Beyond Horspool: A Comparative Analysis in Sampled Matching

Simone Faro[1], Francesco Pio Marino[1,2], and Andrea Moschetto[1]

[1] Dipartimento di Matematica e Informatica, Università di Catania, viale A.Doria n.6, 95125, Catania, Italia
[2] Univ Rouen Normandie, INSA Rouen Normandie, Université Le Havre Normandie, Normandie Univ, LITIS UR 4108, CNRS NormaSTIC FR 3638, IRIB, Rouen F-76000, France

Abstract. The *exact online string matching* problem, pivotal in fields ranging from computational biology to data compression, involves identifying all instances of a specified pattern within a text. Despite extensive examination over the decades, this problem has remained computationally challenging due to the time and space limitations inherent in traditional online and offline methods, respectively. Introduced in 1991, *sampled string matching* has now emerged as a groundbreaking approach, ingeniously combining classical online string matching techniques with efficient text sampling methods. This approach not only addresses the spatial constraints of indexed string matching but also significantly reduces the search duration in online environments, achieving speed increases of up to hundreds of times while requiring less than 4% of the text size for its partial index. In this paper, we explore the adaptability of various online string matching algorithms within the framework of sampled string matching, which has traditionally relied on the Horspool algorithm. Our investigation reveals that integrating alternative string matching algorithms as subroutines markedly enhances overall performance. These findings highlight the potential for reevaluating established methodologies in light of newer, more dynamic solutions and set the stage for transformative impacts across multiple domains.

1 Introduction

Given a text y of length n and a pattern x of length m over some alphabet Σ of size σ, the string matching problem consists of finding all occurrences of the pattern x in the text y. String matching is a crucial topic in the broader domain of text processing [6], and algorithms for this problem serve as fundamental components in the implementation of practical software across various operating systems. Although data are stored in various formats, text remains the primary medium for exchanging information. This is particularly evident in the literature of linguistics, where data consist of extensive corpora and dictionaries. Similarly, in computer science, a significant amount of data is stored in linear files. This holds true in fields like molecular biology as well, where biological molecules are often represented as sequences of nucleotides or amino acids.

Applications necessitate two distinct approaches: *online* and *offline* string matching. The former deals with unprocessed text, requiring real-time scrutiny during the search operation. Its worst-case time complexity is $\Theta(n)$, a milestone initially achieved by the well-known Knuth-Morris-Pratt (KMP) algorithm [20]. However, the average time complexity, $\Theta(\frac{n \log_\sigma m}{m})$ [24], was first achieved by the Backward-Dawg-Matching (BDM) algorithm [5]. Many string matching solutions have also been developed in order to achieve sub-linear performance in practical cases [6]. Among them, the Boyer-Moore-Horspool algorithm [2,17] is worth special mention, as it has been the inspiration for much work.

Simone Faro, Francesco Pio Marino, Andrea Moschetto: Beyond Horspool: A Comparative Analysis in Sampled Matching, pp. 16–26.
Proceedings of PSC 2024, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07328-5 © Czech Technical University in Prague, Czech Republic

Conversely, solutions adopting the second approach aim to expedite searches through preprocessing, constructing data structures that facilitate search operations. Termed *indexed searching*, this methodology consists on various efficient solutions. Notable examples include those leveraging suffix trees [1], boasting a $O(m + occ)$ worst-case time, suffix arrays [21], offering a respectable $O(m + \log n + occ)$ [21], and the FM-index [14] (Full-text index in Minute space), a compressed marvel derived from the Burrows-Wheeler transform, adeptly balancing input compression with swift substring queries. However despite their time performance, full-index data structures such as the ones just mentioned require additional space ranging from 4 to 20 times the size of the original text.

1.1 Sampled String Matching

Another solution in the literature consists in the realm of *Sampled String Matching* algorithms, pioneered by Vishkin in 1991 [23]. This method involves creating a succinct version of the text and then applying online string matching algorithms directly to the new version. This technique enables faster discovery of pattern occurrences, but each discovery within the sampled version of the text requires subsequent verification within the original text. Moreover, the sampled-text approach possesses several characteristics: it typically necessitates straightforward implementation, demands only a limited amount of additional space, and enables fast search and update operations.

Besides Vishkin's theoretical results, a more practical solution of the sampled string matching appeared more recently by Claut *et al.* [4], presenting an alphabet reduction technique (OTS). Their solution requires an extra space of 14% of the original text size, while speeding up the searching procedure up to 5 times traditional online string matching algorithms on English texts. Moreover they also introduced a indexed version of the sampled text, adapting the suffix array by indexing the sampled positions of the text.

More recently, Faro *et al.* have presented several algorithms on the field of sampling, especially with their *Character Distance Sampling* (CDS) approach [8,10,11,12,13]. In practical terms, through sampling absolute positions of some specific characters in the text, called *pivot characters*, their method has reached speedups of up to a factor of 9 on English texts, while demanding a limited additional space, ranging from 11% to 2.8% of the text's size. Achieving a 50% reduction in search times compared to the previous approach (OTS).

1.2 Our Contribution

In this paper, we investigate both the *OTS* and *CDS* algorithms in the context of online string matching. Indeed, both of these algorithms have traditionally been implemented based on the *Horspool* algorithm. We will analyze some practical results regarding the adaptability of these sampling string matching algorithms with various online algorithms. This paper is organized as follows: in Section 2 and Section 3 we briefly talk about the two practical existing sampling methods. In Section 4 we introduce the Online Algorithms selected for this study providing a short description for each of them. Finally in Section 5 we show the experimental results. In Section 6 we draw our conclusions.

2 The Occurrence Text Sampling algorithm

In this section we briefly describe the efficient text-sampling approach proposed by Claude *et al.* [4]. We will refer to this solution as the Occurrence-Text-Sampling algorithm (OTS).

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 informations, 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, 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 $\langle 5, 8, 13, 15 \rangle$. To search for the pattern $x =$ "acab" the algorithm constructs the sampled pattern $\hat{x} =$ "cb" and searches 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 the other match.

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 most frequent characters up to

some limit.[1] 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.

We point out that, despite demonstrating commendable performance in searching texts composed in natural languages, which typically feature relatively large alphabets, this text sampling technique exhibits significant limitations when applied to texts characterized by smaller alphabets, such as those found in genomic sequences and proteins. More recently, an adaptation of the OTS approach was introduced in [11]. This modification involves artificially enlarging the alphabet, thereby achieving improved experimental outcomes through the use of q-grams.

For the sake of completeness it has to be noticed that in [4] the authors also consider indexing the sampled text. Specifically they build a suffix array indexing the sampled positions of the text, and get a sampled suffix array. This approach is similar to the sparse suffix array [18] as both index a subset of the suffixes, but the different sampling properties induce rather different search algorithms and performance characteristics.

3 Characters Distance Sampling in Brief

In this section, we provide concise description of the methodology employed to build partial-index in the *Character Distance Sampling* (CDS).

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.

The algorithm selects a sub-alphabet $C \subseteq \Sigma$ to serve as the *set of pivot characters*. Using these designated pivots, it is possible to sample the text y by calculating the distances between the n_c consecutive occurrences of any pivot character $c \in C$ within y. Formally, this sampling methodology is based on the definition of *position sampling* within a text. Given $\delta : \{1, .., n_c\} \rightarrow \{1, .., n\}$, where $\delta(i)$ is the position of the i-th occurrence of any pivot character c in y. Then the position sampled version of y, indicated by $\dot{y}$, is a numeric sequence, of length n_c, defined as $\dot{y} = \langle \delta(1), \delta(2), .., \delta(n_c) \rangle$.

Example 2. Suppose $y =$ "agaacgcagtata" is a 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" is at position 1 ($y[1] =$ "a"), its second occurrence is at position 3 ($y[3] =$ "a"), and so on.

We can now define the *Character Distance Function* defined by $\Delta(i) = \delta(i+1) - \delta(i)$, 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 $\bar{y} = \langle \Delta(1), \Delta(2), .., \Delta(n_c - 1) \rangle = \langle \delta(2) - \delta(1), \delta(3) - \delta(2), .., \delta(n_c) - \delta(n_c - 1) \rangle$

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

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

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.

In practical scenarios, particularly when dealing with large alphabets, the set of pivot characters may comprise only one character. Consequently, for the sake of simplicity, we will frequently refer to the pivot character in the singular form, rather than mentioning the entire set of pivot characters.

The approach of sampled string matching utilizing CDS maintains a partial index, which is represented by the position-sampled version of the text y. The size of this index is $32n_c$ bits, assuming that this index resides in memory and is readily available for any search operation on the text. When there arises a need to search for a pattern x of length m within y, a preprocessing step is executed on the pattern to compute its sampled version $\bar{x}$. It can be straightforwardly proved that an occurrence of x in y corresponds to an occurrence of $\bar{x}$ in $\bar{y}$, hence it suffices to utilize any string matching algorithm to locate the occurrences of $\bar{x}$ in $\bar{y}$ to solve the problem. However, the reverse scenario is not necessarily true, implying that occurrences of $\bar{x}$ in $\bar{y}$ may not align with occurrences of x in y. Consequently, for each occurrence of $\bar{x}$ in $\bar{y}$, referred to as a candidate occurrence, a validation check in y is required.

Given that the validation process demands $O(m)$ computational time, the entire search operation will consume $O(mn)$ time. Nonetheless, envisioning modifications to the fundamental procedure to ensure that the overall search operation, despite the checks, remains linear in time is not challenging (for further details, refer to [10]).

An essential aspect to highlight in our discourse is that the CDS-based approach does not explicitly maintain the character-distance sampled version $\bar{y}$ of the text. Instead, it maintains the position-sampled version $\dot{x}$ of the text. Indeed, $\bar{y}$ solely retains the distances between the pivot characters and lacks direct ties to the original positions of these pivot characters within the text. Consequently, directly verifying every candidate occurrence becomes impractical. This issue is addressed by retaining the text $\dot{y}$, which holds the positions, and computing $\bar{y}$ on-the-fly during the search. The i-th element of $\bar{y}$ can indeed be computed in constant time using the relationship $\bar{y}(i) = \dot{y}(i + 1) - \dot{y}(i)$.

The CDS-based sampled string matching approach has demonstrated remarkable effectiveness in practical applications, boasting a significant reduction in search times by up to 40 times compared to standard online exact string matching techniques. Remarkably, this enhancement is achieved while incurring a relatively minimal cost, as it entails the construction of a partial index merely equivalent to 2% of the text size. Moreover, sampled string matching has exhibited exceptional flexibility, rendering it adept at addressing text searching challenges, even in the approximate realm. Notably, Faro *et al.*[13] recently introduced the run-length text sampling, tailored for approximate searches on texts. This technique proves particularly well-suited, for instance, for tasks such as *Order Preserving pattern matching* [19].

In addition to its commendable space and time efficiency, sampled string matching offers a plethora of other advantageous features. For instance, ease of programming stands out as a notable advantage, with the construction of the partial index typically being a swift and straightforward process. Moreover, the inherent flexibility of the data structure allows it to seamlessly adapt to text variations. This means that minor alterations in the text, such as character deletions or insertions, can be effortlessly reflected in the corresponding index.

However the one described above is not without its share of pitfalls or weaknesses. One such challenge is the variability in performance based on the choice of pivot character. Consequently, strategic consideration must be given to selecting the pivot character, striking a balance between partial index size and execution times. Research indicates that in the case of the English language the pivot character ranked 8th tends to offer best performances.

Another factor to consider is that if the pattern is exceptionally short and lacks occurrences of the pivot character, resorting to a standard string search within the text becomes necessary. Additionally, this method may not yield significant advantages when applied to texts with small alphabets, as the benefits in terms of space efficiency may not be realized. However, studies by Faro *et al.* [11] have proved the efficacy of a technique leveraging condensed alphabets to expand the underlying alphabet size and achieve markedly improved performance.

Recently new study conducted by the original authors showed new space and time improvement by using the fake distance representation [9].

4 Online String Matching Algorithms

In this section, we delve into the algorithms utilized to execute the searching phase within the context of the sampling methodologies. Our selection of algorithms aims to present a comparative analysis against the original sampling implemented through Horspool algorithm, considering their suitability across varied pattern and alphabet sizes. We have curated three distinct algorithms:

- Quick Search (QS) [22]: specifically tailored for large patterns and extensive alphabets.
- Weak Factor Recognition (WFR) [3]: for normal-sized patterns and smaller alphabets.
- Franek-Jennings-Smyth Algorithm (FJS) [16]: for very small patterns and large alphabets.

The subsequent sections provide an in-depth exploration of each algorithm, elucidating their underlying mechanisms and performance characteristics.

4.1 QS

The Quick-Search algorithm proposed by Sunday presents a simple variation of the Horspool algorithm. After each attempt, the shift is computed based on the character immediately following the current window of the text, denoted as $t[s+m]$. This corresponds to advancing the shift by $qbc_p(t[s+m])$ positions, where

$$qbc_p(c) = \min\{1 \le k \le m \mid p[m-k] = c\} \cup \{m+1\}$$

for all $c \in \Sigma$.

4.2 WFR

In the preprocessing phase, all factors of the pattern x are indexed to speed up the subsequent searching phase. Specifically, we define a hash function $h : \Sigma^* \to \{0, \ldots, 2^\alpha - 1\}$, which associates an integer value $0 \le v < 2^\alpha$ (for a fixed bound

α) with any string over the alphabet Σ. The value of α in the definition of the hash function h may depend on the target machine on which the algorithm is implemented. In our setting, the value of α has been fixed to 16 so that each hash value fits perfectly into a single 16-bit register. Although greater values of α are possible, we observed that the average number of false positives due to the hash function is negligible when the value of α is set to 16. A factor table F will store the hash values of the factors of x. It can be implemented as a table of Boolean values such that, for $0 \leq v < 2^{\alpha}$, $F[v]$ is set (i.e., $F[v] = \text{True}$) if and only if there exists a factor z of x such that $h(z) = v$.

During the searching phase, a window of size m slides along the text, starting at position 0. After each attempt, the window is shifted to the right until the end of the text is reached. Specifically, in the attempt at a given position i of the text, the window is opened on the substring $y[i..i+m-1]$. Thus, the `WFR` algorithm computes the hash values $h_{i,l}$, starting from $l = 1$ and for increasing values of l, until either $F[h_{i,l}] = \text{False}$ or $l = m$ (and, therefore, $F[h_{i,l}] = \text{True}$).

4.3 FJS

The Franek-Jennings-Smyth string matching algorithm (FJS for short) is a simple hybrid algorithm which mixes the linear worst-case time complexity of Knuth-Morris-Pratt algorithm and the sublinear average behavior of Quick-Search algorithm. Specifically the FJS algorithm searches for matches of p in t by shifting a window of size m from left to right along t. Each attempt of the algorithm is divided into two steps. During the first step, in accordance with the Quick-Search approach, the FJS algorithm first compares the rightmost character of the pattern, $p[m-1]$, with its corresponding character in the text, that is, $t[s+m-1]$. If a mismatch occurs, a Quick-Search shift is implemented, moving p along t until the rightmost occurrence in p of the character $t[s+m]$ is aligned with position $s+m$ in the text. At this new location, the rightmost character of p is again compared with the corresponding text position. Only when a match is found the FJS algorithm invokes the second step. Otherwise another Quick-Search shift occurs. The second step of the algorithm consists in a Knuth-Morris-Pratt pattern matching starting from the leftmost character $p[0]$ and, if no mismatch occurs, extending as far as $p[m-2]$. Then whether or not a match of p is found, a Knuth-Morris-Pratt shift is eventually performed followed by a return to the first step. The preprocessing time of the algorithm takes $O(m)$ time for computing the failure function of KMP, and $O(m+\sigma)$ time in order to compute the Quick-Search bad character rule. The authors showed that the worst-case number of character comparisons is bounded by $3n-2m$, so that the corresponding searching phase requires $O(n)$time. The space complexity of the algorithm is $O(m+\sigma)$

5 Experimental Results

In this section, we present experimental results to evaluate the performance of the sampled string matching approaches outlined in this paper.

The algorithms were implemented in the C programming language and tested using the Smart tool [7] 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.[2] The algorithms were compiled using the -O3 optimization option. Performance

[2] The Smart tool can be downloaded from `http://www.dmi.unict.it/~faro/smart/` or from `https://github.com/smart-tool/smart`.

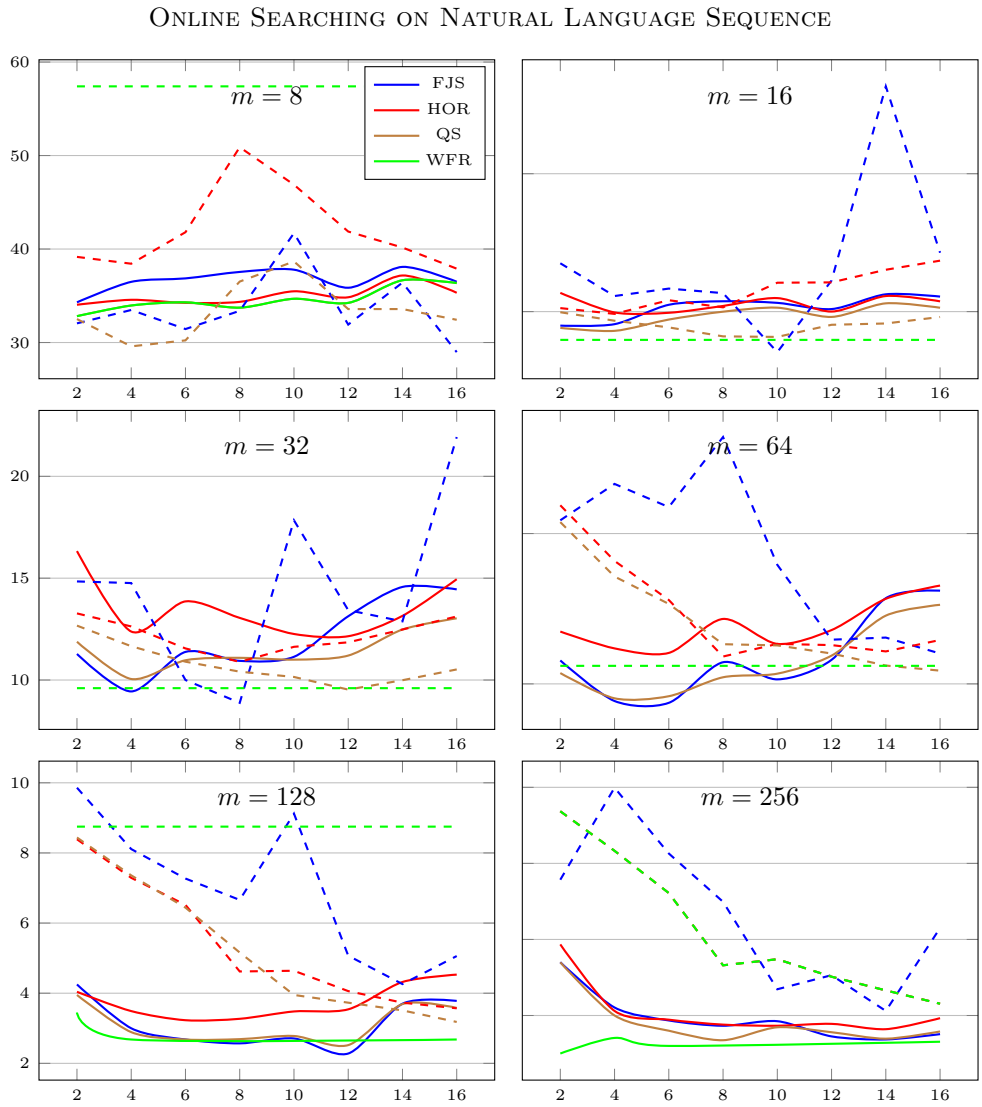

Figure 1. Experimental results of the sampling algorithms implemented through different online string matching algorithms, using English Texts. Straight lines represent the CDS algorithms while dashed lines represent OTS ones. Times are represented in the y axes in milliseconds (ms), while the x axes represents the rank of the pivot character $2 \leq K \leq 16$.

comparisons were made based on searching times and are expressed in milliseconds (ms).

Two text buffers, each with a size of 100 MB, were used, sourced from the *Pizza and Chili* dataset [15], available online for download. Specifically, the algorithms were tested using a genomics data sequence and a natural language text. For each sequence, 500 patterns were randomly selected from the text, and the average running time was computed over the 500 runs.
Moreover as previously pointed out in this paper, sampling technique exhibits significant limitations when applied to texts characterized by smaller alphabets, such as those found in genomics sequences and proteins.

For both the natural language dataset and the genomics sequences, experiments were conducted for normal-sized patterns ($8 \leq m \leq 256$). Furthermore, for algorithms that can be implemented with different values of the parameter q, tests were

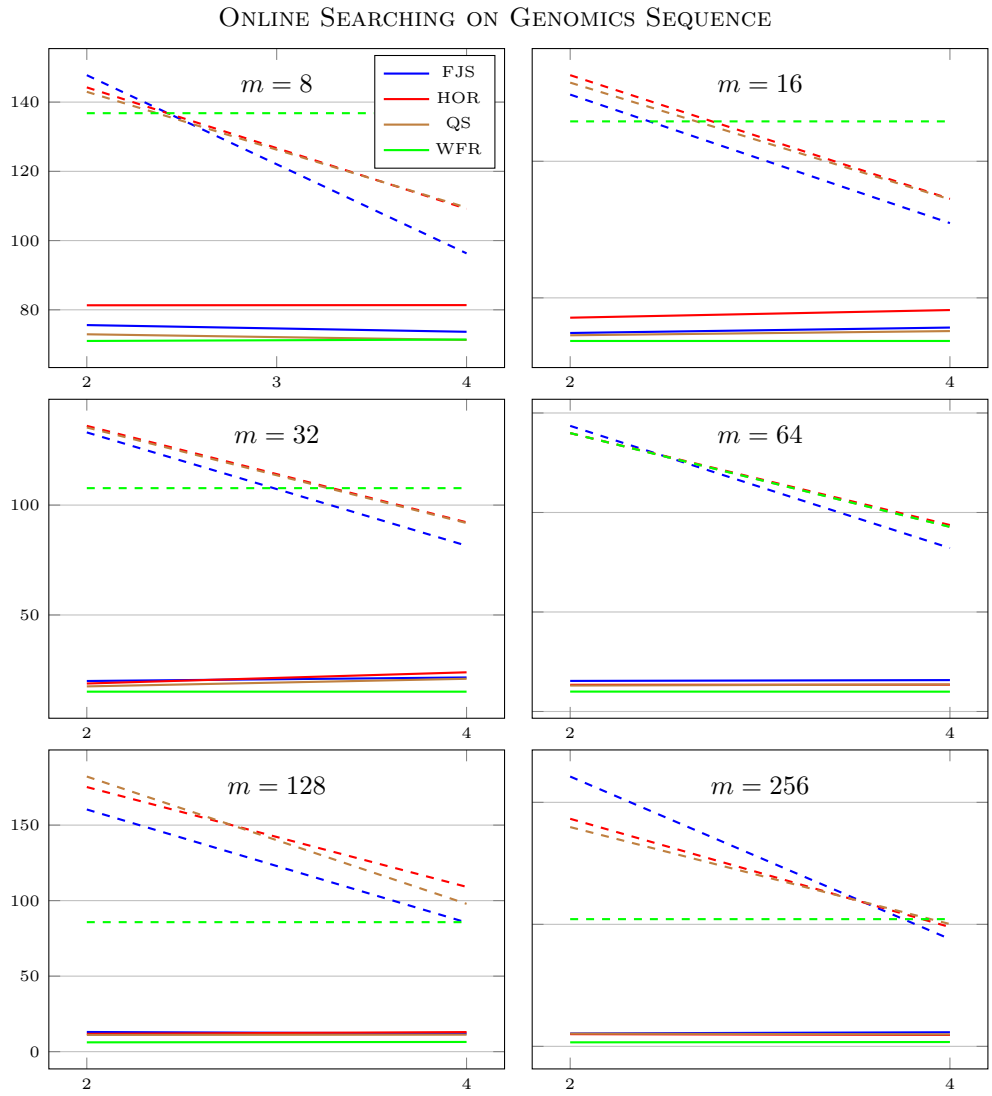

Figure 2. Experimental results of the sampling algorithms implemented through different online string matching algorithms, using Genomics Sequences. Straight lines represent the CDS algorithms while dashed lines represent OTS ones. Times are represented in the y axes in milliseconds (ms), while the x axes represents the rank of the pivot character $2 \leq K \leq 4$.

conducted for multiple values, generally $1 \leq q \leq 8$, and only the best result among the different values of q was considered for each pattern size. Let K be the rank of a character in a totally ordered alphabet, we conducted our tests for different values of K, indeed for the natural language dataset we used $2 \leq K \leq 16$, while for the genomics sequences given $|\Sigma| = 4$ we could use only $2 \leq K \leq 4$. In both Figure 1 and 2 the legend show the color of the algorithm used in the searching phase, while the smooth line represent the CDS algorithm has been used, and the dashed lines are used to represent the OTS algorithm.

6 Conclusions

In this paper we have presented an extension of the text sampling approaches, called Character Distance and Occurrence Text Sampling, to the case of applying different

searching algorithms. This extension was carried out using four different well known algorithms. Our results proved the efficacy of the sampled methods discussed in this paper and their versatility to be adapted to any online string matching algorithms without impacting their original performances.

Although our tests were limited to the exact string matching problem, obtaining excellent results, we believe that the approach can be effectively generalized even to non-standard string matching. Our future studies will focus in this direction in order to apply sampled string matching to other problems related to text processing.

Acknowledgements

Simone Faro was supported by the National Centre for HPC, Big Data and Quantum Computing, Project CN00000013, affiliated to Spoke 10, co-founded by the European Union – NextGenerationEU.

References

1. A. Apostolico: *The myriad virtues of subword trees*, in Combinatorial Algorithms on Words, A. Apostolico and Z. Galil, eds., Berlin, Heidelberg, 1985, Springer Berlin Heidelberg, pp. 85–96.
2. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Commun. ACM, 20(10) oct 1977, p. 762–772.
3. D. Cantone, S. Faro, and A. Pavone: *Speeding up string matching by weak factor recognition*, in Proceedings of the Prague Stringology Conference 2017, Prague, Czech Republic, August 28-30, 2017, J. Holub and J. Zdárek, eds., Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2017, pp. 42–50.
4. F. Claude Faust, G. Navarro, H. Peltola, L. Salmela, and J. Tarhio: *String matching with alphabet sampling.* Journal of Discrete Algorithms, 11 12 2010.
5. M. Crochemore: *Speeding up two string-matching algorithms.* Algorithmica, 12(4) 1994, pp. 247–267.
6. S. Faro and T. Lecroq: *The exact online string matching problem: A review of the most recent results.* ACM Comput. Surv., 45(2) mar 2013.
7. S. Faro, T. Lecroq, S. Borzi, S. D. Mauro, and A. Maggio: *The string matching algorithms research tool*, in Proceedings of the Prague Stringology Conference 2016, Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2016, pp. 99–111.
8. S. Faro and F. P. Marino: *Reducing time and space in indexed string matching by characters distance text sampling*, in Prague Stringology Conference 2020, Prague, Czech Republic, August 31 - September 2, 2020, J. Holub and J. Zdárek, eds., Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2020, pp. 148–159.
9. S. Faro, F. P. Marino, A. Moschetto, A. Pavone, and A. Scardace: *The Great Textual Hoax: Boosting Sampled String Matching with Fake Samples*, in 12th International Conference on Fun with Algorithms (FUN 2024), A. Z. Broder and T. Tamir, eds., vol. 291 of Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2024, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 13:1–13:17.
10. S. Faro, F. P. Marino, and A. Pavone: *Efficient online string matching based on characters distance text sampling.* Algorithmica, 82(11) 2020, pp. 3390–3412.
11. S. Faro, F. P. Marino, and A. Pavone: *Enhancing characters distance text sampling by condensed alphabets*, in Proceedings of the 22nd Italian Conference on Theoretical Computer Science, Bologna, Italy, September 13-15, 2021, C. S. Coen and I. Salvo, eds., vol. 3072 of CEUR Workshop Proceedings, CEUR-WS.org, 2021, pp. 1–15.
12. S. Faro, F. P. Marino, and A. Pavone: *Improved characters distance sampling for online and offline text searching.* Theor. Comput. Sci., 946 2023, p. 113684.

13. S. Faro, F. P. Marino, A. Pavone, and A. Scardace: *Towards an efficient text sampling approach for exact and approximate matching*, in Prague Stringology Conference 2021, Prague, Czech Republic, August 30-31, 2021, J. Holub and J. Zdárek, eds., Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2021, pp. 75–89.
14. P. Ferragina and G. Manzini: *Indexing compressed text.* J. ACM, 52(4) jul 2005, p. 552–581.
15. P. Ferragina and G. Navarro: *Pizza&Chili*, Available online: pizzachili.dcc.uchile.cl/, 2005.
16. F. Franek, C. G. Jennings, and W. F. Smyth: *A simple fast hybrid pattern-matching algorithm.* J. Discrete Algorithms, 5(4) 2007, pp. 682–695.
17. R. N. Horspool: *Practical fast searching in strings.* Software: Practice and Experience, 10(6) 1980, pp. 501–506.
18. J. Kärkkäinen and E. Ukkonen: *Sparse suffix trees*, in Computing and Combinatorics, J.-Y. Cai and C. K. Wong, eds., Berlin, Heidelberg, 1996, Springer Berlin Heidelberg, pp. 219–230.
19. J. Kim, P. Eades, R. Fleischer, S.-H. Hong, C. S. Iliopoulos, K. Park, S. J. Puglisi, and T. Tokuyama: *Order-preserving matching.* Theoretical Computer Science, 525 2014, pp. 68–79, Advances in Stringology.
20. D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt: *Fast pattern matching in strings.* SIAM Journal on Computing, 6(2) 1977, pp. 323–350.
21. U. Manber and G. Myers: *Suffix arrays: A new method for on-line string searches.* SIAM Journal on Computing, 22(5) 1993, pp. 935–948.
22. D. Sunday: *A very fast substring search algorithm.* Commun. ACM, 33(8) 1990, pp. 132–142.
23. U. Vishkin: *Deterministic sampling–a new technique for fast pattern matching.* SIAM Journal on Computing, 20(1) 1991, pp. 22–40.
24. A. C.-C. Yao: *The complexity of pattern matching for a random string.* SIAM Journal on Computing, 8(3) 1979, pp. 368–387.