alpha + 1$, 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. $\bar{m} = 0$, the algorithm merely reduces to search for x in the original text y.

Example 7. *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 $h = 2$ 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 $y[\rho(2) + 3 - 1..\rho(3)]$, finding a match at position 10.*

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.[5] are removed from the original alphabet. 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.

4.2. Online Searching Using the CDS Approach

Let y be an input text of length n over an alphabet Σ of size σ, let $q > 1$ and let $\Sigma_y^{(q)}$ be the condensed alphabet over Σ. In addition let $C \subseteq \Sigma^{(q)}$ be the set of pivot characters.

During the preprocessing phase the algorithm performs a scanning of the text y and builds the corresponding position sampled text $\dot{y}^{(q)}$.

Let now x be an input pattern of length m and let m_C be the number of occurrences of any pivot character in $x^{(q)}$. 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. The idea behind such searching procedures is to take advantage of the sampled text $\dot{y}^{(q)}$ computed during the preprocessing phase in order to quickly locate any candidate substring s of the original text which may include an occurrence of the pattern.

If such candidate substring s has length m then the algorithm simply performs a character-by-character comparison between the pattern and the substring. Otherwise if the candidate substring s has length greater than m, then a searching procedure is called, based on a standard exact online string matching algorithm, for searching the pattern x in s.

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.

Case 1: $m_C = 0$

If the pattern contains no occurrence of any pivot characters, we have that m_C is equal to 0. Under this assumption the algorithm searches for the pattern x in all substrings of the original text which do not contain the pivot characters. Specifically such substrings are identified in the original text by the intervals $[\delta^{(q)}(i) + 1..\delta^{(q)}(i+1) + q - 2]$, for each $0 \leq i \leq n_C$, assuming $\delta^{(q)}(0) = 0$ and $\delta^{(q)}(n_C + 1) = n - q + 2$.

[5] 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 13 most frequent characters

```
SEARCH-0(x, ẏ^(q), y, q)
1.  m ← len(x)
2.  n_C ← len(ẏ)
3.  ẏ^(q)[0] ← 0
3.  ẏ^(q)[n_C + 1] ← n − q + 2
4.  for i ← 1 to n_C + 1 do
5.      if (ẏ^(q)[i] − ẏ^(q)[i − 1] + q − 2 ≥ m) then
6.          l ← ẏ^(q)[i − 1] + 1
7.          r ← ẏ^(q)[i] + q − 2
8.          search for x in y[l..r]
```

Figure 1: The pseudocode of procedure SEARCH-0 for the sampled string matching problem, when no pivot character occurs in the input pattern x.

Specifically, for each $1 \leq i \leq n_C + 1$, the algorithm checks if the value $\dot{y}^{(q)}(i) - \dot{y}^{(q)}(i-1) + q - 2$ is greater or equal to m. In such a case the algorithm searches for x in the substring of the text $y[\dot{y}^{(q)}[i-1] + 1..\dot{y}^{(q)}[i] + q - 2]$ using any standard string matching algorithm. Otherwise the substring is skipped, since no occurrence of the pattern could be found at such position. The pseudocode of procedure SEARCH-0 for the sampled string matching problem, when no pivot character occurs in the input pattern x, is depicted in Figure 1.

Case 2: $m_C = 1$

If the pattern x contains a single occurrence of any character of the set C, then the length of the sampled version of the pattern is still equal to 0. However also in this case the algorithm is able to efficiently take advantage of the information precomputed in $\dot{y}^{(q)}$ using the positions of the pivot character in $y^{(q)}$ as an anchor to locate all candidate occurrences of x.

Specifically, let α be the unique position in x which contains the pivot character, i.e. we assume that $x[\alpha..\alpha+q-1] = c$ and that both $x[1..\alpha-1]$ and $x[\alpha+1..m]$ do not contain any pivot character. Then, for each $0 \leq i \leq n_C - 1$, the algorithm checks if the value $\dot{y}^{(q)}(i-1) - \dot{y}^{(q)}(i-2)$ is greater than $\alpha - 1$ and if the value $\dot{y}^{(q)}(i) - \dot{y}^{(q)}(i-1)$ is greater than $m - \alpha$. In such a case the algorithm merely checks if the substring of the text $y[\dot{y}^{(q)}[i-1] - \alpha + 1..\dot{y}^{(q)}[i-1] - \alpha + m]$ is equal to the pattern. Otherwise the substring is skipped. As before we assume that $\dot{y}(0) = 0$ and $\dot{y}(n_C + 1) = n + 1$. The last

```
SEARCH-1(x, ẏ^(q), y, q)
1.   m ← len(x)
2.   n_C ← len(ẏ)
3.   α ← min{i : x^(q)[i] ∈ C}
4.   ẏ^(q)[0] ← 0
5.   ẏ^(q)[n_C + 1] ← n − q + 2
6.   for i ← 1 to n_C + 1 do
7.        if (ẏ^(q)[i − 1] − ẏ^(q)[i − 2] > α − 1 and
                ẏ^(q)[i] − ẏ^(q)[i − 1] > m − α) then
8.             l ← ẏ^(q)[i − 1] − α + 1
9.             r ← ẏ^(q)[i − 1] − α + m
10.            compare x and y[l..r]
```

Figure 2: The pseudocode of procedure SEARCH-1 for the sampled string matching problem, when the pattern x contains a single occurrence of the pivot character.

alignment of the pattern in the text is verified separately at the end of the main cycle. The pseudocode of procedure SEARCH-1 for the sampled string matching problem, when the pattern x contains a single occurrence of the pivot character, is depicted in Figure 2.

Case 3: $m_C \geq 2$

If the number of occurrences of any pivot character in C is greater than 1 then the algorithm uses the sampled text $\dot{y}^{(q)}$ to compute on the fly the sampled version $\bar{y}^{(q)}$ of $y^{(q)}$ and use it to search for any occurrence of $\bar{x}^{(q)}$. This is used as a filtering phase for locating in y any candidate occurrence.

First the character distance sampled version $\bar{x}$ of x is computed. Then the algorithm searches for $\bar{x}$ in $\bar{y}$ using any exact online string matching algorithm. Notice that $\bar{y}$ can be efficiently retrieved online from the sampled text $\dot{y}$, using relation given in (2).

For each candidate occurrence i of $\bar{x}$ located 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. For this purpose the algorithm checks if the substring of the text $y[\dot{y}^{(q)}[i]-\dot{y}^{(q)}[0]..\dot{y}^{(q)}[i]+m-1]$ is equal to x, where $\dot{y}^{(q)}[0]$ is the position of the first occurrence of the pivot character into the pattern. The pseudocode of procedure SEARCH-2^+ for the sampled string matching problem, when the pattern x contains at least 2 occurrences of the pivot character, is depicted

SEARCH-$2^+(x,\ \dot{y}^{(q)},\ y,\ q)$
1. $m \leftarrow len(x)$
2. $(\bar{x}^{(q)}, \bar{m}) \leftarrow$ COMPUTE-DISTANCE-SAMPLING(x, m, C)
3. search for $\bar{x}^{(q)}$ in $\bar{y}^{(q)}$:
4. for each i such that $\bar{x}^{(q)} = \bar{y}^{(q)}[i..i + \bar{m} - 1]$ do
5. $l \leftarrow \dot{y}^{(q)}[i] - \dot{y}^{(q)}[0]$
6. $r \leftarrow \dot{y}^{(q)}[i] + m - 1$
7. compare x and $y[l..r]$

Figure 3: The pseudocode of procedure SEARCH-2^+ for the sampled string matching problem, when the pattern x contains at least 2 occurrences of the pivot character.

in Figure 3.

4.3. Complexity Issues

In this section we prove that, assuming an underlying auxiliary string matching algorithm with a linear worst case and a $O(n \log(m)/m))$ average case time complexity, the resulting algorithm based on Character Distance Sampling over condensed alphabets achieves an optimal $O(n)$ time complexity in the worst-case and a $O(n \log(m)/m))$ time complexity in the average case. The following lemmas prove that procedures SEARCH-0, SEARCH-1 and SEARCH-2^+, respectively, achieve, under suitable conditions, optimal time complexity in both worst and average cases.

Lemma 1. *Le x and y be two strings of size m and n, respectively, over an alphabet Σ of size $\sigma > 1$. Let $C \subseteq \Sigma^{(q)}$ be set of pivot characters and let $\dot{y}^{(q)}$ be 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* SEARCH-0 *are $O(n)$ and $O(n \log_\sigma m/m)$, respectively.*

Proof. In our argumentation we refer to the pseudo-code reported in Fig.1. In order to evaluate the worst-case time complexity of SEARCH-0, we can notice that each substring of the text is scanned at least once in line 8, 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

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

where, the $O(n)$ term is related to the pre-processing of the pattern, while each $O(\Delta^{(q)}(i))$ term corresponds to the time required to process the substring $y^q[\delta(i)..\delta(i+1)]$.

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

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

□

Lemma 2. *Le x and y be two strings of size m and n, respectively, over an alphabet Σ of size $\sigma > 1$. Let $C \subseteq \Sigma^{(q)}$ be the set of pivot characters and let $\dot{y}^{(q)}$ be the 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 our argumentation we refer to the pseudo-code reported in Fig.2. In order to evaluate the worst-case time complexity of the procedure, notice that each character could be involved in, at most, two consecutive checks in line 10. Specifically any text position in the interval $[\delta(i-1)+1..\delta(i)-1]$ could be involved in the verification of the substrings $y[\delta(i-1)-\alpha+1..\delta(i-1)+m-\alpha]$ and $y[\delta(i)-\alpha+1..\delta(i)+m-\alpha]$. 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 procedure, notice that the expected number of occurrences in y^q of the set of pivot characters 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

$$\begin{aligned} T^1_{\mathsf{avg}}(n) = \; & \mathbf{E}(n_c) \cdot \mathbf{E}(\text{insp}) = \\ & O\left(\tfrac{n}{\sigma}\right) \cdot O\left(\tfrac{\sigma}{\sigma-1}\right) = \\ & O\left(\tfrac{n}{\sigma-1}\right) \end{aligned}$$

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 $k \geq \sigma$. □

Lemma 3. *Le x and y be two strings of size m and n, respectively, over an alphabet Σ of size $\sigma > 1$. Let $C \subseteq \Sigma^{(q)}$ be the set of pivot characters and let $\dot{y}^{(q)}$ be the 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 our argumentation we refer to the pseudo-code reported in Fig.3. 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 $O(n_c + n_x m)$, 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 it is not difficult to suppose to implement procedure Verify based on a linear algorithm, as KMP, in order to remember all positions of the text which have been already verified, allowing the algorithm to run in overall $T^{2+}_{\mathsf{wst}}(n) = O(n)$ 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 $O(n_c \log m_c/m_c)$. 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

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

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

5. Offline Sampled String Matching

In this section we describe an approach to indexed searching which makes use of a suffix array constructed over the sampled version of the text. In our evaluation we have chosen to use the suffix array [30] as a reference point since it can be counted among the most efficient standard solutions to the offline

string matching problem and because this solution was previously adopted by Claude *et al.* [5] for comparison with their approach to text sampling.

We report that the suffix array data structure can be improved in various ways. For instance it can be effectively compressed with relative Lempel-Ziv (RLZ) dictionary compression, in such a way that arbitrary sub-arrays can be rapidly decompressed, thus facilitating compressed indexing [34, 35]. Among other possibilities we also report that a Suffix Array can be enhanced by representing a sequence of integers using Fibonacci encodings, thereby reducing the space requirements while retaining the searching functionalities [1]. In addition a Suffix Array can be improved in efficiency in various ways [25]. However, these improvements are beyond the scope of this work, which intends to verify how a sampled text-based approach can improve the search efficiency of offline approaches, although it is possible to imagine that different data structures can achieve different improvements.

We remember that a suffix array is a sorted array of all suffixes of a string. Such data structure has been introduced by Manber and Myers in 1990 [30] as a simple, space efficient alternative to suffix trees [9]. It has been extensively studied in the last three decades and in 2016 Li, Li and Huo [27] gave the first in-place $O(n)$-time construction algorithm that is optimal both in time and space, where in-place means that the algorithm only needs $O(1)$ additional space beyond the input string and the output suffix array.

Formally, given a text y of length n, the suffix array s_y of y is defined to be an array of integers providing the starting positions of suffixes of y in lexicographical order. This means that $s_y[i]$ contains the starting positions of the i-th smallest suffix in y and thus for all $1 \leq i \leq n$, we have $y[s_y[i-1]..n] < y[s_y[i]..n]$.

Example 8. *Let $y =$ "agaacgcagtata" be a text of length 13, over the alphabet $\Sigma = \{a,c,g,t\}$. The suffix array, s_y, contains the starting positions of all suffixes of y, arranged in lexicographical order. Specifically we have:*

$s_y[1] = 12 \rightarrow \langle a \rangle$
$s_y[2] = 2 \rightarrow \langle aacgcagtata \rangle$
$s_y[3] = 3 \rightarrow \langle acgcagtata \rangle$
$s_y[4] = 0 \rightarrow \langle agaacgcagtata \rangle$
$s_y[5] = 7 \rightarrow \langle agtata \rangle$
$s_y[6] = 10 \rightarrow \langle ata \rangle$
$s_y[7] = 6 \rightarrow \langle cagtata \rangle$
$s_y[8] = 4 \rightarrow \langle cgcagtata \rangle$
$s_y[9] = 1 \rightarrow \langle gaacgcagtata \rangle$
$s_y[10] = 5 \rightarrow \langle gcagtata \rangle$
$s_y[11] = 8 \rightarrow \langle gtata \rangle$
$s_y[12] = 11 \rightarrow \langle ta \rangle$
$s_y[13] = 9 \rightarrow \langle tata \rangle$

The time complexity needed to build suffix array is $O(n^2 \log(n))$ if an $O(n \log(n))$ algorithm is used for sorting the array of all suffixes. However, there are many efficient algorithms to build suffix array [22]. Once the suffix array is built, it is possible to search a pattern using the suffix array by a binary search in $O(n \log(n))$ time. However it has been proved that we can report all *occ* occurrences of a pattern in a text in $O(m + \log(n) + occ)$ [2]. In the following two subsections we show how the suffix array based solution can be adapted to the OTS and CDS approaches presented in this paper.

5.1. Offline Searching Using the OTS Approach

To turn the sampling approach into an index, Claude *et al.* use a suffix array to index the sampled positions of the text. When constructing the suffix array, only suffixes starting with a sampled character will be considered, but the sorting will still be done considering the full suffixes. The resulting sampled suffix array is like the suffix array of the original text where suffixes starting with unsampled characters have been omitted.

Example 9. *Let again y = "agaacgcagtata" be a text of length 13, over the alphabet* $\Sigma = \{a,c,g,t\}$*. The sampled suffix array,* $s_{\hat{y}}$*, contains the starting positions of all suffixes of y beginning with a character with the sampled alphabet, arranged in lexicographical order. Specifically, assuming that* $\hat{\Sigma} = \{a, g\}$*, we have:*

$$
\begin{aligned}
s_{\hat{y}}[1] &= 12 \rightarrow \langle a \rangle \\
s_{\hat{y}}[2] &= 2 \rightarrow \langle aacgcagtata \rangle \\
s_{\hat{y}}[3] &= 3 \rightarrow \langle acgcagtata \rangle \\
s_{\hat{y}}[4] &= 0 \rightarrow \langle agaacgcagtata \rangle \\
s_{\hat{y}}[5] &= 7 \rightarrow \langle agtata \rangle \\
s_{\hat{y}}[6] &= 10 \rightarrow \langle ata \rangle \\
s_{\hat{y}}[7] &= 1 \rightarrow \langle gaacgcagtata \rangle \\
s_{\hat{y}}[8] &= 5 \rightarrow \langle gcagtata \rangle \\
s_{\hat{y}}[9] &= 8 \rightarrow \langle gtata \rangle
\end{aligned}
$$

Search on the sampled suffix array is carried out as follows. Given a pattern x the algorithm finds the first sampled character of the pattern. Assume such character is at index j of x. The pattern is then partitioned into the prefix $x[1..j-1]$ and the suffix starting with the first sampled character $x[j..m]$. The algorithm then searches the sampled suffix array for the suffix of the pattern like in an ordinary suffix array. Each candidate occurrence located by this search will then be verified by comparing the prefix $x[1..j-1]$ against the text. Observe that the OTS suffix array can be used for searching a text only for patterns that contain at least one sampled character.

5.2. Offline Searching Using the CDS Approach

The algorithm we propose is divided into two phases: a first *preprocessing phase* which consists in the construction of a sampled version, $s_{\bar{y}}$, of the suffix array and a *searching phase* which is used to search any pattern x of length m in y making use of the suffix array $s_{\bar{y}}$ and the sampled text $\dot{y}$. We notice that, as it happens in any offline string matching solution, the preprocessing phase is performed only once for the construction of the partial index, while the searching phase can be run for an indeterminate number of queries. We notice also that the algorithm must maintain the original text y, the sampled version of the text $\dot{y}$ and the corresponding suffix array $s_{\bar{y}}$.

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

As before, let y be an input text of length n over an alphabet Σ of size σ and let $C \subseteq \Sigma$ be the set of pivot characters. During the preprocessing phase the algorithm builds and stores the position sampled text $\dot{y}$ of y. This requires $O(n)$-time and $O(n_c)$-space, where n_c is the number of occurrences

of any pivot character in y. Subsequently a suffix array of $\bar{y}$ is constructed on the fly using information maintained in $\dot{y}$.

As a consequence, when constructing the suffix array of $\bar{y}$, the algorithm takes into account only suffixes beginning with a pivot character in the original text, drastically reducing the space requirement for maintaining the whole index.

Definition 6 (CDS Suffix Array)**.** *Let y be 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 $c \in C$ in the input text y. Le $\delta : \{1, .., n_C\} \to \{1, .., n\}$ be the position function and let $\dot{y}$ be the position sampled version of y.*

The CDS suffix array $s_{\bar{y}}$ of y is defined to be an array of all index positions i, with $1 \leq i \leq n_C - 1$ such that $\bar{y}[s_{\bar{y}}[i-1]..n_C - 1] < \bar{y}[s_{\bar{y}}[i]..n_C - 1]$.

Example 10. *Let $y =$ "agaacgcagtata" be a text of length 13, over the alphabet $\Sigma = \{a,c,g,t\}$. Let $C = \{a\}$ be the set of pivot characters. Thus the position sampled version of y is $\dot{y} = \langle 1, 3, 4, 8, 11, 13\rangle$, while the character distance sampled version of y is $\bar{y} = \langle 2, 1, 4, 3, 2\rangle$.*

The sampled suffix array, $s_{\bar{y}}$, contains all positions in y, starting with a pivot character $c \in C$, arranged in lexicographical order with respect to the suffixes of $\bar{y}$. Specifically we have:

$$
\begin{aligned}
s_{\bar{y}}[1] &= 1 \quad \to \langle 1, 4, 3, 2\rangle \\
s_{\bar{y}}[2] &= 4 \quad \to \langle 2\rangle \\
s_{\bar{y}}[3] &= 0 \quad \to \langle 2, 1, 4, 3, 2\rangle \\
s_{\bar{y}}[4] &= 3 \quad \to \langle 3, 2\rangle \\
s_{\bar{y}}[5] &= 2 \quad \to \langle 4, 3, 2\rangle
\end{aligned}
$$

Thus, $s_{\bar{y}}[0] = 1$ indicates that the smallest suffix, in lexicographical order relative to $\bar{y}$, begins at position $\dot{y}[1]$ in y (it is $\bar{y}[1..4] = \langle 1, 4, 3, 2\rangle$). Similarly $s_{\bar{y}}[1] = 4$ indicates that the second suffix begins at position $\dot{y}[4]$ in y.

During the searching phase the algorithm uses the suffix array of the sampled text $s_{\bar{y}}$ as an index to quickly locate every occurrences of a sampled pattern $\bar{x}$ in $\bar{y}$. Each of these occurrences is treated as a candidate occurrence of x in y, and as such it will be verified by a comparison procedure.

The searching algorithm works as a standard search on a suffix array. It is based of the fact that finding every occurrence of the pattern $\bar{x}$ is equivalent

to find every suffix in $\bar{y}$ that begins with the $\bar{x}$. Thanks to the lexicographical ordering of the suffix array, all such suffixes are grouped together and can be found efficiently with a single binary search, which locates the starting position of the interval.

For the sake of completeness we observe that both the OTS and the CDS suffix arrays resemble a sparse suffix array [21], which indexes regularly sampled text positions. However, such data structure only need to make one search of the sampled pattern, while using a sparse suffix array h searches are needed if the suffix array indexes every h-th position. The drawback of such data structures is that they can only be used for patterns that contain at least one sampled character, whereas the sparse suffix array can be used if the pattern length is at least q. The variance of the search time when using the sampled suffix array is also larger than when using a sparse suffix array because in the sampled suffix array we have much less control over the length of the string that is used in the suffix array search.

5.3. Complexity Issues

In order to to compute the time complexity needed for searching a pattern x, of length m, in a text y, of length n, we assume that n_C is the number pivot characters appearing in y. Then finding the first position of a sampled pattern $\bar{x}$ of length m_C in a suffix array $s_{\bar{y}}$ of length n_C takes $O(m_C \log n_C)$-time [2] while finding the set of all ρ occurrences of $\bar{x}$ in $\bar{y}$ takes $O(\rho)$-time. Since each occurrence must be verified in the original text we need $O(m\rho)$ additional time for the verification phase. The overall time complexity of the searching algorithm is then $O(m_C \log(n_C) + m\rho)$.

It is important to notice that such complexity leads to a worst-case scenario that cannot be compared with the $O(m \log(n))$-time complexity obtained by suffix arrays. Indeed, the verification phase component $O(m\rho)$, which is not required in standard suffix arrays, may in many cases be dominant over the search phase compontent $O(m_C \log(n_C))$.

If we assume, for instance that $y = (abc)^n$ and $x = (abc)^{m-1}(acb)^m$ and the set of pivot characters is $C = \{a\}$, the sampled suffix array reports all text positions as candidate occurrences and the verification needs to be run n time. This scenario leads to an overall worst-case time complexity equal to $O(nm)$ overall.

6. Experimental Results

In this section we present experimental results obtained by comparing the new proposed sampling approaches (using values of q ranging between 2 and 4) against the standard Character Distance Sampling (CDS) approach (obtained with q set to 1). Experimental evaluations were conducted in both the online and offline scenarios.

We compare our approaches against the Occurrence Text Sampling (OTS) approach and against the standard solution for which no text sampling is included.[6] In order to conduct a comparison as fair as possible we also implemented the OTS approach using q-grams, for values of q ranging between 2 and 4

All algorithms have been implemented using the `C` programming language, and have been tested using a variant of the SMART[7] tool [10] properly tuned for testing string matching algorithms based on a text-sampling. Tests have been executed on a MacBook Pro with 4 Cores, a 2.7 GHz Intel Core i7 processor, 16 GB RAM 2133 MHz LPDDR3, 256 KB of L2 Cache and 8 MB of Cache L3. The code of the algorithms used to perform the experiments presented in this section is available online at `https://www.dmi.unict.it/faro/SAMPLING/`.

All algorithms, for both the online and offline scenarios, have been tested on two 100MB text buffers containing a real biological sequence and a natural language text in the English language. Specifically the biological sequence is a collection of newline-separated gene DNA sequences (without descriptions, just the bare DNA code) obtained from files `01hgp10` to `21hgp10`, plus `0xhgp10` and `0yhgp10`, from Gutenberg Project. Each of the 4 bases is coded as an uppercase letter `A`, `G`, `C`, `T`, with few occurrences of other special characters. The natural language text buffer is the concatenation of English text files selected from `etext02` to `etext05` collections of Gutenberg Project, where the headers related to the project have been deleted so as to leave just the real text. Both sequences are available for download in the PIZZA&CHILI Corpus (`http://pizzachili.dcc.uchile.cl`).

In the experimental evaluation (for both online and offline searching),

[6] The standard solutions taken as a reference point are the Boyer-Moore-Horspool algorithm for the online scenario and the search on suffix arrays for the offline scenario.

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

patterns of length m were randomly extracted from the sequences, with m ranging over the set of values $\{2^i | 3 \leq i \leq 8\}$. For each value of m, the mean over the running times (expressed in hundredths of seconds) of 1000 runs has been reported.

In our implementations we selected the pivot character on the basis of its *rank value*, where we remember that the rank of a character c is the position of c in the alphabet Σ, if we assume that all characters are sorted by their frequencies inside the text (see Definition 3).

Then we evaluated the behaviour of our algorithms for different values of the rank r of the selected pivot character and specifically for r ranging between 1 (the most frequent character) and 16. Observe that if σ is the size of the original alphabet Σ, then σ^q is the size of the condensed alphabet $\Sigma^{(q)}$. As a consequence, in the case of experimental tests on genome sequences and $q = 1$, the value of the rank r is limited in the range between 1 and 4, since 4 is the size of the alphabet. We underline also that, in the case of the OTS approach, the value of the rank r refers to variations of the size of the set of sampled characters. Also in this case r ranges from 2 to 16.

6.1. 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 original input sequences, is required by a given solution to solve the problem.

Text-sampling algorithms require to store the whole text together with the additional sampled-text which is used to speed-up the searching phase. Although sampled texts have the good property to allow a direct access to the input text (when they are scanned sequentially), to be of any practical interest they should require as little extra space as possible.

Fig. 4 and Fig. 5 show the space consumption of the newly proposed text-sampling approaches, in the case of a genome sequence and an English text, respectively. Data are reported for different values of q in terms of percentage of memory used, in comparison with the original text size. We recall that, in the case of CDS, memory space consumption is plotted on variations of the rank of the pivot character. As expected, the function which describes memory requirements shows a decreasing trend while the rank of the pivot character increases. Similarly space consumption drastically decreases when the size of q increases.

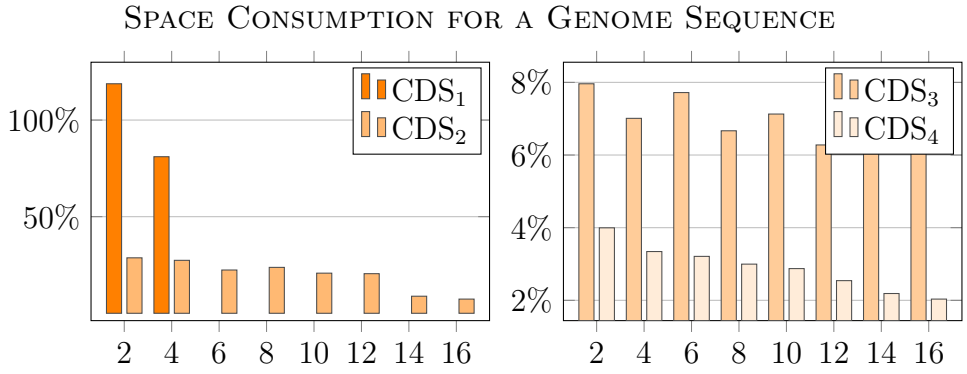

Figure 4: Space consumption of CDS approaches on a genome sequence, for different pivot characters with rank ranging from 2 to 16 and for different values of the parameter q, ranging from 1 to 4. Data are reported in terms of percentage of memory used relative to the original text size.

Data reported in Fig. 4, related to the sampling of a genome sequence, show that, when compared against the standard sampling approach (obtained with $q = 1$), the benefit in space consumption obtained by the approaches based on condensed alphabets is impressive. Specifically the gain for CDS ranges from 72% (for $r = 1$ and $q = 2$) to 95% (for $r = 16$ and $q = 4$). In addition we can observe a sensible gain in the space consumption also in comparison with the OTS algorithms implemented using condensed alphabets.

Data reported in Fig. 5, related to the English text, show that, when compared against the standard sampling approach (obtained with $q = 1$), the benefit in space consumption obtained by the approaches based on condensed alphabets is even more advantageous. Specifically the gain for CDS ranges from 90% (for $r = 1$ and $q = 2$) to 98.8% (for $r = 16$ and $q = 4$). Also in this case, the CDS approach shows a significant reduction in the space consumption when compared with the OTS approach implemented using condensed alphabets.

For the sake of completeness we would like to point out that standard algorithms for the online string matching problem require an amount of space which is, in general, proportional to the length of the pattern and/or to the size of the alphabet. In this particular case (a 5MB text buffer) the Boyer-Moore-Horspool algorithm requires only 1.24 KB of memory for implementing the occurrence heuristic (equivalent to a $O(\sigma)$-space complexity), while

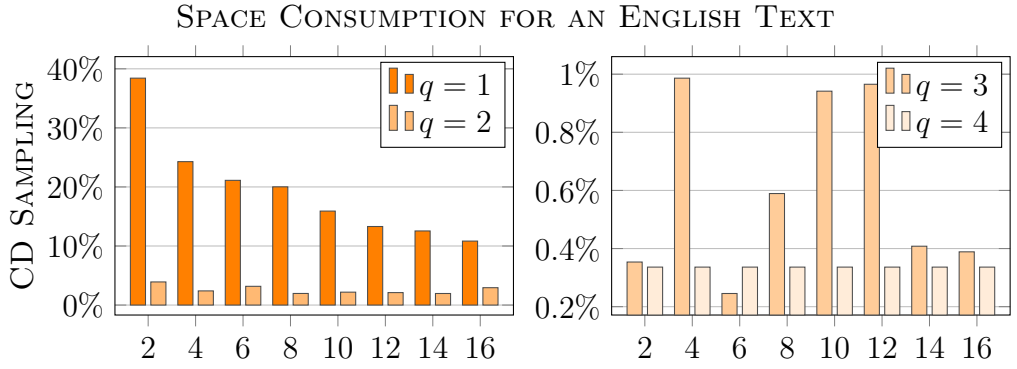

Figure 5: Space consumption of CDS approaches on an English text, for different pivot characters with rank ranging from 2 to 16 and for different values of the parameter q, ranging from 1 to 4. Data are reported in terms of percentage of memory used relative to the original text size.

some among the most effective algorithms (for instance WFRq [4], SKIPq [8]) are implemented by means of a hash table of size 65536, requiring 0.2 MB of additional space. Thus it turns out that, under particular conditions (texts of moderate lengths), the practical space requirements of our proposed sampling algorithms are comparable with those of standard online string matching solutions.

6.2. Online Searching

In this section we compare the different text-sampling approaches in terms of online searching times. In this context we refer to the *searching time* as the time needed to perform the searching of the pattern on both sampled and original texts, including any preprocessing of the underlying algorithm. However, in our analysis the searching time doesn't include the preprocessing time needed to construct the partial index.

Following the same lines of previous papers on sampled string matching [5, 12] we tested all sampling solutions in combination with the Boyer-Moore-Horspool (HOR) algorithm [20] for the implementation of the underlying standard searching procedure. As a consequence, in our comparison we also included the Boyer-Moore-Horspool string matching algorithm (in its standard implementation) in order to understand how much the proposed sampling approach contributes to speed-up a standard online string matching

solution.[8]

In our experimental results we also included the best running time obtained by OTS solutions implemented with condensed alphabets. Specifically if $\mathrm{OTS}_{(q,r)}$ is the searching speed of the OTS algorithm implemented using q-grams and rank r, we compute $\mathrm{OTS}_{(q,r)}$ as

$$\mathrm{OTS}_{(q,r)} = \max_{\substack{1 \le q \le 4 \\ 1 \le r \le 16}} \left(\mathrm{OTS}_{(q,r)} \right)$$

Observe that the original OTS approach is obtained by setting $q = 1$. Our experiments showed that the values that yield the best results are $q = 3$ and $r = 12$ for searching genomic sequences, and $q = 4$ and $r = 12$ for searching English texts. Thus we have reported only these values in the graphs.

In addition, for the sake of completeness, we also included in our experimental results three among the most efficient algorithm recently introduced for the exact online string matching problem. Specifically we included the Weak-Factor-Recognition algorithm [4] (WRF), the Brackward-Range-Automaton-Matcher [14] (BRAM) and the Skip-Search algorithm [8]. All algorithms have been implemented in several variants using q-grams, for values of q ranging from 1 to 8. Here again, we have only reported the results obtained with the best variant for each algorithm.

Fig. 6 and Fig. 7 show the resulting searching times of all tested algorithms when they were used for searching on a genome sequence and on an English text, respectively. Results are expressed in terms of searching speed, reported in Gigabytes per second (GB/s).

In general, the search speeds achieved by the new variants are extremely high and their advantage over the performance of standard algorithms is impressive. This is due to the fact that the main loop of the search phase iterates over the sampled text, the length of which, as we have seen above, is much shorter than that of the original text. Added to this is the fact that the number of candidate occurrences is very low, especially for long patterns.

From experimental results on a genome sequence (Fig. 6) it turns out that in all cases the best results are obtained by the variants based on condensed

[8] Although there exists many other searching algorithms able to show better practical performances on biological data (see for instance [4, 8]) this kind of comparison goes beyond the objectives of this paper. We expect that the proposed approach is able to enhance the performances of different string matching algorithms with different, though similar, rates.

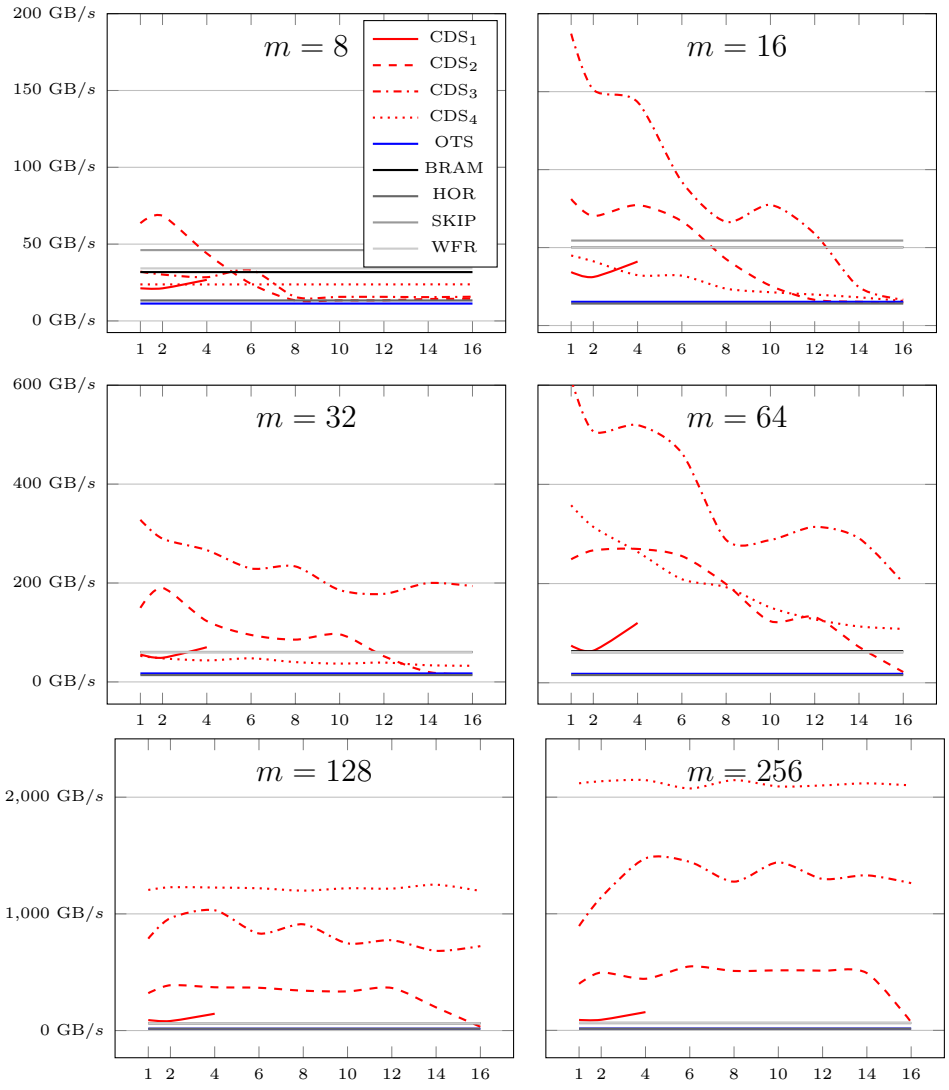

Figure 6: Searching speed on a genome sequence. Red lines represent the CDS_q algorithm implemented with $1 \leq q \leq 4$, the solid gray-tones lines represent the standard algorithms while the blue solid line represents the best searching time of the OTS solution implemented with q-grams. The x axis represents the rank r of the pivot character in the case of the sampling algorithms, with $1 \leq r \leq 16$.

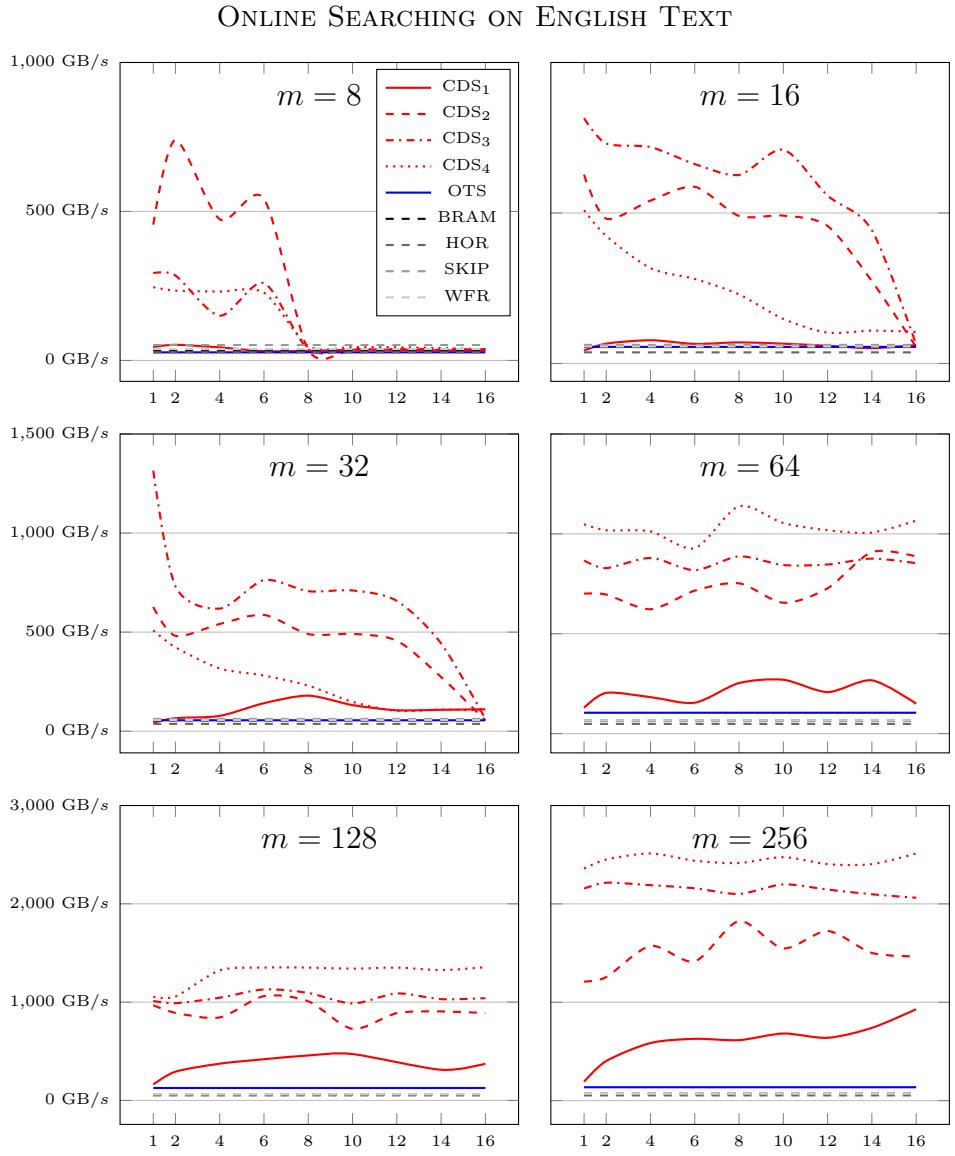

Figure 7: Searching speed on a natural language text. Red lines represent the CDS_q algorithm implemented with $1 \leq q \leq 4$, the solid gray-tones lines represent the standard algorithms while the blue solid line represents the best searching time of the OTS solution implemented with q-grams. The x axis represents the rank r of the pivot character in the case of the sampling algorithms, with $1 \leq r \leq 16$.

alphabets. Specifically, in the case of short patterns ($m = 8$) the best running times are obtained by the variant with $q = 2$, but as the length of the pattern increases, the variants with higher q values prove to be faster and faster until the variant with $q = 4$ proves to be the fastest from $m \geq 128$.

As might be expected, search speeds also increase as the pattern length increases, from speeds of just under 100 GB/s for $m = 8$ to speeds of just over 2.000 GB/s for $m = 256$.

When using a value of q greater than 1, the speed up obtained by CDS is always greater than 50% and reaches the value of 90% under suitable conditions, i.e. for $q = 4$ and long patterns.

Observe that the behaviour of algorithms based in CDS follow a decreasing trend for increasing rank values. Thus in most cases the better choice is to use the most frequent element as the pivot character. Observe indeed that, when the rank of the pivot character is greater than a given threshold, the performances of the CDS algorithms based on q-grams sensibly degrades. Specifically this threshold is approximately equal to 6 for short patterns ($q \leq 3$ and $m = 8$), while it increases as the pattern gets longer or for greater values of q.

Similarly, from experimental results on an English text (Fig. 7) it turns out that the variants based on condensed alphabets obtain the best results in all cases. Again, the best choice for short patterns is $q = 2$, but as the length of the pattern increases, the variants with higher q values prove to be faster and faster until the variant with $q = 4$ proves to be the fastest from $m >= 64$. The maximum speed reached by such solution is very close to 3.000 GB/s. When using a value of q greater than 1, the speed up obtained by CDS is always greater than 50% and reaches the value of 98% under suitable conditions, i.e. for $q = 4$ and long patterns.

We observe that on natural language texts and long patterns the behaviour of algorithms based in CDS follow a slightly increasing trend for increasing rank values, but, in general, the search speed obtained from the different values of the rank is comparable. As in the case of genome sequence, in the case of short patterns, when the rank of the pivot character is greater than a given threshold, the performances of the CDS algorithms based on q-grams sensibly degrades.

Going into details of the improvement in terms of running times we observe that the original CDS approach ($q = 1$) leads to improvements which are in percentage between 74% (in the case of short patterns) and 77% (in

the case of long patterns) if compared with the underlying standard string matching algorithm. The new CDS algorithms based on condensed alphabets give instead much more evident improvements which range from 96% (for short patterns) and 99.6% (in the case of long patterns) compared with the same algorithm. This improvements translate into a gain up to 70% for short patterns and up to 96% in the case of long patterns, if compared with OTS approach.

6.3. Offline Searching

In this section we compare the different text-sampling approaches in the offline scenario. In this context we refer to the *searching time* as the time needed to perform the searching of the pattern on the text-index. In our analysis the searching time doesn't include the preprocessing time needed to construct the index.

Following the same lines of [5] we tested all sampling solutions using a modified Suffix Array, as described in Section 5. Thus, in our comparison we also included the original Suffix Array algorithm (STD), in its standard implementation, in order to understand how much the proposed sampling approaches contribute to speed-up a standard offline searching solution.

We mention that Ferragina and Manzini [15] showed that it is possible to search a pattern x of length m backwards in the suffix array of y without storing it. A backward search means that we first search for the substring $x[m..m]$, then for the substring $x[m-1..m]$, and so on, until the whole pattern x is found. In the computer science literature, any data structure that allows to search a pattern x backwards in the (conceptual) suffix array of a text y is called an FM-index of y. In our experimental results, however, we have not used the backward-search technique, limiting ourselves to a simple binary search within the suffix array constructed on the text y.

Table. 1 and Table. 2 show the resulting performance of all tested algorithms when they were used for searching on a genome sequence and on an English text, respectively. Results are reported in terms of searching speed, expressed in number of queries per second (QR/s).

For an easier reading of the results, we have listed in the tables, for each algorithm, only the results of the variants with the best performance. Specifically for OTS we reported the results with $q = 4$ and $r = 12$, in the case of genomic sequences, and $q = 3$ and $r = 12$, in the case of natural language texts. In the case of CDS we reported the results for pairs of values $(q, r) \in \{(1, 2), (2, 8), (3, 10), (4, 8)\}$.

m	STD	$\text{OTS}_{(4,12)}$	$\text{CDS}_{(1,2)}$	$\text{CDS}_{(2,8)}$	$\text{CDS}_{(3,10)}$	$\text{CDS}_{(4,8)}$
8	152	161	188	191	**201**	198
16	154	172	183	197	213	**227**
32	161	183	201	217	216	**234**
64	163	199	218	234	241	**268**
128	168	202	233	247	246	**299**
256	181	216	257	281	293	**322**

Table 1: Offline searching on a genome sequence. Values are reported in thousands of queries per second. We used values q ranging from 1 to 4, values of r ranging from 1 to 16 and pattern lengths m ranging from 8 to 256. Best results have been bold-faced.

m	STD	$\text{OTS}_{(3,12)}$	$\text{CDS}_{(1,8)}$	$\text{CDS}_{(2,8)}$	$\text{CDS}_{(3,10)}$	$\text{CDS}_{(4,8)}$
8	198	212	244	244	247	**261**
16	212	224	257	254	**271**	267
32	219	222	271	278	284	**299**
64	218	238	321	338	337	**355**
128	219	243	322	343	356	**385**
256	224	278	366	368	382	**401**

Table 2: Offline searching on a natural language text. Values are reported in thousands of queries per second. We used values q ranging from 1 to 4, values of r ranging from 1 to 16 and pattern lengths m ranging from 8 to 256. Best results have been bold-faced.

From the experimental results it turns out that the standard solution based on a suffix array offers performances of about 200K queries per second, while the solutions based on the OTS approach oscillate between 200K and 300K queries per second, proposing a search speed mildly faster than the previous one. The solutions based on the CDS approach, on the other hand, offer significantly better performance, oscillating between 250K queries per second ($q = 1$ and short patterns) and 400K queries per second ($q = 4$ and for almost all values of m). These last solutions offer performances that are therefore between 1.5 and 1.8 times faster than the standard solution.

Finally, we note that, as might be expected, there is no significant variation in the results for different values of m. In fact, although there is a difference between the results, going from patterns of length 8 to patterns of length 256, the gain obtained is at most just over 50%. This is due to the fact that the search is only partially dependent, in its $O(m \log(n))$ time complexity, on the length m of the pattern while the dominant factor consists of the logarithm of the size n of the data structure.

7. Conclusions

In this paper we have presented an extensions of a text sampling approach, called Character Distance Sampling, to the case of texts over small alphabets and in the case of offline searching. The first extension was carried out using condensed alphabets in which consecutive groups of q characters are assimilated to a single element of the alphabet, significantly extending its size. The result obtained by this extension was to significantly lower the execution time in the search phase while keeping the space used by the index below the space used by the previous approaches. In our second extension we have proposed a suffix array model built directly on the sampled text in order to decrease the number of candidate occurrences and, consequently, the time required for the response to each single query. This approach contrasts with those currently proposed in the literature which are limited to reducing the number of suffixes taken into account in the original data structure. The experimental results proposed in our extensive experimentation, conducted both on the online and offline scenarios, show how this new proposal offers significantly better performances both in terms of space and in terms of search time. Our future studies will focus in this direction in order to apply sampled string matching to other problems related to text processing.

8. Acknowledgement

We gratefully acknowledge support from project STORAGE—Università degli Studi di Catania, Piano della Ricerca 2020/2022

References

[1] Ekaterina Benza, Shmuel T. Klein, and Dana Shapira. Smaller compressed suffix arrays†. *Comput. J.*, 64(5):721–730, 2021. doi:10.1093/comjnl/bxaa016.

[2] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977. doi:10.1145/359842.359859.

[3] Domenico Cantone, Simone Faro, and Emanuele Giaquinta. Adapting boyer-moore-like algorithms for searching huffman encoded texts. *Int. J. Found. Comput. Sci.*, 23(2):343–356, 2012. doi:10.1142/S0129054112400163.

[4] Domenico Cantone, Simone Faro, and Arianna Pavone. Linear and efficient string matching algorithms based on weak factor recognition. *ACM J. Exp. Algorithmics*, 24(1):1.8:1–1.8:20, 2019. doi:10.1145/3301295.

[5] Francisco Claude, Gonzalo Navarro, Hannu Peltola, Leena Salmela, and Jorma Tarhio. String matching with alphabet sampling. *J. Discrete Algorithms*, 11:37–50, 2012. doi:10.1016/j.jda.2010.09.004.

[6] Maxime Crochemore, Artur Czumaj, Leszek Gasieniec, Stefan Jarominek, Thierry Lecroq, Wojciech Plandowski, and Wojciech Rytter. Speeding up two string-matching algorithms. *Algorithmica*, 12(4/5):247–267, 1994. doi:10.1007/BF01185427.

[7] Edleno Silva de Moura, Gonzalo Navarro, Nivio Ziviani, and Ricardo A. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Trans. Inf. Syst.*, 18(2):113–139, 2000. doi:10.1145/348751.348754.

[8] Simone Faro. A very fast string matching algorithm based on condensed alphabets. In Riccardo Dondi, Guillaume Fertin, and Giancarlo Mauri, editors, *Algorithmic Aspects in Information and Management - 11th International Conference, AAIM 2016, Bergamo, Italy, July 18-20, 2016,*

Proceedings, volume 9778 of *Lecture Notes in Computer Science*, pages 65–76. Springer, 2016. doi:10.1007/978-3-319-41168-2_6.

[9] Simone Faro and Thierry Lecroq. The exact online string matching problem: A review of the most recent results. *ACM Comput. Surv.*, 45(2):13:1–13:42, 2013. doi:10.1145/2431211.2431212.

[10] Simone Faro, Thierry Lecroq, Stefano Borzi, Simone Di Mauro, and Alessandro Maggio. The string matching algorithms research tool. In Jan Holub and Jan Zdárek, editors, *Proceedings of the Prague Stringology Conference 2016, Prague, Czech Republic, August 29-31, 2016*, pages 99–111. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2016. URL: http://www.stringology.org/event/2016/p09.html.

[11] Simone Faro and Francesco Pio Marino. Reducing time and space in indexed string matching by characters distance text sampling. In Jan Holub and Jan Zdárek, editors, *Prague Stringology Conference 2020, Prague, Czech Republic, August 31 - September 2, 2020*, pages 148–159. Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2020. URL: http://www.stringology.org/event/2020/p13.html.

[12] Simone Faro, Francesco Pio Marino, and Arianna Pavone. Efficient online string matching based on characters distance text sampling. *Algorithmica*, 82(11):3390–3412, 2020. doi:10.1007/s00453-020-00732-4.

[13] Simone Faro, Francesco Pio Marino, and Arianna Pavone. Enhancing characters distance text sampling by condensed alphabets. In Claudio Sacerdoti Coen and Ivano Salvo, editors, *Proceedings of the 22nd Italian Conference on Theoretical Computer Science, Bologna, Italy, September 13-15, 2021*, volume 3072 of *CEUR Workshop Proceedings*, pages 1–15. CEUR-WS.org, 2021. URL: http://ceur-ws.org/Vol-3072/paper1.pdf.

[14] Simone Faro and Stefano Scafiti. The range automaton: An efficient approach to text-searching. In Thierry Lecroq and Svetlana Puzynina, editors, *Combinatorics on Words - 13th International Conference, WORDS 2021, Rouen, France, September 13-17, 2021, Proceedings*, volume 12847

of *Lecture Notes in Computer Science*, pages 91–103. Springer, 2021. doi:10.1007/978-3-030-85088-3_8.

[15] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.

[16] Kimmo Fredriksson and Szymon Grabowski. A general compression algorithm that supports fast searching. *Inf. Process. Lett.*, 100(6):226–232, 2006. doi:10.1016/j.ipl.2006.04.020.

[17] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in bwt-runs bounded space. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 1459–1477. SIAM, 2018. doi:10.1137/1.9781611975031.96.

[18] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in bwt-runs bounded space. *J. ACM*, 67(1):2:1–2:54, 2020. doi:10.1145/3375890.

[19] Szymon Grabowski and Marcin Raniszewski. Sampled suffix array with minimizers. *Softw. Pract. Exp.*, 47(11):1755–1771, 2017. doi:10.1002/spe.2481.

[20] R. Nigel Horspool. Practical fast searching in strings. *Softw. Pract. Exp.*, 10(6):501–506, 1980. doi:10.1002/spe.4380100608.

[21] Tomohiro I, Juha Kärkkäinen, and Dominik Kempa. Faster sparse suffix sorting. In Ernst W. Mayr and Natacha Portier, editors, *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014), STACS 2014, March 5-8, 2014, Lyon, France*, volume 25 of *LIPIcs*, pages 386–396. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2014. doi:10.4230/LIPIcs.STACS.2014.386.

[22] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955. Springer, 2003. doi:10.1007/3-540-45061-0_73.

[23] Shmuel T. Klein and Dana Shapira. A new compression method for compressed matching. In *Data Compression Conference, DCC 2000, Snowbird, Utah, USA, March 28-30, 2000*, pages 400–409. IEEE Computer Society, 2000. doi:10.1109/DCC.2000.838180.

[24] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977. doi:10.1137/0206024.

[25] Tomasz Marek Kowalski, Szymon Grabowski, and Kimmo Fredriksson. Suffix arrays with a twist. *Comput. Informatics*, 38(3):555–574, 2019. URL: http://www.cai.sk/ojs/index.php/cai/article/view/2019_3_555.

[26] Alan Kuhnle, Taher Mun, Christina Boucher, Travis Gagie, Ben Langmead, and Giovanni Manzini. Efficient construction of a complete index for pan-genomics read alignment. *J. Comput. Biol.*, 27(4):500–513, 2020. doi:10.1089/cmb.2019.0309.

[27] Zhize Li, Jian Li, and Hongwei Huo. Optimal in-place suffix sorting. In Travis Gagie, Alistair Moffat, Gonzalo Navarro, and Ernesto Cuadros-Vargas, editors, *String Processing and Information Retrieval - 25th International Symposium, SPIRE 2018, Lima, Peru, October 9-11, 2018, Proceedings*, volume 11147 of *Lecture Notes in Computer Science*, pages 268–284. Springer, 2018. doi:10.1007/978-3-030-00479-8_22.

[28] Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. In Alberto Apostolico, Maxime Crochemore, and Kunsoo Park, editors, *Combinatorial Pattern Matching*, pages 45–56, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[29] Udi Manber. A text compression scheme that allows fast searching directly in the compressed file. *ACM Trans. Inf. Syst.*, 15(2):124–136, 1997. doi:10.1145/248625.248639.

[30] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, oct 1993. doi:10.1137/0222058.

[31] Gonzalo Navarro. *Compact Data Structures - A practical approach*. 2016.

[32] Gonzalo Navarro. The compression power of the BWT: technical perspective. *Commun. ACM*, 65(6):90, 2022. doi:10.1145/3531443.

[33] Gonzalo Navarro and Jorma Tarhio. Lzgrep: a boyer-moore string matching tool for ziv-lempel compressed text. *Softw. Pract. Exp.*, 35(12):1107–1130, 2005. doi:10.1002/spe.663.

[34] Simon J. Puglisi and Bella Zhukova. Relative lempel-ziv compression of suffix arrays. In Christina Boucher and Sharma V. Thankachan, editors, *String Processing and Information Retrieval - 27th International Symposium, SPIRE 2020, Orlando, FL, USA, October 13-15, 2020, Proceedings*, volume 12303 of *Lecture Notes in Computer Science*, pages 89–96. Springer, 2020. doi:10.1007/978-3-030-59212-7_7.

[35] Simon J. Puglisi and Bella Zhukova. Smaller rlz-compressed suffix arrays. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *31st Data Compression Conference, DCC 2021, Snowbird, UT, USA, March 23-26, 2021*, pages 213–222. IEEE, 2021. doi:10.1109/DCC50243.2021.00029.

[36] Yusuke Shibata, Takuya Kida, Shuichi Fukamachi, Masayuki Takeda, Ayumi Shinohara, Takeshi Shinohara, and Setsuo Arikawa. Speeding up pattern matching by text compression. In Gian Carlo Bongiovanni, Giorgio Gambosi, and Rossella Petreschi, editors, *Algorithms and Complexity, 4th Italian Conference, CIAC 2000, Rome, Italy, March 2000, Proceedings*, volume 1767 of *Lecture Notes in Computer Science*, pages 306–315. Springer, 2000. doi:10.1007/3-540-46521-9_25.

[37] Uzi Vishkin. Deterministic sampling - A new technique for fast pattern matching. *SIAM J. Comput.*, 20(1):22–40, 1991. doi:10.1137/0220002.

[38] Andrew Chi-Chih Yao. The complexity of pattern matching for a random string. *SIAM J. Comput.*, 8(3):368–387, 1979. doi:10.1137/0208029.