Reducing Time and Space in Indexed String Matching by Characters Distance Text Sampling

Simone Faro and Francesco Pio Marino

Dipartimento di Matematica e Informatica,
Università di Catania, viale A.Doria n.6, 95125, Catania, Italia
faro@dmi.unict.it

Abstract. *Sampled string matching* is an efficient approach to the string matching problem which tries to overcome the prohibitive space requirements of indexed matching, on the one hand, and drastically reduce searching time for the online solutions, on the other hand. Sampled string matching dates back to 1991, however practical solutions to the problem only appeared more recently. They are able to speed up the online searching up to 9 times while using less than 5% of the text size. In this paper we take into account the problem of indexing sampled texts in order to reduce the space requirements of indexed matching and maintaining its very high practical and theoretical performances. Specifically we present a new efficient indexed string matching approach based on a characters distance sampling. The main idea is to sample the distances between consecutive occurrences of a given set of *pivot characters* and then to create a suffix array of the sampled text. From our experimental results it turns out that the newly presented approach is able to obtain a searching time gain up to 91%, when compared to the standard indexed approach, while using less than 15% of the space needed for the standard suffix array.

Keywords: string matching, ext processing, efficient searching, text indexing

1 Introduction

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. In literature such problem is called *String Matching*, and formally consists in finding all 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 remain the main form to store information. This is particularly evident in literature and in linguistics where data are in the form of huge corpora and dictionaries. But this applies as well to computer science where large amounts 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 [15], while the optimal average time complexity of the problem is $\Theta(\frac{n \log_\sigma m}{m})$ [22], achieved for example by the Backward-Dawg-Matching (BDM) algorithm [6]. Many string matching algorithms have been also developed to obtain

Simone Faro, Francesco Pio Marino: Reducing Time and Space in Indexed String Matching by Characters Distance Text Sampling, pp. 148–159.
Proceedings of PSC 2020, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06749-9 © Czech Technical University in Prague, Czech Republic

sub-linear performance in practice [7]. Among them the Boyer-Moore-Horspool algorithm [2,12] 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 $O(\sigma^2)$ [7]. However their performances 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 [1], 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, 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 compressions schemes, as Ziv-Lempel [20] and Huffman [3], the best practical behaviour are achieved by ad-hoc schemes designed for allowing fast searching [17,19,14,21,11]. 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.

A more suitable solution to the problem is *sampled string matching*, introduced in 1991 da Vishkin [23], 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.

Apart the theoretical result of Vishkin, a more practical solution to sampled string matching has been recently introduced by Claude *et al.* [5], based on an alphabet

[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) [4].

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

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 texts. Thus it turns out to be one of the most effective and flexible solution for this kind of searching problems.

They also consider indexing the sampled text. 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 [13] as both index a subset of the suffixes, but the different sampling properties induce rather different search algorithms and performance characteristics.

In this paper we present a new approach to the indexed string matching problem based on a different text sampling approach. The main idea behind our new text sampling approach is to sample the distances between consecutive occurrences of a given set of *pivot characters* and then to create a suffix array of the sampled text. For this reason we call this approach *characters distance sampling* (Cds). From our experimental results it turns out that our indexed solution based on such new sampling approach is able to obtain a searching time gain up to 91%, when compared to the standard indexed approach, while using less than 15% of the space needed for the standard suffix array.

The paper is organized as follows. In Section 2 we introduce in details the sampled string matching problem and present the first indexed approach to the problem proposed by Claude *et al.* [5]. Then, in Section 3, we present the new Cds approach to text sampling and propose an alternative indexed algorithm based on the suffix array. In Section 4 we present experimental results in order to compare our new solution against that proposed by Claude *et al.* and against the standard suffix array solution, in terms of both space and time.

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.

Sampled string matching has been introduced for the first time by Claude *et al.* in [5] where the author presented an online and an offline solution to the problem. More recently Faro *et al.* presented in an unpublished paper [13] an alternative solution for the online problem which turns out to be more efficient both in terms of space consumption and running times.

In this section we briefly describe the efficient indexed text-sampling approach proposed by Claude *et al.*. We will refer to this solution as the Occurrence-Text-Sampling approcah (OTS). To the best of our knowledge it is the only effective and flexible solution known in literature for the indexed sampled matching problem.

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 $y[\rho(2) + 3 - 1..\rho(3)]$, 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.

In [5] the authors also consider indexing the sampled text. Specifically they build a suffix array for the sampled text in order to index the sampled positions of the text. This approach is similar to the sparse suffix array [13] as both index a subset of the suffixes, but the different sampling properties induce rather different search algorithms and performance characteristics.

To optimize the entire process when constructing the suffix array their algorithm stores only suffixes starting with a sampled character. As a consequence, this approach can only be used for patterns which contain at least one character of the sampled alphabet.

The searching phase can be summarised as follows: the pattern is divided into two parts, the unsampled prefix, and the suffix starting with a sampled character. First of all the algorithm searches the sampled suffix of the pattern using the suffix array. Each candidate occurrence retrieved by this preliminary search will then be verified by comparing the unsampled prefix against the original text.

This approach turns out to perform very well for moderate to long patterns. Specifically according to their experimental evaluation it turns out that, when searching on an English text, the best performance are obtained when the number of removed characters from the original alphabet ranges between 13 and 16.

3 An Approach Based on Characters Distance Sampling

In this section we present a new efficient approach to the sampled string matching problem, introducing a new method[4] 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.* [5]. In the next subsections we illustrate in details our idea and describe the algorithms for the construction of the sampled text.

3.1 Characters Distance Sampling

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

[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.

[4] The Charcaters Distance Sampling approach presented in this section has been previously introduced by the authors in [9] where the online string matching problem has been taken into account.

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 characters we sample the text y by taking into account the distances between consecutive positions of any character of C in y. More formally our sampling approach is based on the following definition of *position sampling* of a text.

Definition 2 (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*

$$\begin{array}{ll} (i)\ 1 \leq \delta(i) < \delta(i+1) \leq n & \text{for each } 1 \leq i \leq n_c - 1 \\ (ii)\ y[\delta(i)] \in C & \text{for each } 1 \leq i \leq n_c \\ (iii)\ y[\delta(i)+1..\delta(i+1)-1] \text{ contains no character of } C & \text{for each } 0 \leq i \leq n_c \end{array}$$

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

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 3. 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 c is at position 1 ($y[1] =$ a), its second occurrence is at position 3 ($y[3] =$ a), and so on.

Definition 4 (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*

$$\bar{y} = \langle \Delta(1), \Delta(2), .., \Delta(n_c - 1) \rangle = \langle \delta(2) - \delta(1), \delta(3) - \delta(2), .., \delta(n_c) - \delta(n_c - 1) \rangle \tag{2}$$

Plainly we have

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

Example 5. Let $y =$ "agaacgcagtata" be a text of length 13, over the alphabet $\Sigma = \{$a,c,g,t$\}$. Let $C = \{$a$\}$ be the pivot character. 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 6 (Rank of a character). *Let x be a pattern of length m, and let $c \in \Sigma$. We define $\phi : \Sigma \rightarrow \{0..m\}$ 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. More formally the rank of c is given by the cardinality of the set $\{k \in \Sigma \mid \phi(k) < \phi(c)\} + 1$*

3.2 String Matching with the Suffix Array of the Sampled Text

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.

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 [18] as a simple, space efficient alternative to suffix trees. It has been extensively studied in the last three decades and in 2016 Li, Li and Huo [16] 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 $y[s_y[i-1]..n] < y[s_y[i]..n]$.

The algorithm proposed in this section is divided into two phases: a first *preprocessing phase* which consists in the construction of a suffix array of the sampled version of the text 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 the preprocessing phase is performed only once for the construction of the partial index, while the searching phase can be run an indeterminate number of times. 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 $s_{\bar{y}}$.

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 4) we will show how it is enough to put a single character in the set of pivot characters to obtain very good and competitive results. We select such character 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 6).

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

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.

However 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. Apart from this detail, all other features of the data structure remain unchanged.

Example 7. Let $y =$ "agaacgcagtata" be a text of length 13, over the alphabet $\Sigma = \{\text{a,c,g,t}\}$. Let $C = \{\text{a}\}$ be the pivot character. Thus the character distance sampling version of y is $\bar{y} = \langle 2, 1, 4, 3, 2 \rangle$.

$s_{\bar{y}}[0] = 1 \rightarrow \langle 1, 4, 3, 2\rangle$
$s_{\bar{y}}[1] = 0 \rightarrow \langle 2\rangle$
$s_{\bar{y}}[2] = 4 \rightarrow \langle 2, 1, 4, 3, 2\rangle$
$s_{\bar{y}}[3] = 3 \rightarrow \langle 3, 2\rangle$
$s_{\bar{y}}[4] = 2 \rightarrow \langle 4, 3, 2\rangle$

During the searching phase the algorithm uses the suffix array of the sampled text $s_{\bar{y}}$ as an index to quickly locate every occurrence 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 finding 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. All other occurrence are then grouped together close the first one.

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 $O(\log n_c)$-time [18] while finding the set 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 $O(\log(n_c) + m\rho)$.

4 Experimental Evaluation

In this section we report the results of an extensive evaluation of the new presented indexed algorithm based on Character Distance Sampling (CDS) in comparison with the standard searching algorithm based on the suffix array and the indexed algorithm based on the Occurrence Text Sampling (OTS) approach by Claude *et al.* [5].

The algorithms have been implemented in C, and have been tested using a variant of the SMART tool [8], properly tuned for testing string matching algorithms based on the indexed 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]

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

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.

Figure 1 shows the space consumption of the newly proposed text-sampling algorithms for different values of the rank r of the pivot character, whose value ranges from 2 to 20. Specifically it shows the size of the additional space (the size of 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`.

[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 5 MB of the resulting text buffer have been used in our experimental results.

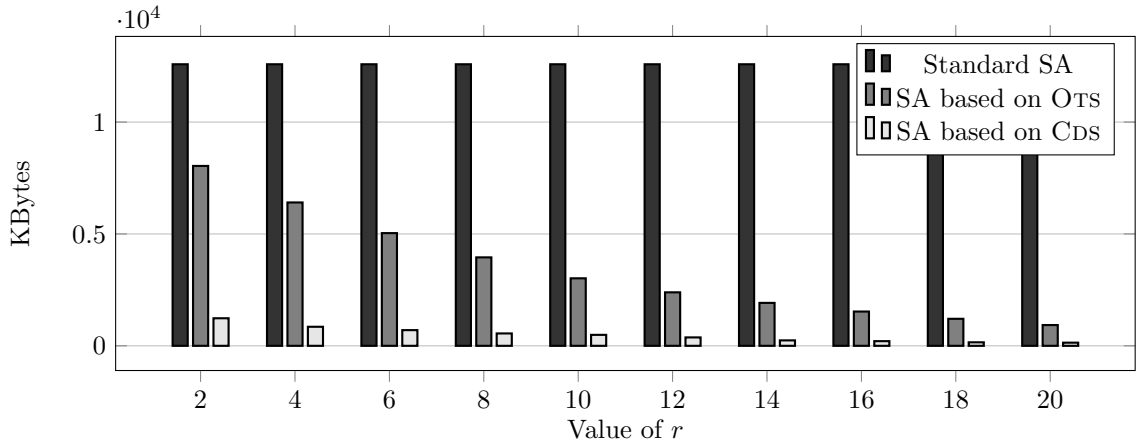

Figure 1. Size of the additional space (the size of the suffix array plus the size of the sampled text) consumed using the three compared approaches and specifically: the standard suffix array, the suffix array of a sampled text using an Ots approach and the suffix array of a sampled text using an Cds approach. Sizes are represented in KBytes. The x axis represents the rank of the pivot character in the case of the Cds approach, while represents the number of removed characters in the case of the Ots approach. We used a natural language text of size 5 Mb.

suffix array plus the size of the sampled text) consumed using the three compared approaches: the standard suffix array, the suffix array of a sampled text using an Ots approach and the suffix array of a sampled text using an Cds approach. Notice that sizes are represented in KBytes. The x axis represents the rank of the pivot character in the case of the Cds approach, while represents the number of removed characters in the case of the Ots approach. We used a natural language text of size 5 Mb.

As expected, the function which describes memory requirements follows a decreasing trend while the value of r decreases. Specifically the benefit in space consumption obtained by the algorithms based on character distance sampling ranges from 70% to 80% when compared with the Ots approach.

Figure 2 and Figure 3 show the experimental results obtained by comparing the three approaches to indexed searching in terms of running times, in a graphic and in a tabular representation, respectively. 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 $\{8, 16, 32, 64, 128, 256\}$. In all cases, the sum over the running times (expressed in milliseconds) of 1000 runs has been reported.

From such experimental results it turns out that the indexed searching approach based on Ots reaches a speed-up between 61% and 74%, while our new proposed solution reaches a speed-up between 80% and 91%. In addition the best results are obtained in all cased by the approach based on character distance sampling. In general the behaviour of text-sampling algorithms follow an increasing trend for increasing rank values. Thus in most cases the better choice is to use the most frequent element as the pivot character.

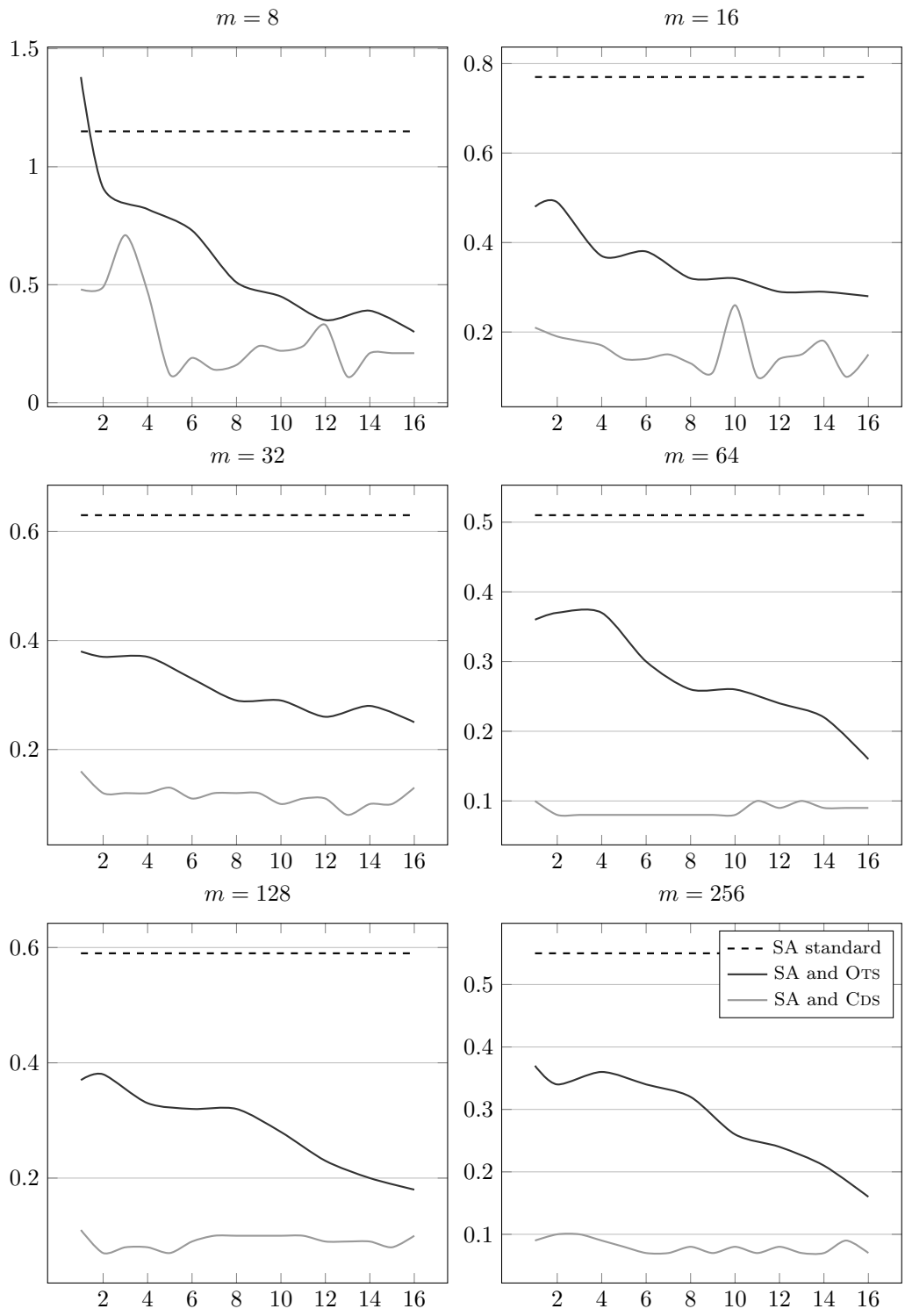

Figure 2. Running times of the text sampling algorithms in the case of long patterns ($8 \leq m \leq 256$). The dashed red line represent the Searching time of the standard Suffix array. Running times (in the y axis) are represented in thousands of seconds. The x axis represents the rank of the pivot character in the case of the new algorithm, while represents the number of removed characters in the case of the OTS algorithms. The searching time represented is the sum of all the 1000 tests executed for each length of the pattern. We are using a text of size 5 Mb.

$m = 8$

r	1	2	4	6	8	10	12	14	16
SA standard	1.15	1.15	1.15	1.15	1.15	1.15	1.15	1.15	1.15
SA and Ots	0.48	0.49	0.37	0.38	0.32	0.32	0.29	0.29	0.28
SA and Cds	0.48	0.49	0.47	0.19	0.16	0.22	0.33	0.21	0.21

$m = 16$

r	1	2	4	6	8	10	12	14	16
SA standard	0.77	0.77	0.77	0.77	0.77	0.77	0.77	0.77	0.77
SA and Ots	0.48	0.49	0.37	0.38	0.32	0.32	0.29	0.29	0.28
SA and Cds	0.21	0.19	0.17	0.14	0.13	0.26	0.14	0.18	0.15

$m = 32$

r	1	2	4	6	8	10	12	14	16
SA standard	0.63	0.63	0.63	0.63	0.63	0.63	0.63	0.63	0.63
SA and Ots	0.38	0.37	0.37	0.33	0.29	0.29	0.26	0.28	0.25
SA and Cds	0.16	0.12	0.12	0.11	0.12	0.10	0.11	0.10	0.13

$m = 64$

r	1	2	4	6	8	10	12	14	16
SA standard	0.51	0.51	0.51	0.51	0.51	0.51	0.51	0.51	0.51
SA and Ots	0.36	0.37	0.37	0.30	0.26	0.26	0.24	0.22	0.16
SA and Cds	0.10	0.08	0.08	0.08	0.08	0.08	0.10	0.09	0.09

$m = 128$

r	1	2	4	6	8	10	12	14	16
SA standard	0.59	0.59	0.59	0.59	0.59	0.59	0.59	0.59	0.59
SA and Ots	0.37	0.38	0.33	0.32	0.32	0.28	0.23	0.20	0.18
SA and Cds	0.11	0.07	0.08	0.09	0.10	0.10	0.09	0.09	0.10

$m = 256$

r	1	2	4	6	8	10	12	14	16
SA standard	0.55	0.55	0.55	0.55	0.55	0.55	0.55	0.55	0.55
SA and Ots	0.37	0.34	0.36	0.34	0.32	0.26	0.24	0.21	0.16
SA and Cds	0.09	0.1	0.09	0.07	0.08	0.08	0.08	0.07	0.07

Figure 3. Running times of the text sampling algorithms in the case of long patterns ($8 \leq m \leq 256$). The dashed red line represent the Searching time of the standard Suffix array. 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. The searching time represented is the sum of all the 1000 tests executed for each length of the pattern. We are using a text of size 5 Mb.

References

1. A. Apostolico: *The myriad virtues of suffix trees.* In: A. Apostolico, Z. Galil (Eds.), Combinatorial Algorithms on Words, Vol. 12 of NATO Advanced Science Institutes, Series F, Springer-Verlag, pp. 85–96 (1985).
2. R.S. Boyer and J.S. Moore: *A fast string searching algorithm.* Commun. ACM 20(10), 762-772 (1977).
3. D. Cantone, S. Faro, and E. Giaquinta: *Adapting Boyer-Moore-like Algorithms for Searching Huffman Encoded Texts.* Int. J. Found. Comput. Sci. 23(2), pp. 343–356 (2012).
4. D. Cantone, S. Faro, A. Pavone: *Speeding Up String Matching by Weak Factor Recognition.* Stringology 2017, pp. 42–50 (2017).
5. F. Claude, G. Navarro, H. Peltola, L. Salmela, and J. Tarhio: *String matching with alphabet sampling.* Journal of Discrete Algorithms, vol. 11, pp. 37–50 (2012).
6. M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, W. Rytter: *Speeding up two string-matching algorithms.* Algorithmica 12 (4), pp. 247–267 (1994).
7. S. Faro and T. Lecroq: *The Exact Online String Matching Problem: a Review of the Most Recent Results.* ACM Computing Surveys (CSUR) vol. 45 (2), pp. 13 (2013).

8. S. Faro, T. Lecroq, S. Borzì, S. Di Mauro, and A. Maggio: *The String Matching Algorithms Research Tool.* In Proc. of Stringology, pages 99–111, 2016.
9. S. Faro, F.P. Marino, and A. Pavone: *Efficient Online String Matching Based on Characters Distance Text Sampling.* arXiv:1908.05930, 2018.
10. P. Ferragina and G. Manzini: *Indexing compressed text.* Journal of the ACM, 52 (4), pp. 552–581, 2005.
11. K. Fredriksson and S. Grabowski: *A general compression algorithm that supports fast searching.* Information Processing Letters, vol. 100 (6), pp. 226–232 (2006).
12. R. N. Horspool: *Practical fast searching in strings.* Software: Practice & Experience 10 (6), pp. 501–506 (1980).
13. J. Karkkainen and E. Ukkonen: *Sparse suffix trees.* In: Proc. 2nd Annual International Conference on Computing and Combinatorics (COCOON), LNCS 1090, pp. 219–230 (1996).
14. S. T. Klein and D. Shapira: *A new compression method for compressed matching.* In: Data Compression Conference, IEEE. pp. 400–409 (2000).
15. D. E. Knuth, J. H. Morris, and V. R. Pratt: *Fast pattern matching in strings.* SIAM J. Comput. 6 (2), pp. 323–350 (1977).
16. Z. Li, J. Li, and H. Huo: *Optimal In-Place Suffix Sorting.* Proceedings of the 25th International Symposium on String Processing and Information Retrieval (SPIRE). Lecture Notes in Computer Science. 11147. Springer. pp. 268–284 (2016).
17. U. Manber: *A text compression scheme that allows fast searching directly in the compressed file.* ACM Trans. Inform. Syst., 15(2), pp.124–136 (1997).
18. U. Manber and G. Myers: *Suffix arrays: A new method for online string searches.* SIAM J. Comput. 22 (5), pp. 935–948 (1993).
19. E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates: *Fast and flexible word searching on compressed text.* ACM Transactions on Information Systems (TOIS), 18(2), pp.113–139 (2000).
20. G. Navarro and J. Tarhio: *LZgrep: A Boyer-Moore string matching tool for Ziv-Lempel compressed text.* Software Practice & Experience, vol. 35, pp. 1107–1130 (2005).
21. Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa: *Speeding Up Pattern Matching by Text Compression.* CIAC 2000: pp. 306–315.
22. A. C. Yao: *The complexity of pattern matching for a random string.* SIAM J. Comput. 8 (3), pp. 368–387 (1979).
23. U. Vishkin: *Deterministic sampling – A new technique for fast pattern matching.* In Proc. of the ACM Symposium on Theory of Computing (STOC), pp.170-180 (1990).