Improving Sampled Matching through Character Context Sampling

Simone Faro[1,†], Thierry Lecroq[2], Francesco Pio Marino[1,2,†], Arianna Pavone[3,‡] and Stefano Scafiti[1]

[1]Università di Catania, viale A. Doria n.6, 95125, Catania, Italy
[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
[3]Università di Palermo, via Archirafi n.34, 90123, Palermo, Italy

Abstract

Sampled String Matching is a hybrid approach to the string matching problem, blending online and offline solutions. Among various sampling methods, *Character Distance Sampling* (CDS) is one of the fastest and most versatile techniques. In optimal conditions, CDS can achieve speedup of up to 100 times, while requiring minimal additional space — ranging from 10% to as little as 0.8% of the original text size. Furthermore, CDS is adaptable, effectively handling non-classical string matching problems and computing structural properties of strings such as periods and coverages. As with all sampling-based matching algorithms, a verification phase on the original text is necessary after the initial matching on the sampled strings. Often, the computational effort required for this verification phase can be substantial. In this article, we introduce a novel sampling method that tracks the context around each sampled location rather than the distances to those locations. This approach aims to reduce the number of candidate occurrences and the subsequent verification effort. Our experimental results indicate that the proposed method outperforms CDS, particularly for short patterns, achieving a speedup of between 15% and 40%.

Keywords

String Matching, Sampled String Matching, Text Processing, Contextual Information

1. Introduction

The *string matching* problem involves identifying all instances of a pattern x of length m within a text y of length n, both defined over an alphabet Σ of size σ. This task is fundamental in text processing and underpins various software implementations across multiple operating systems. Its importance is highlighted by the continued prevalence of text as the primary medium for information exchange, despite diverse data storage formats. This is particularly evident in linguistics, which relies on extensive corpora and dictionaries, and in computer science, where large amounts of data are stored in linear files.

ICTCS'24: Italian Conference on Theoretical Computer Science, September 11–13, 2024, Torino, Italy
[†] Supported by University of Catania, progetto PIACERI 2024-2026 - Linea di Intervento 1
[‡] Supported by PNRR project ITSERR - Italian Strengthening of the ESFRI RI RESILIENCE
faro@dmi.unict.it (S. Faro); thierry.lecroq@univ-rouen.fr (T. Lecroq); francesco.marino@phd.unict.it (F. P. Marino); ariannamaria.pavone@unipa.it (A. Pavone); stefano.scafiti@unict.it (S. Scafiti)
0000-0001-5937-5796 (S. Faro); 0000-0002-1900-3397 (T. Lecroq); 0000-0003-4722-9542 (F. P. Marino); 0000-0002-8840-6157 (A. Pavone); 0000-0001-7042-3912 (S. Scafiti)

Applications of string matching require two main approaches: *online* and *offline* string matching. The online approach handles unprocessed text, necessitating real-time analysis during the search operation. Its worst-case time complexity is $\Theta(n)$, first achieved by the renowned Knuth-Morris-Pratt (KMP) algorithm [1]. Its average time complexity is $\Theta\left(\frac{n \log_\sigma m}{m}\right)$ and was initially realized by the Backward-Dawg-Matching (BDM) algorithm [2]. Numerous solutions have been devised to attain sub-linear performance in practical scenarios [3], with the Boyer-Moore-Horspool algorithm [4, 5] standing out, having influenced subsequent research.

In contrast, the offline approach aims to expedite searches through text preprocessing, creating data structures that facilitate search operations. Known as *indexed searching*, this method includes several efficient solutions. Prominent examples include suffix trees [6], which offer an $O(m+occ)$ worst-case time, suffix arrays [7] with a time complexity of $O(m+\log n+occ)$, where occ it the number of occurrences of the searched pattern, and the FM-index [8], a compressed structure derived from the Burrows-Wheeler transform that combines input compression with efficient substring queries. However, these full-indexes require additional storage space, ranging from four to twenty times the size of the text size.

A hybrid technique in the literature is *Sampled String Matching*, introduced by Vishkin in 1991 [9]. This method involves creating a partial index of the text and applying online string matching algorithms to this index. While such approach accelerates the detection of candidate pattern occurrences, each match in the sampled text must be verified within the original text.

The sampled-text methodology offers several benefits: it generally requires straightforward implementation, minimal additional space, and enables fast search and update operations. Beyond Vishkin's theoretical contributions, a more practical solution was presented by Claude *et al.* [10], who developed an alphabet reduction technique. Their method requires an extra space of 14% of the original text size and achieves up to a fivefold increase in search speed compared to traditional online string matching algorithms on English texts. They also introduced an indexed version of the sampled text, modifying the suffix array to index sampled positions.

Recently, Faro *et al.* have introduced several algorithms in the sampling field, notably their *Character Distance Sampling* (CDS) approach [11, 12, 13, 14, 15, 16]. By sampling absolute positions of specific characters, referred to as *pivot characters*, their method has achieved up to a ninefold increase in speed on English texts, while requiring only 11% to 2.8% additional space relative to the text size. This method provides a 50% reduction in search times compared to previous text sampling techniques [12].

In this paper, we introduce a novel sampling method called *Character Context Sampling* (CCS), which is designed to compactly track the context surrounding pivot characters identified within the text. This method involves storing the hash of the substring (or a portion thereof) located between two consecutive occurrences of the pivot character. Our experimental results demonstrate that this technique significantly reduces the number of verifications required to identify matches, thereby substantially decreasing search times, while maintaining the same amount of space required for storing the partial index.

The paper is organized as follows. In Section 2 we briefly review the CDS method and the most recent results achieved in this field. Section 3 introduces the new sampling method based on the use of context around pivot characters. In Section 4 we provide some experimental results and in Section 5 we draw our conclusions.

2. Characters Distance Sampling in Brief

This section outlines the methodology employed to build a partial index using the *Character Distance Sampling* (CDS) technique. These concepts are relevant for the description of the new sampling method introduced in this paper.

Consider an input text y of length n and an input pattern x of length m, both defined over an alphabet Σ of size σ. We treat all strings as vectors starting at position 0. Thus, $x[i]$ refers to the $(i+1)$-th character of the string x for $0 \leq i < m$.

The algorithm selects a sub-alphabet $C \subseteq \Sigma$ to serve as the *set of pivot characters.*[1] Using these designated pivots, the text y is sampled by calculating the distances between consecutive occurrences of any pivot character $c \in C$ within y. Formally, this sampling methodology is based on the concept of *position sampling* within the text.

Define $\delta : \{0, \ldots, n_c - 1\} \rightarrow \{0, \ldots, n-1\}$, where $\delta(i)$ represents the position of the $(i+1)$-th occurrence of a pivot character c in y. The position-sampled version of y, denoted by $\dot{y}$, is a numerical sequence of length n_c defined as: $\dot{y} = \langle \delta(0), \delta(1), \ldots, \delta(n_c - 1) \rangle$.

Define the *Character Distance Function* $\Delta : \{0, \ldots, n_c - 1\} \rightarrow \{0, \ldots, n-1\}$, where $\Delta(i) = \delta(i+1) - \delta(i)$ represents the distance between two consecutive occurrences of any pivot character in y. The *character-distance sampled version* of the text y, denoted by $\bar{y}$, is a numerical sequence of length $n_c - 1$ defined as:

$$\bar{y} = \langle \Delta(0), \Delta(1), \ldots, \Delta(n_c - 1) \rangle = \langle \delta(1) - \delta(0), \delta(2) - \delta(1), \ldots, \delta(n_c - 1) - \delta(n_c - 2) \rangle.$$

Example 1. *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. The position-sampled version of y is $\dot{y} = \langle 0, 2, 3, 7, 10, 12 \rangle$. Specifically, the first occurrence of character "a" is at position 0 ($y[0] =$ "a"), its second occurrence is at position 2 ($y[2] =$ "a"), and so on. In addition, the character-distance sampled version of y is $\bar{y} = \langle 2, 1, 4, 3, 2 \rangle$. Specifically, $\bar{y}[0] = \Delta(0) = \delta(1) - \delta(0) = 2 - 0 = 2$, while $\bar{y}[2] = \Delta(2) = \delta(3) - \delta(2) = 7 - 3 = 4$, and so on.*

The sampled string matching approach using CDS maintains a partial index, represented by the position-sampled version of the text y. The size of this index is $32n_c$ bits, assuming the index resides in memory and it is readily available for any search operation on the text.

When searching for a pattern x of length m within y, a preprocessing step computes its sampled version $\bar{x}$. It can be proven that an occurrence of x in y corresponds to an occurrence of $\bar{x}$ in $\bar{y}$. Thus, any string matching algorithm can be used to locate occurrences of $\bar{x}$ in $\bar{y}$ to solve the problem. However, the reverse is not necessarily true. Therefore, each occurrence of $\bar{x}$ in $\bar{y}$, termed a candidate occurrence, requires a validation check in y.

Given that the validation process takes $O(m)$ time, the entire search operation consumes $O(mn)$ time. Nevertheless, modifications to the fundamental procedure can ensure that the overall search remains linear in time (see [12] for further details) and can also be implemented using any online algorithm studied in literature [17].

An important aspect of the CDS-based approach is that it does not explicitly maintain the character-distance sampled version $\bar{y}$ of the text. Instead, it keeps the position-sampled version

[1] In practical applications, particularly when dealing with large alphabets, the set of pivot characters may include only one character. For simplicity, we often refer to the pivot character in the singular form.

$\dot{y}$. Since $\bar{y}$ retains only distances between pivot characters without direct ties to their original positions, direct verification of every candidate occurrence is impracticable. This is resolved by maintaining $\dot{y}$ and computing $\bar{y}$ on-the-fly during the search. The i-th element of $\bar{y}$ can be computed in constant time as $\bar{y}(i) = \dot{y}(i+1) - \dot{y}(i)$.

The CDS-based sampled string matching approach has proven highly effective in practical applications, significantly reducing search times by up to 40 times compared to standard online exact string matching techniques. This improvement comes at a relatively low cost, requiring the construction of a partial index only 2% of the text size.

Moreover, the sampled string matching method has shown exceptional flexibility, making it suitable for text searching challenges, including approximate searches. Notably, Faro *et al.* [15] recently introduced the run-length text sampling, which is tailored for approximate searches in texts, proving useful for tasks such as *Order Preserving pattern matching* [18, 19].

In addition to its space and time efficiency, the sampled string matching approach offers other advantages, such as ease of programming and the ability to adapt to text variations. Minor alterations in the text, like character deletions or insertions, can be easily reflected in the index.

However, this method is not without its challenges. One such challenge is the performance variability based on the choice of pivot character. Strategic selection of the pivot character is crucial to balance partial index size and execution times. Research suggests that in the English language, the pivot character ranked 8th often provides the best performance.

Another consideration is that if the pattern is very short and lacks occurrences of the pivot character, a standard string search within the text may be necessary. Additionally, the method may not yield significant benefits for texts with small alphabets, as space efficiency gains may not be realized. However, studies by Faro *et al.* [13] have demonstrated the effectiveness of techniques that use condensed alphabets to expand the alphabet size and improve performance.

A recent study introduced significant advancements in space and time efficiency through the introduction of *fake samples* [16], slightly increasing the number of elements in the partial index. Paradoxically, this results in a substantial three-quarters reduction in the overall space required to represent the data structure while maintaining algorithmic correctness. The idea lies in storing distances between pivot characters rather than their absolute positions within the text, which reduces the space used but introduces the challenge of direct addressing of positions within the original text.

The resulting *fake distance representation* of CDS leverages a clever balance between adding minimal redundancy and achieving significant space reduction. By interspersing fake samples within the pivot character set, the method ensures that the partial index retains its efficiency in identifying potential matches. This leads to quicker verification processes and overall faster search times.

For the sake of completeness, we emphasize that Lecroq and Marino have recently proven that the CDS representation can accelerate not only string matching algorithms but also other types of algorithms, such as those that compute structural properties of strings, including finding periodicities and covers [20]. This broadens the scope of CDS beyond traditional string matching, showcasing its adaptability and effectiveness in a wider range of computational problems.

3. Character Context Sampling

As introduced in the previous sections, sampled string matching stands as an hybrid method that allows a practical speed-up in the searching phase with minimal costs required for space and pre-processing time. This technique leverages the benefits of sampling to reduce the overall search complexity, making it highly efficient for large datasets.

On the other hand, a crucial requirement of all sampled string matching algorithms is the necessity of a verification phase after any candidate match is identified in sampled versions of the text. Although such verification phase can be easily implemented in $O(m)$ time by comparing all the pattern characters with the candidate substring in the text, in the worst-case scenario, where false matches occur at every sampled position, it would be necessary to perform a verification at each sampled position of the text. This results in a complexity of $O(nm)$, which is sub-optimal if compared with the linear time complexity achieved by several well-known algorithms. This trade-off highlights the importance of optimizing the sampling strategy to balance the speed-up in the search phase with the cost of running the verification phases.

While the information provided by the CDS approach is enough to compute positions in the original text and execute the searching phase using various online algorithms [17], it is not well designed for avoiding (or reducing) false matches between the sampled version of the text and the pattern. To obtain this result it is instead necessary to store additional data.

In this section, we introduce the *Character Context Sampling* (CCS) approach, an enhanced variant of the CDS method which stores information computed on the context around each pivot character instead of the distances stored by the CDS method. At the core of this idea is the necessity to incorporate contextual information about the position of each pivot character within the partial index used for the search. This approach aims to reduce the number of false positives, thereby minimizing the number of verifications required during the search phase.

When we refer to a *context*, we mean a fixed-size substring, of fixed length q, within the vicinity of the pivot character. Figure 1 schematically shows the idea on which the context-based sampling model is based. The Figure compares the CDS and CCS methods if the same pivot character is used. Although the context size is set to $q = 2$, when two occurrences of the pivot character closer than 2 characters apart then the context is reduced accordingly.

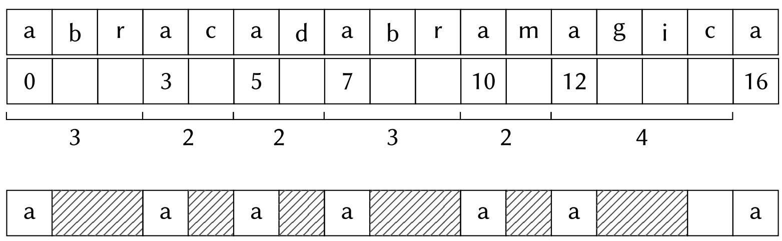

Figure 1: Text sampling of the string $x =$ abracadabramagica. On top, the Character Distance Sampling representation of x using the character a as the sole pivot. The exact positions of each occurrence of the pivot are stored, with the distances between consecutive occurrences indicated below. On the bottom, an example of Character Context Sampling of x, using the same pivot character and $q = 2$. Note that the last context has a length of 3, which is greater than our q; therefore, the context is reduced to $q = 2$.

However, we immediately notice that memorizing entire substrings is extremely more expensive than memorizing individual positions. For this reason, due to the potentially high memory cost of storing the entire set of substrings representing the set of contexts, our method stores the context in a compact and approximate form by computing a fingerprint of the substrings, specifically a hash value.

More formally, let $hash : \Sigma^* \times \{0, 1, \ldots, q-1\} \times \{0, 1, \ldots, q-1\}$ be a function which, taking as input a string $w \in \Sigma^*$ and two indices i and j such that $0 \leq i < j \leq |w|-1$, computes an hash value of the substring $w[i..j]$. In addition, let q be an integer parameter such that $q \geq 2$, and let $\dot{y}$ be the position sampled version of the text y. Then, the CCS version of a string y is defined as $\tilde{y} = \langle \text{ccs}(0), \text{ccs}(1), \ldots, \text{ccs}(n_c - 1) \rangle$, where $\text{ccs}(i)$ corresponds to the context value of the i-th pivot character, which is defined by

$$\text{ccs}(i) = \begin{cases} hash(y, \dot{y}[i]+1, \dot{y}[i+1]-1) & \text{if } \dot{y}[i+1] - (\dot{y}[i]+1) \leq q \\ hash(y, \dot{y}[i]+1, \dot{y}[i]+q) & \text{otherwise.} \end{cases}$$

In other words, the CCS version, $\tilde{y}$, of the text y is the sequence of hash values computed on the substring starting at the positions just after each pivot character in the text, whose length is at most q and which does not include the next pivot character. These hash values encapsulate contextual information that can be used during the search phase to reduce the number of verifications.

We also notice that, in order to locate the candidate occurrences in the original text for the verification phase, the CCS approach necessitates storing both the hash values and the exact positions of each pivot character. In the worst-case scenario, storing the complete set of positions may lead to an hybrid method consuming a significant amount of additional space. To address this issue, an alternative approach involves using a mapping table ρ, as demonstrated in the OTS algorithm [10]. In their method, a mapping position is stored at regular intervals; specifically, every t pivots, the exact position of the pivot is recorded. When a sampled match is found at position i, the verification phase commences at the position $\rho[\lfloor i/k \rfloor]$ and continues until the next pivot character (see [10] for more details).

The searching procedure is divided into two distinct phases: the pre-matching phase, also known as sample-matching, and the verification phase. The pseudo-code of the searching procedure is described in Figure 2 (on the right).

Le y be a text of length n, let x be a pattern of length m, both strings over an alphabet Σ, and let $C \subseteq \Sigma$ be the set of pivot characters. In order to retrieve the exact position in the text, an additional position f must be stored. This position f corresponds to the distance between the first pivot and the initial position of the text. Specifically $f = \min(i : y[i] \in C)$. Thus $f = 0$ if the text starts with a pivot character, otherwise $f > 0$.

Both phases are straightforward and similar in nature. The first phase checks if all the contexts of the sampled pattern $\tilde{x}$ occurs in the sampled text $\tilde{y}$. If a candidate occurrence is found at position i of $\tilde{y}$, then the second verification phase is initiated to check the presence of a real match.

During the verification phase, the starting position is computed. This requires $\dot{y}[s]$, which is the exact position of the first pivot that was compared, and the distance f between the first pivot in the pattern and the initial position of the text. Consequently, the position p can be computed

```
CHARACTER-CONTEXT-SAMPLING(x, m, q, p)
i ← 0
ṁ ← 0
while i < n do
    if x[i] = p then
        if ṁ = 0 then
            f ← i
        j ← 0
        while j < q and x[i + j] ≠ p do
            j ← j + 1
        x̃[ṁ] ← hash(x, i, j)
        i ← j
        ṁ ← ṁ + 1
    else i ← i + 1
```

```
SEARCH(ỹ, x̃, ṁ, ṅ, x, m, y, ρ, f)
res ← 0
s ← 0
while s ≤ ṅ − ṁ do
    i ← 0
    while i < ṁ and ỹ[s + i] = x̃[i] do
        i ← i + 1
    if i = ṁ then
        p ← COMPUTE-POSITION(ρ, s) − f
        i ← 0
        while i < m and y[p + i] = x[i] do
            i ← i + 1
        if i = m then
            res ← res + 1
return res
```

Figure 2: (On the left) The pseudo-code of the procedure to compute the Character Context Sampling version of a given string. (On the right) The pseudo-code for the brute-force searching procedure for the Character Context Sampling Matching.

as $p = \dot{y}[s] - f$. However, we point out that, in the procedure shown in Figure 2, the exact position $\dot{y}[s]$ is computed by sub-routine COMPUTE-POSITION which uses the mapping table ρ to identify the value of such position. Once the initial position p in the original text is computed, a verification check is performed to ensure that all characters in the pattern $x[0..m-1]$ match the corresponding substring $y[p..p+m-1]$.

While its worst-case complexity can reach $O(nm)$, necessitating verification of all positions in the text, the algorithm described shows competitive, and often superior, performance compared to other established methods in practical scenarios. The subsequent section will provide supporting evidence in this regard.

4. Experimental Results

In this section, we present the results of an experimental evaluation of the new text sampling approaches introduced in this article. Our evaluation compares our approaches with the best solution available in the literature, focusing on three key metrics: time efficiency, memory space requirements, and the computational effort needed for verifying candidate occurrences.

Algorithms and implementation. We compared the text sampling approach proposed in this paper (CCS_q) for values of $q \in 2, 4, 6$ with the FCDS+ approach which is currently the most effective CDS variant for natural language texts. We recall that FCDS+ is a CDS variant enhanced with fake samples and a mapping table which maps one element every k elements to its corresponding position within the text. For details of the FCDS+ implementation see [17].

In addition, we report that the CCS algorithm has been implemented using an hashing function wich maps any substring of q characters on a 1 byte bucket. For both the CCS and FCDS+

$\textsc{Hash}_2(x)$
$h \leftarrow 0$
$h \leftarrow h + (x[0] << 4)$
$h \leftarrow h + x[1]$
return h

$\textsc{Hash}_4(x)$
$h \leftarrow 0$
$h \leftarrow h + (x[0] << 6)$
$h \leftarrow h + (x[1] << 4)$
$h \leftarrow h + (x[2] << 2)$
$h \leftarrow h + x[3]$
return h

$\textsc{Hash}_6(x)$
$h \leftarrow 0$
for $i \leftarrow 0$ **to** 6 **do**
  $s \leftarrow 6 - i$
  $h \leftarrow h + (x[i] << s)$
return h

Figure 3: The implementations of three hash functions that are used in our implementation of the CCS sampling method. The implementations expect a hash value of 1 byte in size as output. The q values vary in the set $\{2, 4, 6\}$, while the shift values vary in the interval $\{4, 2, 1\}$, respectively.

algorithms, the mapping table utilizes values of $k \in \{8, 16, 32\}$, selecting the optimal execution time for each case. The functions used to calculate the hash value in our implementation are those shown in Figure 3. A hash value of 1 byte was selected to ensure that the space requirements of CCS closely match those of FCDS+, thereby enabling a fair comparison between the two solutions. We compared our algorithms using multiple pivot characters based on the rank of their frequency j.

Testbed. All the algorithms were implemented in the C programming language and tested using the Smart tool[2] [21]. The experiments were executed on a MacBook Pro with a 2.7 GHz Intel Core i7 processor, 4 cores, 16 GB RAM 2133 MHz LPDDR3, 256 KB L2 Cache, and 8 MB L3 Cache. Compilation was performed with the `-O3` optimization flag.

We tested all the algorithms on a text buffer of size 100 MB, sourced from the *Pizza and Chili* dataset [22], which is available for download online. The algorithms were specifically tested using the natural language text from this dataset. In our experiments we limited ourselves to searching for patterns of fixed length $m = 2^p$, with $p \in \{4, 5, 6, 7\}$. For the purpose of our experimental evaluation, 1000 patterns were randomly selected from the text buffer, all algorithms were run to search for each of the patterns in this set, and the average running time was computed over these 1000 runs.

The choice of using a 100 MB text buffer ensures a substantial and representative sample size for evaluating algorithm performance. The natural language text from the *Pizza and Chili* dataset provides a realistic testing scenario, reflecting typical use cases in text processing applications.

Analysis of search times. The analysis of search times is particularly important in this context, as sampled matching solutions are designed to achieve significantly superior performance compared to online solutions, with minimal cost in terms of spatial resources.

Figure 4 reports the time performance obtained in our experimental results, highlighting the average running times of the tested algorithms. The results shows that the new variant proposed in this paper achieves superior performance in terms of time compared to the FCDS+ variant, with improvements ranging from 50% to 70%. As expected, the greatest advantages

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

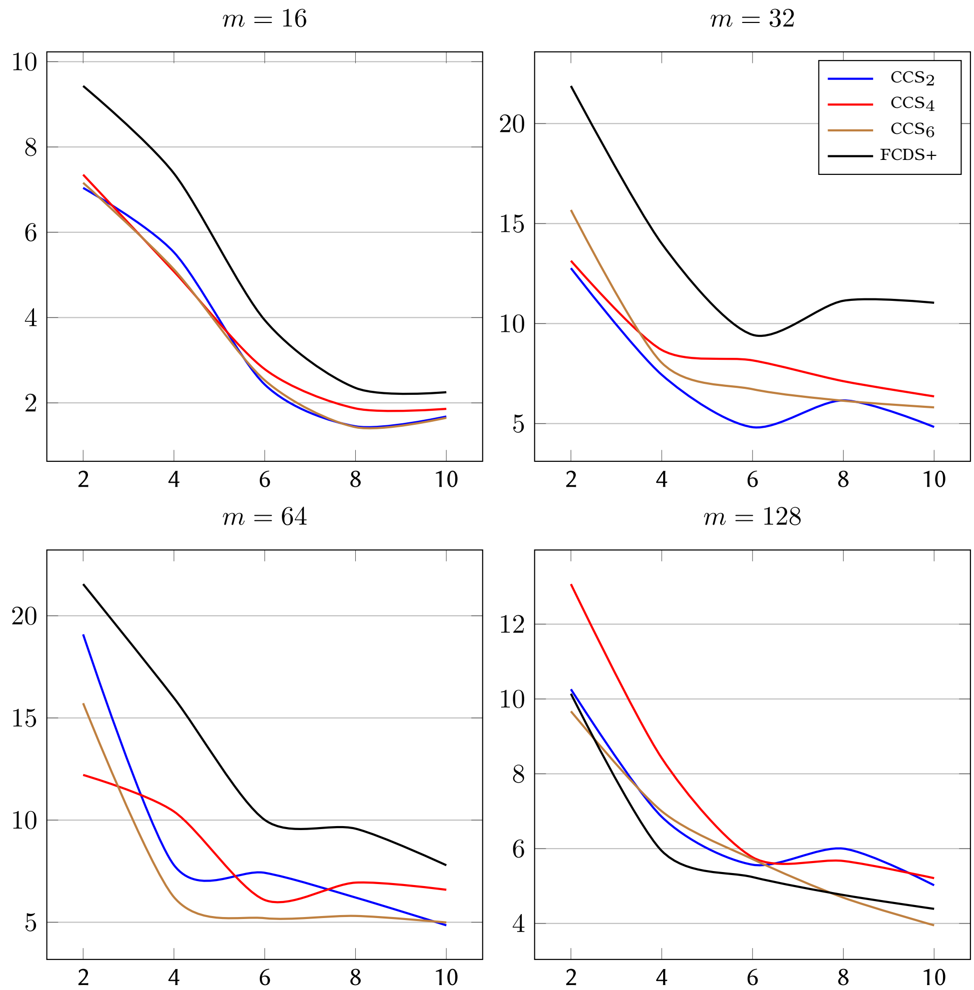

Figure 4: Search time for patterns of size $16 \leq m \leq 128$ comparing the CDS algorithm and the CCS$_q$ algorithm for different values of q. The abscissa shows the rank of the selected pivot character j, while the ordinate shows the time expressed in milliseconds (ms).

are observed for short to medium-length patterns, where context plays a fundamental role in identifying possible occurrences of the pattern.

In our experimental results, a detailed description of the best pivot character selection is not provided. Regarding the selection of the pivot character, superior performance is once again observed with a rank around the value of 8 for both CCS and FCDS+.

Finally, we note that FCDS+ becomes the most efficient solution for very long patterns. In this scenario, the context has less influence on identifying candidate occurrences, and sampling based on the distance between pivot characters proves to be the best strategy.

Table 1 (on the left) presents the experimental results in terms of running times, compared with the search times of the original Horspool algorithm (HOR) [5]. In the table, the execution times of the Horspool algorithm are expressed in milliseconds, while the results for the sampled matching algorithms are expressed in terms of speedup relative to the Horspool algorithm. The best results, in terms of speedup, are highlighted with more intense shades of grey.

As clearly shown in Table 1, the CCS method provides superior speedups compared to FCDS+. These speedups range from an impressive 33 times for short patterns ($m = 16$) to 5 times for very long patterns ($m = 128$). These results shows that solutions based on sampled matching are significantly more effective than those based on the standard approach.

Impact on the number of verifications. In order to provide a more comprehensive understanding of the improvements brought by the proposed new technique, we analyzed the average number of verifications executed by each algorithm under various settings. These settings include different pattern lengths, ranks of the chosen pivot character, and the number of characters used for hashing. The results of this analysis are illustrated in Table 1.

The experimental results show that for small patterns, the CDS algorithm requires, in the worst case, more than $30,000$ verifications, which is significantly more than the actual number of matches (147). In contrast, under optimal conditions, the CCS algorithm requires only 758 verifications, which is less than 2.5% of the original algorithm's verification count.

In the most favorable cases, the new CCS variants significantly reduce the number of verifications to values remarkably close to the actual number of pattern occurrences within the text. Specifically, for $m = 64$, the CCS_4 variant reduces the number of false positives to nearly match the number of actual occurrences (25 versus 13). Furthermore, for $m = 128$, the CCS_6 variant completely eliminates false positives. This demonstrates that the context-based approach manages to drastically reduce the number of verifications required during the search phase, justifying the superior performance in terms of search times discussed previously.

Analysis of the space consumption. Given that the memory required to store the hashing is equivalent to the memory needed to store all the distances using the fake-representation, and that the mapping tables are identical, the newly proposed algorithm eliminates the need for additional fake-samples. Consequently, the CCS algorithm demands fewer memory resources while providing a speed-up in the searching time. Specifically, Figure 5 shows the space consumption of both the FCDS+ and CCS algorithm for different rank of the pivot character, ranging from 2 to 10. We observe that the size of the partial index is always below 10% (corresponding to 10 MB) of the text size and reaches 5% (5 MB) for the rank 8 pivot.

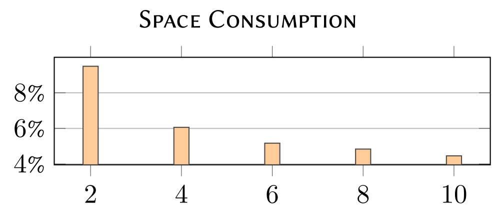

Figure 5: Space consumption of both the FCDS+ algorithm and the CCS algorithm for different rank of the pivot character.

m	j	HOR	FCDS+	CCS_q ($q=2$)	CCS_q ($q=4$)	CCS_q ($q=6$)
16	2	47.39	5.02	6.73	6.44	6.61
	4		6.42	8.56	9.32	9.23
	6		12.02	19.50	16.98	18.73
	8		20.16	32.68	25.34	**33.13**
	10		21.06	28.20	25.47	28.72
32	2	34.82	1.59	2.72	2.64	2.22
	4		2.48	4.68	4.01	4.33
	6		3.68	**7.22**	4.26	5.18
	8		1.18	5.65	4.89	5.67
	10		3.15	7.20	5.47	5.99
64	2	25.94	1.20	1.35	2.12	1.65
	4		1.62	3.32	2.48	4.16
	6		2.59	3.49	4.25	4.98
	8		2.70	4.17	3.73	4.88
	10		3.32	**5.34**	3.93	5.19
128	2	22.32	2.20	2.17	1.70	2.30
	4		3.75	3.25	2.65	3.18
	6		4.25	4.00	3.86	3.90
	8		4.68	3.72	3.93	4.75
	10		5.08	4.44	4.28	**5.65**

	j	$m=16$	$m=32$	$m=64$	$m=128$
Matches	-	147	18	13	11
FCDS+	2	19571	18412	5473	27
	4	30928	75715	2228	173
	6	22778	7702	6470	1040
	8	20235	4986	4487	25215
	10	19571	4099	5007	31603
CCS_2	2	11175	12956	25	13
	4	28514	3390	476	35
	6	3330	7271	461	192
	8	2874	4520	707	350
	10	1957	2467	377	347
CCS_4	2	9292	9832	**25**	13
	4	28403	6487	575	31
	6	**758**	1895	482	191
	8	1548	2990	891	163
	10	1957	780	51	168
CCS_6	2	8215	9069	25	13
	4	28385	1749	277	**11**
	6	1814	6050	443	189
	8	1379	1377	174	194
	10	1957	**512**	31	168

Table 1
(On the left) Experimental results for the Exact String Matching problem. Running times of the HOR are expressed in milliseconds. Results for all other algorithms are expressed in terms of speed-up against the reference HOR algorithm. Over a text of 100MB. (On the right) Number of verifications performed by the sampled matching algorithms during the search phase. The values shown represent the average results obtained over 1000 runs. The first row displays the exact number of pattern occurrences within the text. The subsequent rows show the number of false positives identified by each sampled matching algorithm during the search phase.

5. Conclusions and Future Works

In this paper, we introduced a novel sampling method called Character Context Sampling (CCS), designed to enhance the efficiency of the string matching process. This method tracks the context surrounding each sampled location, rather than just the distances between these locations. Our experimental results demonstrate that CCS significantly reduces the number of verifications required, thereby substantially decreasing search times while maintaining minimal additional space requirements. CCS stands out by outperforming the existing Character Distance Sampling (CDS) method, especially for short patterns, achieving a speedup of between 15% and 40%. This improvement is attributed to the effective use of contextual information, which helps in reducing false positives during the verification phase.

Future research could focus on several areas to further enhance the performance and applicability of the CCS method. First, exploring more efficient hashing techniques and investigating their impact on the speed and accuracy of CCS could yield valuable insights. Second, adapting the CCS method for other types of string matching problems, such as approximate matching or order-preserving matching, could broaden its utility.

Additionally, integrating CCS with other advanced data structures and algorithms, such as suffix trees, may provide hybrid solutions that combine the strengths of different approaches. Finally, optimizing the selection of pivot characters based on specific text characteristics or application requirements could further improve the efficiency of the CCS method.

References

[1] D. E. Knuth, J. H. Morris, Jr., V. R. Pratt, Fast pattern matching in strings, SIAM Journal on Computing 6 (1977) 323–350. URL: https://doi.org/10.1137/0206024. doi:10.1137/0206024. arXiv:https://doi.org/10.1137/0206024.

[2] M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, W. Rytter, Speeding up two string-matching algorithms, Algorithmica 12 (1994) 247–267. URL: https://doi.org/10.1007/BF01185427. doi:10.1007/BF01185427.

[3] S. Faro, T. Lecroq, The exact online string matching problem: A review of the most recent results, ACM Comput. Surv. 45 (2013). URL: https://doi.org/10.1145/2431211.2431212. doi:10.1145/2431211.2431212.

[4] R. S. Boyer, J. S. Moore, A fast string searching algorithm, Commun. ACM 20 (1977) 762–772. URL: https://doi.org/10.1145/359842.359859. doi:10.1145/359842.359859.

[5] R. N. Horspool, Practical fast searching in strings, Software: Practice and Experience 10 (1980) 501–506. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380100608. doi:https://doi.org/10.1002/spe.4380100608. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380100608.

[6] A. Apostolico, The myriad virtues of subword trees, in: A. Apostolico, Z. Galil (Eds.), Combinatorial Algorithms on Words, Springer Berlin Heidelberg, Berlin, Heidelberg, 1985, pp. 85–96.

[7] U. Manber, G. Myers, Suffix arrays: A new method for on-line string searches, SIAM Journal on Computing 22 (1993) 935–948. URL: https://doi.org/10.1137/0222058. doi:10.1137/0222058. arXiv:https://doi.org/10.1137/0222058.

[8] P. Ferragina, G. Manzini, Indexing compressed text, J. ACM 52 (2005) 552–581. URL: https://doi.org/10.1145/1082036.1082039. doi:10.1145/1082036.1082039.

[9] U. Vishkin, Deterministic sampling–a new technique for fast pattern matching, SIAM Journal on Computing 20 (1991) 22–40. URL: https://doi.org/10.1137/0220002. doi:10.1137/0220002. arXiv:https://doi.org/10.1137/0220002.

[10] F. Claude, G. Navarro, H. Peltola, L. Salmela, J. Tarhio, String matching with alphabet sampling, Journal of Discrete Algorithms 11 (2010). doi:10.1016/j.jda.2010.09.004.

[11] S. Faro, F. P. Marino, Reducing time and space in indexed string matching by characters distance text sampling, in: J. Holub, J. Zdárek (Eds.), Prague Stringology Conference 2020, Prague, Czech Republic, August 31 - September 2, 2020, Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2020, pp. 148–159. URL: http://www.stringology.org/event/2020/p13.html.

[12] S. Faro, F. P. Marino, A. Pavone, Efficient online string matching based on characters distance text sampling, Algorithmica 82 (2020) 3390–3412. URL: https://doi.org/10.1007/s00453-020-00732-4. doi:10.1007/S00453-020-00732-4.

[13] S. Faro, F. P. Marino, A. Pavone, Enhancing characters distance text sampling by condensed alphabets, in: C. S. Coen, I. Salvo (Eds.), Proceedings of the 22nd Italian Conference on Theoretical Computer Science, Bologna, Italy, September 13-15, 2021, volume 3072 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2021, pp. 1–15. URL: https://ceur-ws.org/Vol-3072/paper1.pdf.

[14] S. Faro, F. P. Marino, A. Pavone, Improved characters distance sampling for online and

offline text searching, Theor. Comput. Sci. 946 (2023) 113684. URL: https://doi.org/10.1016/j.tcs.2022.12.034. doi:10.1016/J.TCS.2022.12.034.

[15] S. Faro, F. P. Marino, A. Pavone, A. Scardace, Towards an efficient text sampling approach for exact and approximate matching, in: J. Holub, J. Zdárek (Eds.), Prague Stringology Conference 2021, Prague, Czech Republic, August 30-31, 2021, Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2021, pp. 75–89. URL: http://www.stringology.org/event/2021/p07.html.

[16] S. Faro, F. P. Marino, A. Moschetto, A. Pavone, A. Scardace, The Great Textual Hoax: Boosting Sampled String Matching with Fake Samples, in: A. Z. Broder, T. Tamir (Eds.), 12th International Conference on Fun with Algorithms (FUN 2024), volume 291 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2024, pp. 13:1–13:17. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FUN.2024.13. doi:10.4230/LIPIcs.FUN.2024.13.

[17] S. Faro, F. P. Marino, A. Moschetto, Beyond Horspool: A comparative analysis in sampled matching, in: Proceedings of the Prague Stringology Conference 2024, Prague, Czech Republic, Prague Stringology Club, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, 2024.

[18] J. Kim, P. Eades, R. Fleischer, S.-H. Hong, C. S. Iliopoulos, K. Park, S. J. Puglisi, T. Tokuyama, Order-preserving matching, Theoretical Computer Science 525 (2014) 68–79. URL: https://www.sciencedirect.com/science/article/pii/S0304397513007585. doi:https://doi.org/10.1016/j.tcs.2013.10.006, advances in Stringology.

[19] D. Cantone, S. Faro, M. O. Külekci, An efficient skip-search approach to the order-preserving pattern matching problem, 2015, p. 22 – 35. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84978488339&partnerID=40&md5=d5c48fdb1cde28eb746b06b0bee63b35.

[20] T. Lecroq, F. P. Marino, Fast computation of the period and of the shortest cover of a string using its character-distance-sampling representation, 2024. URL: https://arxiv.org/abs/2407.18216. arXiv:2407.18216.

[21] S. Faro, T. Lecroq, S. Borzi, S. D. Mauro, 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. URL: http://www.stringology.org/event/2016/p09.html.

[22] P. Ferragina, G. Navarro, Pizza&Chili, Available online: pizzachili.dcc.uchile.cl/, 2005.