The Great Textual Hoax: Boosting Sampled String Matching with Fake Samples

Simone Faro[1]
Department of Mathematics and Computer Science, University of Catania, Italy

Francesco Pio Marino
Department of Mathematics and Computer Science, University of Catania, Italy
Univ Rouen Normandie, INSA Rouen Normandie, Université Le Havre Normandie, Normandie Univ, LITIS UR 4108, CNRS NormaSTIC FR 3638, IRIB, Rouen, F-76000, France

Andrea Moschetto
Department of Mathematics and Computer Science, University of Catania, Italy

Arianna Pavone
Department of Mathematics and Computer Science, University of Palermo, Italy

Antonio Scardace
Department of Mathematics and Computer Science, University of Catania, Italy

Abstract

Sampled String Matching is presented as an efficient solution to the string matching problem, aiming to tackle the space constraints of indexed string matching while purportedly reducing search times for online solutions. Despite the problem's inception dating back to 1991, practical solutions have only recently emerged. These purportedly accelerate online searches by up to 35 times compared to conventional methods, achieved through a partial index occupying a mere 5% of the text size. This paper delves into the intricacies of one of the latest and most effective text sampling techniques, character distance sampling, which revolves around sampling distances between characters of a selected alphabet within the text. Specifically, we introduce fake samples while remaining honest! In other words, the study reveals that, interestingly, strategically introducing fake samples within the sampled sequence slashes the required index space by almost half, just avoid compromising the algorithm's correctness. Additionally, since efficiency is everything, this approach, in turn, purportedly enhances the algorithm's efficiency under specific conditions.

2012 ACM Subject Classification Theory of computation → Pattern matching; Information systems → Information retrieval

Keywords and phrases string matching, sampling

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.13

Funding *Simone Faro*: 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.
Arianna Pavone: Supported by PNRR project ITSERR - Italian Strengthening of the ESFRI RI RESILIENCE.

1 Introduction

In the intricate world of computer science, the pursuit of pinpointing patterns within a text stands as a formidable challenge, spanning across diverse fields such as natural language processing, information retrieval, and computational biology. This endeavor, dubbed *string*

[1] Corresponding author

matching in academic circles, involves the task of identifying every occurrence of a given pattern x, spanning a length of m, within a text y, stretching to a length of n. Both sequences are crafted from characters drawn from an alphabet Σ of size σ.

While the methods for storing data may vary, textual data remains a stalwart pillar of information storage. This reliance is particularly evident in literature and linguistics, where data take the form of vast corpora and extensive dictionaries. Yet, this reliance extends into the realm of computer science, where vast volumes of data are stored in linear files. Even in the domain of molecular biology, where biological entities are often simplified to sequences of nucleotides or amino acids, the need for swift text-searching solutions persists, propelling researchers to seek ever-faster methodologies.

Applications necessitate two distinct approaches: *online* and *offline* string matching. The former contends with unprocessed text, demanding real-time scrutiny during the search operation. Its worst-case time complexity clocks in at $\Theta(n)$, a milestone initially conquered by the venerable Knuth-Morris-Pratt (KMP) algorithm [16]. However, the holy grail of average time complexity, $\Theta(\frac{n \log_\sigma m}{m})$ [20], finds its manifestation in the Backward-Dawg-Matching (BDM) algorithm [4], a gem in the algorithmic crown.

While legions of string matching algorithms strive for sub-linear performance in practice [5], the Boyer-Moore-Horspool algorithm [2, 13] deserves a standing ovation for its resounding success and the avalanche of subsequent research it has inspired.

On the flip side, solutions embracing the second approach aim to expedite searches through judicious preprocessing, erecting data structures that render search operations a breeze, at least proportional to the pattern's length. Dubbed *indexed searching*, this methodology enjoys a cornucopia of efficient solutions. Notable mentions include those leveraging suffix trees [1], boasting a $O(m + occ)$ worst-case time, suffix arrays [17], offering a respectable $O(m + \log n + occ)$ [17], and the formidable FM-index [12] (Full-text index in Minute space), a compressed titan born of the Burrows-Wheeler transform, deftly balancing input compression with swift substring queries.

But ah, the catch! Despite their dazzling time performance[2], the voracious appetite for space exhibited by full-index data structures, such as suffix-trees and suffix-arrays, dwarfs that of the text itself, ranging from 4 to 20 times its girth. And while the FM-Index appears leaner, often weighing in at less than the text's own bulk, its construction demands almost as much space as a full-index, leaving many practical applications gasping for breath.

Fear not, for the *sampled string matching* emerges as the solution to our space-hungry conundrum, offering a ray of hope amidst the darkness of bloated data structures!

1.1 Sampled String Matching

A more apt remedy for the quandary lies in the realm of *sampled string matching*, first delineated by Vishkin in 1991 [19]. Here, the strategy involves fashioning a succinct sampled version of the text and then applying any online string matching algorithm directly onto this trimmed version. While this technique may unearth potential pattern occurrences more swiftly, the caveat is that each discovery within the sampled text necessitates subsequent validation within the original corpus. Nonetheless, a sampled-text approach flaunts several virtues: firstly, it often boasts ease of implementation, standing head and shoulders above its more labyrinthine counterparts. Secondly, it exhibits a penchant for parsimony, requiring

[2] A fast offline solution's search speed is as swift as a cheetah's sprint, typically under 1 millisecond per query.

only a trifling amount of additional space. Moreover, it's no slouch in the speed department, often zipping through searches. Furthermore, it's not averse to updates, allowing for swift alterations to the underlying data structure.

Besides Vishkin's theoretical breakthrough, a more pragmatic incarnation of sampled string matching has emerged, courtesy of Claude *et al.* [3], leveraging an alphabet reduction technique (OTS). Their brainchild touts an extra space overhead of a mere 14% of the text's size, while clocking in at speeds up to 5 times faster than traditional online string matching algorithms on English texts. It thus earns its stripes as one of the premier solutions for such search approach.

Their ingenuity doesn't stop there. They've also dabbled in indexing the sampled text, concocting a sampled suffix array by indexing the sampled positions of the text. Although reminiscent of the sparse suffix array [14], this variant dances to its own beat, with divergent sampling properties birthing distinct search algorithms and performance metrics.

More recently, Faro *et al.* have injected a dose of innovation into the sampling sphere with their *Character Distance Sampling* (CDS) approach [7–11]. In practical terms, through sampling absolute positions of some specific characters in the text, called *pivot characters*, their method has yielded speedups of up to a factor of 9 on English texts, while demanding a mere pittance of additional space, ranging from 11% to 2.8% of the text's size. This translates to a whopping 50% reduction in search times compared to the previous approach (OTS).

1.2 Our Contribution

In this paper, we propose a variation of the CDS method that enhances it in terms of both space efficiency and search performance. The ingenious tweak involves introducing a set of additional false samples of the pivot characters, amusingly dubbed *fake samples*, which marginally increase the number of elements in the partial index. Paradoxically, this leads to a whopping three-quarters reduction in the overall space required to represent the data structure, all while ensuring algorithmic correctness. Quite a nifty advantage, wouldn't you say, especially considering the significant boost in search performance it offers?

The crux of the new approach lies in storing distances between pivot characters rather than their absolute positions within the text. This allows us to reduce the space used but introduces the problem of direct addressing of positions within the original text. But fear not, for we shall delve into the details in due course giving a solution to this problem.

The structure of the paper unfolds as follows: In Section 2 we briefly summarize the CDS method as it is used to efficiently solve the string matching problem. In Section 3, we lay out the conceptual foundation of the new sampling approach, delineating its costs and benefits. Section 5 presents an empirical analysis comparing the new approach to the standard CDS methodology in terms of space utilization and search time efficiency. Finally, our findings and insights are encapsulated in Section 6.

2 Characters Distance Sampling in Brief

In this section, we embark upon a concise yet comprehensive description of the methodology employed in crafting the partial-index built in the *Character Distance Sampling* (CDS).

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

We select a sub-alphabet $C \subseteq \Sigma$ to serve as the *set of pivot characters.* Using this designated pivots, we sample the text y by calculating the distances between consecutive occurrences of any pivot character $c \in C$ within y. Formally, our sampling methodology is based on the following definition of *position sampling* within a text.

▶ **Definition 1** (Position Sampling). *Let y be a text of length n, let $C \subseteq \Sigma$ be the set of pivot characters and let n_c be the number of occurrences of any $c \in C$ in the input text y.*

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

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

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 2.** Suppose $y =$ "agaacgcagtata" is a sequence of length 13, over the alphabet $\Sigma = \{\text{a,c,g,t}\}$. Let $C = \{\text{a}\}$ 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.

▶ **Definition 3** (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. The* characters distance function *is 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*

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

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

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 [8]).

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 $\dot{y}$ 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 impracticable. 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.* [11] recently introduced the run-length text sampling, tailored for approximate searches. This technique proves well-suited, for instance, for tasks such as *Order Preserving pattern matching* [15].

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 [8].

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.* [9, 10] have proved the efficacy of a technique leveraging condensed alphabets to expand the underlying alphabet size and achieve markedly improved performance.

3 The Cunning Hoax of Fake Samples

Let's point out the paradox of text sampling based on position distance!
While it does indeed seek to reduce the burden of representation by sampling elements, it naively inflates the space required for each element by a whopping factor of 4. Imagine, if you will, the humble character of a text, typically content with a mere 8 bits for its expression, suddenly finding itself encased in the luxurious padding of a 32-bit integer!

In the realm of extreme scenarios, where the sampled positions sprawl to encompass more than a quarter of the input sequence, we encounter a most peculiar predicament. The very act of sampling, intended to lighten the load, paradoxically threatens to devour more memory than the original text itself. Consider, for instance, the vast genomic landscape, where the four noble bases – A, C, G, and T – hold court. Here, the uniform distribution of these alphabetic constituents renders sampling based on position distance a pointless endeavor, yielding a partial index that, although it manages to significantly improve search performance, rivals the text in sheer magnitude of memory consumption. A nice scam!

In this section we present a useful strategy to avoid this unpleasant and somewhat embarrassing situation at a very low cost in almost all practical cases. Although our discussion focuses on the CDS sampling method, the proposed technique is suitable for any distance-based text sampling method between sampled locations where a certain limit is imposed on the representation space of each distance.

To do this, we assume that p_1 and p_2 are two consecutive sampled positions, and we also assume that $d = p_2 - p_1$ is the distance between these two positions. We use the symbol γ to denote the *distance representation bound* (DRB), a value related to the memory space used for the representation of each individual value of the sequence. In this context each value of the sequence of distances is represented using exactly γ bits, so that each distance contained in the sequence should have a value between 0 and $2^\gamma - 1$.

As we have already discussed, this constraint on the representation of distances introduces the problem of not being able to represent all those distances whose value is greater than or equal to the limit 2^γ, making it inapplicable in many practical cases.

Our solution to this problem involves the introduction of a certain additional, arbitrarily large number of samples, which can allow us to decompose a distance $d \geq 2^\gamma$ into a sequence of distances whose value is contained within the limit imposed on us. Since these additional samples are not foreseen in the actual sampling, we will call them *fake samples*, to underline the fact that they are not sampled positions but rather fictitious and therefore incorrect values, introduced exclusively for the purpose of ensuring that all distances can fall within the 2^γ limit that we have imposed for the representation of the sampled text.

To avoid ambiguity in the sequence representation, the decomposition should be non-ambiguous. For this reason, among the many possible alternatives, this work proposes a decomposition that involves the introduction of a false sample, if necessary, every $2^\gamma - 1$ characters of the original text. In other words, if two samples are at a distance $d = p_2 - p_1 \geq 2^\gamma$, we introduce a false sample at position $p'_1 = p_1 + 2^\gamma - 1$ and we operate recursively on the residual interval $p_2 - p'_1$, provided that it is greater than or equal to 2^γ. An interval of d positions between two real samples will then be decomposed, as needed, into a sequence of $\lceil d/2^\gamma \rceil$ intervals of size $2^\gamma - 1$, made exception for the last interval whose size will be equal to $d \mod (2^\gamma)$. The above is achieved through the insertion of fake samples. An example of the Sampling procedure is shown on **Algorithm** 1.

More formally we introduce the following definition of *Fake Distance Decomposition.*

▶ **Definition 5** (Fake Distance Decomposition). *Let d be an integer value representing the distance between two sampled text positions. The fake distance decomposition $[d]_\gamma$ for a given DRB γ is a numeric sequence*

$$[d]_\gamma = \langle d_0, d_1, .., d_{k-1} \rangle$$

such that:

(a) $k = \lceil d/2^\gamma \rceil$*;*

(b) $d_i = 2^\gamma - 1$*, for* $0 \leq i < k - 1$*:*

(c) $d_{k-1} = d \mod (2^\gamma)$*.*

By Definition 5 it turns out that the sum of the distances obtained from a fake distance decomposition $[d]_\gamma$ results in the value of the original distance. Formally we have

$$\begin{aligned}\sum_{u\in[d]_8} u &= \sum_{i=0}^{k-1} d_i \\ &= \sum_{i=0}^{k-2}(2^\gamma - 1) + (d \mod (2^\gamma)) \\ &= (2^\gamma - 1) \times \left\lfloor \frac{d}{2^\gamma} \right\rfloor + (d \mod (2^\gamma)) \\ &= d\end{aligned} \tag{3}$$

Observe also that the fake distance decomposition of a distance value $d < 2^\gamma$ leaves the original sequence unchanged, since $k = \lceil d/2^\gamma \rceil = 1$. This can be expressed by the relationship $[d]_\gamma = \langle d \rangle$, if $d < 2^\gamma$.

▶ **Example 6.** Let $\gamma = 8$ be the distance representation bound which assumes a binary representation of any distance value by using 8 bits, and therefore a maximum representable value equal to $2^\gamma - 1 = 255$. Then, if we take into account the distance values 841, 134, 256 and 255, we have the following fake distance decompositions:

$$\begin{aligned}[841]_8 &= \langle 255, 255, 255, 76 \rangle \\ [134]_8 &= \langle 134 \rangle \\ [265]_8 &= \langle 255, 10 \rangle \\ [255]_8 &= \langle 255 \rangle\end{aligned}$$

The definition of Fake Distance Decomposition given above can be easily extended in order to decompose a distance sequence and make it representable by using γ bits for each individual sample value. We then introduce the following definition of *Faked Distance Sampling.*

▶ **Definition 7** (Fake Distance Sampling). *Let $y = \langle d_0, d_1, .., d_{n-1} \rangle$ be a sequence of distances, of length n, obtained from any kind of distance sampling performed on an input text. The fake distance sampled version of y, with DRB γ, is a numeric sequence, $[y]_\gamma$, obtained by the concatenation of the fake distance decompositions of its values. Formally*

$$[y]_\gamma = [d_0]_\gamma + [d_1]_\gamma + .. + [d_{n-1}]_\gamma$$

▶ **Example 8.** Let $y = \langle 3, 124, 15, 255, 7, 9, 15, 262, 9, 841, 3 \rangle$ be a sequence of distances obtained from any kind of sampling performed on the text y. According to Definition 7 the fake distance sampling of y, using a DRB value equal to 8, is given by

$$[y]_8 = \langle 3, 124, 15, 255, 7, 9, 15, \underline{255, 7}, 9, \underline{255, 255, 255, 76}, 3 \rangle$$

where we underlined the sub-sequences obtained by a fake distance decomposition.

▶ **Example 9.** Let $y = \langle 3, 124, 15, 255, 7, 9, 32, 15, 262, 9, 841, 3 \rangle$ be a sequence of distances obtained from any kind of sampling performed on the text y and let $x = \langle 15, 262, 9 \rangle$ be a sequence of distances obtained from the same kind of sampling performed on the pattern x.

Algorithm 1 Fake-Position-Distance-Sampling($y, n, pivot$).

```
Data: a string $y$, its length $m$ and the selected pivot character
Result: The faked character distance representation of $y$
$\bar{y} \longleftarrow \langle\rangle$
$j \longleftarrow 0$
$prev \leftarrow 0$
for $i \longleftarrow 0$ to $n-1$ do
    if $(y[i] = pivot)$ then
        $a \longleftarrow i - prev$
        while $(a > 255)$ do
            $\bar{y}[j] \longleftarrow 255$
            $j \longleftarrow j + 1$
            $a \longleftarrow a - 255$
        end
        $\bar{y}[j] \longleftarrow a$
        $j \longleftarrow j + 1$
    end
    return $\bar{y}$
end
```

According to Definition 5 the fake sample sequences are as follows:

$$[y]_8 = \langle 3, 124, \underline{15, 255, 7, 9}, 32, \underline{15, 255, 7, 9}, 255, 255, 255, 76, 3\rangle$$
$$[x]_8 = \langle 15, 255, 7, 9\rangle$$

Therefore with this kind of representation, if we intend to search $[x]_8$ into $[y]_8$, we will have two different *verify*.
The first verify will be at position 7, will give negative response, due for the different original distances in fact, the $255, 7$ of the pattern was originally a 262 differently from the one in the text. The second one at the position 12 which can also give positive response depends on the original text and pattern. It's noteworthy to highlight that employing solely the *CDS* representation without employing the *Fake Decomposition*, the number of verification steps would be reduced to just one at position 12. This consideration arises from the fact that the occurrence at position 7 would not match the sampled pattern if it were not decomposed.

3.1 How Much Space Does This Hoax Cost?

In the realm of text sampling, the spatial demand for string matching emerges as a pivotal consideration. It delineates the additional space, relative to the text's size, that a given solution consumes to tackle the task.

Let's consider the partial index derived from our innovative approach: a clever concoction that, despite utilizing more characters than the original technique (refer to Figure 1), owes its efficiency to the inclusion of a certain number of fake samples. Despite the incorporation of these fake samples, our experimental analysis conducted on an English text[3] reveals only a marginal deviation in the sizes of these two indices, maintaining a comparable character count.

[3] In our experiments, we utilized the 100MB dataset of English texts sourced from *Pizza and Chili* [18].

Specifically Figure 1 compares CDS and Faked-CDS approaches. On the left we show the number of elements contained in the two indexes depending on the rank of the pivot character, computed as a percentage of the number of characters of the text on which the indexes are built. We can see how the values are almost identical with very slight variations: this suggests that only a negligible number of samples are added to the original index.

This can be attributed to the fact that only a small fraction of the distances between occurrences of the pivot character exceed the imposed 2^γ threshold. This observation is confirmed in Figure 1, on the right, which shows the percentage of fake samples added in the construction of the partial index in the Faked-CDS approach. As a percentage, the addition of false samples we are compelled to include rises with the rank of the pivot character used in index construction, albeit remaining negligible.

It is very interesting to observe how the pivot of rank 14, which is identified by the character `\s` (the blank space) in our dataset, reduces the number of fake samples to 0. This is due to the fact that this character has a more uniform distribution within the text and this means that two occurrences of the same character are never more than 256 characters apart.

However, the devil lies in the details! Upon closer examination of the space required by these data structures, a significant contrast becomes apparent. While the previous index allocates 32 bits for each element, our method showcases newfound parsimony, utilizing a mere 8 bits for distance representation. Behold the marvel: a nearly fourfold reduction in effective space compared to its predecessor.

To highlight this aspect, Figure 2 shows the additional amount of space required for storing the two data structures, computed as a percentage of the space required for storing the text on which the indexes are built. As expected, the space required by the Faked-CDS approach is almost four times less than that required by the standard CDS approach.

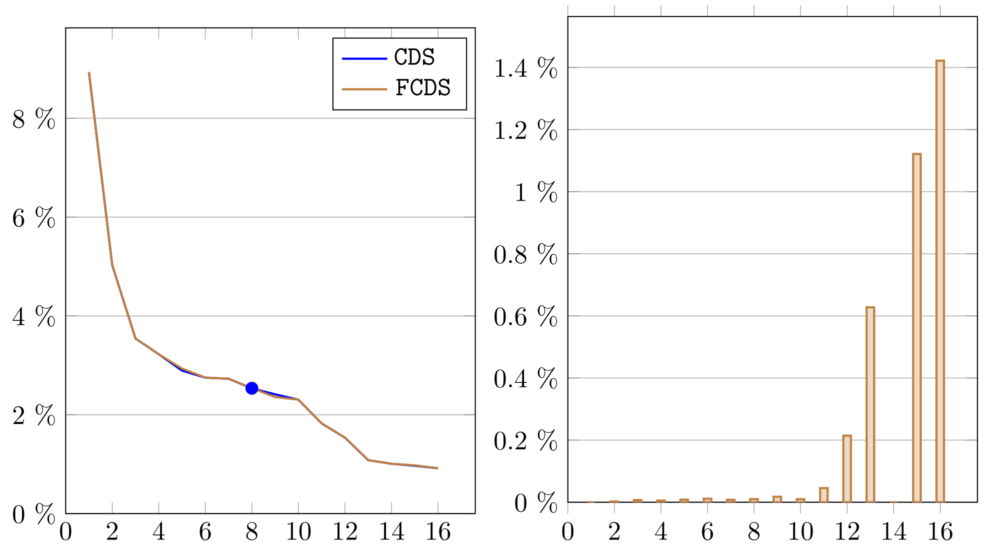

Figure 1 CDS and Faked-CDS approaches compared. On the left, the number of elements contained in the two indexes, calculated as a percentage of the number of characters of the text on which the indexes are built. On the right, the percentage of fake samples added in the construction of the partial index in the Faked-CDS approach, calculated relative to the number of characters in the text. In both cases, the x-axis identifies the rank of the pivot character used to construct the two indices.

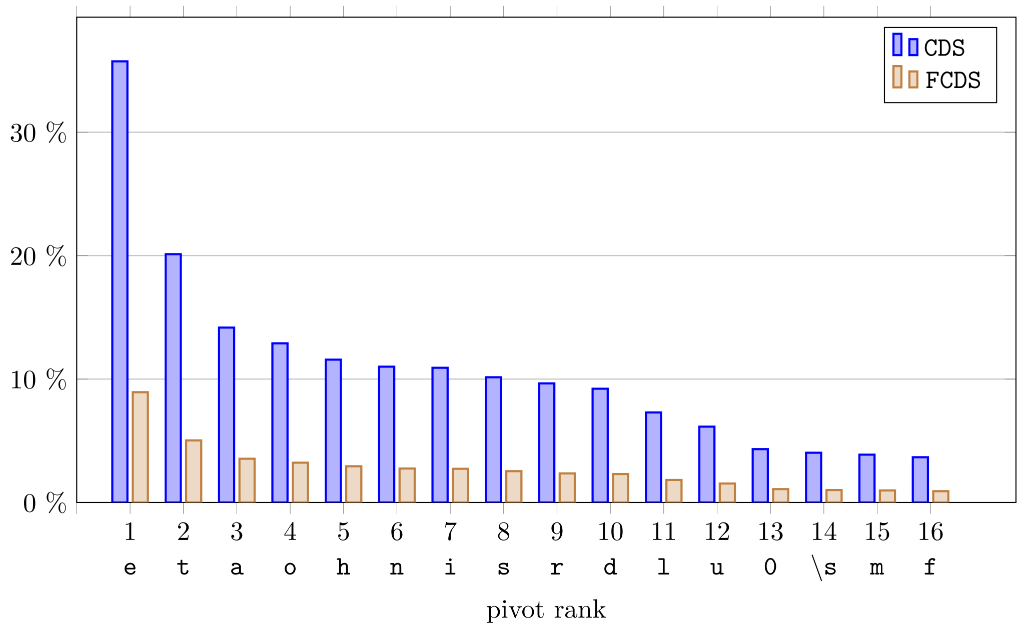

Figure 2 CDS and Faked-CDS approaches compared in terms of additional amount of space required for storing the two data structures. The percentage is computed as a relative to the space required for storing the text on which the indexes are built. The x-axis identifies the rank of the character used as a pivot in the construction of the index. The corresponding character for each rank is showed below each rank (`\s` is the space charcater).

Ultimately, the new approach manages to reduce the space required to store the partial index by a factor ranging from 69% to 73% with respect to the standard CDS approach, despite utilizing more samples in its representation. Quite a feat, considering it also significantly enhances search performance (see Section 5 for details). Specifically, employing the pivot of rank 8, which yields the best search time performance [8], the new approach yields an index occupying only 2.5% of the text size, compared to the standard CDS method's 10%.

In Section 5, we will delve into an analysis of performance, examining both time and space metrics.

4 The Details of the Search Algorithm

The search phase in a sampling-based searching approach entails the straightforward application of any exact string matching algorithm. Its task? To ferret out all instances of the sampled pattern lurking within the sampled text. For each of these candidate occurrences, a thorough verification is then undertaken within the text. The goal? To sift through the text and discern whether the candidate occurrence is a real occurrence or merely a cunning impostor.

In the conventional rendition of the CDS approach, an additional amount of space was allocated to ensure a direct mapping of each pivot character's position within the index to its corresponding position within the original text. In simpler terms, every index element $\dot{y}[i]$ denotes the position of the i-th pivot character within the text, facilitating swift navigation from the sampled text. Consequently, upon identifying a candidate occurrence, one could swiftly access the corresponding text position for verification, requiring only $O(m)$ time.

Algorithm 2 Search-FCDS($x, m, y, n, \bar{x}, \bar{m}, \bar{y}, \bar{n}$).

```
Data: a pattern $x$ of length $m$, a text $y$ of length $n$ and their faked decomposition
Result: the number of occurrences of $x$ in $y$
$hbc \leftarrow []$
for $i \leftarrow 0$ to $\Sigma$ do
    $hbc[i] \leftarrow m$
end
for $i \leftarrow 0$ to $\bar{m}$ do
    $hbc[\bar{x}[i]] \leftarrow \bar{m} - i - 1$
end
$s \leftarrow 0$
$count \leftarrow 0$
$last\_position \leftarrow 0$
$last\_position\_index \leftarrow 0$
while $while(s \leq \bar{n} - \bar{m})$ do
    $i \leftarrow 1$
    while $i < \bar{m}$ and $\bar{x}[i-1] == \bar{y}[s+i-1]$ do
        $i = i + 1$
    end
    if $i == \bar{m}$ then
        $position \leftarrow s$
        while $position > last\_position\_index$ do
            $last\_position = last\_position + \bar{y}[position]$
            $position = position - 1$
        end
        $count = count + verify(x, m, y, last\_position)$
    end
    $s = s + hbc[\bar{y}[s + \bar{m} - 1]]$
    return count
end
```

However, the advent of the novel sampling representation embraced by the Fake Decomposition approach alters this dynamic. This approach opts to store the distances $\bar{y}[i]$ between consecutive occurrences of pivot characters rather than their positions, thereby relinquishing the explicit mapping of pivot character positions. Consequently, the dilemma arises: how does one access the corresponding text position to verify each candidate occurrence?

By working with the distances between consecutive positions of pivot characters, we can derive the mapping $\dot{y}[i]$ of the i-th character of the index using the following formula:

$$\dot{y}[i] = \sum_{j=0}^{i} \bar{y}[j]$$

While this approach theoretically lends itself to direct application through a linear index scan algorithm, such as the renowned KMP algorithm [16], let's not be too hasty in celebrating its efficacy. Assuming we've computed the mapping for all positions less than i, accessing the i-th position of the index theoretically enables us to compute the direct mapping of the i-th pivot character using the formula $\dot{y}[i] = \dot{y}[i-1] + \bar{y}[i]$.

However, despite its apparent simplicity and linear execution time, let's not overlook the practical performance implications. The most efficient string matching approaches are those employing forward jumps [5], allowing for the avoidance of scanning extensive text portions. The Boyer-Moore [2] and Horspool [13] algorithms are among the most representative elements of this family. Yet, such approaches are unable to update the mapping on the text in constant time. Oh, the joys of theoretical elegance versus practical pragmatism!

In this study, we scrutinize two potential remedies for this dilemma:

- The first approach, shown in Figure 2, entails retracing the mapping steps, starting from the last checked position. Put simply, we retain the position v of the last index position from which a text check was conducted. Assuming we've already established a direct mapping for that position, we proceed to scan all text positions from the v-th to the i-th to determine the direct mapping for the new verification position i. This method boasts swift indexing during the search phase, as mapping concerns are deferred until a candidate occurrence surfaces. However, its Achilles' heel lies in the verification phase, which could prove arduous if the last verification occurred significantly earlier in the text. We will refer to this solution as Faked-CDS (FCDS).
- In our second approach, shown in Algorithm 3, we aimed to integrate a technique reminiscent of the one used in the *OTS* approach [3]. Much like their method, we chose to periodically link elements in the sampled text to their positions in the original text. This deliberate choice was made to simplify the backtracking process, sparing us the effort of computing positions by summing distances. Furthermore, it enables quick identification of a character's position, assuming it has already been mapped. Consequently, computing positions for unmapped characters is noticeably accelerated in real-world scenarios.
  In a more formal sense, we define a parameter $k \geq 1$ and allocate an array ρ of $\lceil n/k \rceil$ positions, serving as a link between the sampled elements of the partial index and their actual positions in the text. Specifically, within the partial index, one element is sampled every k, totaling $\lceil n/k \rceil$ elements. Consequently, the additional space required amounts to $4 \times \lceil n/k \rceil$ bytes. When verifying position i with v as the index of the last verified position, it becomes necessary to backtrack the mapping, commencing from the larger value between the mapping of v and $\rho[\lfloor i/k \rfloor]$.
  The advantage of this second approach lies in its ability to circumvent the need for extensive backtracking through text, thereby reducing verification times. Of course, the downside is the inevitable consumption of extra memory to accommodate the mapping array. We will refer to this solution as Faked-CDS + Mapping (FCDS+).

5 Experimental Evaluation

In this section, we present experimental results in order to evaluate the performances of the sampling approaches presented in this paper.

In particular, we tested the original CDS approach (`CDS`), the Faked-CDS approach (`FCDS`) and the Faked-CDS enhanced with mapping (`FCDS+`) approach. The latter has been implemented using values of k in the set $\{8, 16, 32, 64\}$. However, the experimental results for these 4 variants are essentially identical, so we will simply unify the values of the 4 variants. Moreover we choose to set $\gamma = 8$ to conduct our experiments, where we remember that γ is the number of bits we use to represent each single element of the sampled text.

In all three instances, we opted for the pivot character ranked 8 within the text, identified as the letter `s`, as depicted in Figure 2. This particular choice was made based on its observed superior performance in terms of execution times, as detailed in [8].

Algorithm 3 Search-FCDS+$(x, m, y, n, \bar{x}, \bar{m}, \bar{y}, \bar{n}, \rho, k)$.

```
Data: a pattern $x$ of length $m$, a text $y$ of length $n$, their sampled versions and the
      mapping array
Result: the number of matches
$hbc \leftarrow []$
for $i \leftarrow 0$ to $\Sigma$ do
    $hbc[i] \leftarrow m$
end
for $i \leftarrow 0$ to $\bar{m}$ do
    $hbc[\bar{x}[i]] \leftarrow \bar{m} - i - 1$
end
$s \leftarrow 0$
$count \leftarrow 0$
while $while(s \leq \bar{n} - \bar{m})$ do
    $i \leftarrow 1$
    while $i < \bar{m}$ and $\bar{x}[i-1] == \bar{y}[s+i-1]$ do
        $i = i + 1$
    end
    if $i == \bar{m}$ then
        $position \leftarrow \rho[s/k]$
        $idx \leftarrow s - (s\%k)$
        while $idx < s$ do
            $position = position + \bar{y}[idx]$
            $idx = idx + 1$
        end
        $count = count + verify(x, m, y, position)$
    end
    $s = s + hbc[\bar{y}[s + \bar{m} - 1]]$
    return count
end
```

The algorithms have been implemented using the C programming language, and have been tested using the SMART tool [6] and executed locally 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.[4] During the compilation we use the -O3 optimization option.

Comparisons were conducted in terms of search times. For our experiments, we utilized a 100MB dataset of English texts sourced from *Pizza and Chili* [18]. We employed various pattern sizes ranging from 2^7 to 2^{12}. The space used to maintain the partial indexes and any mapping arrays is shown in the following table.

Name	Index Size	Mapping Size	Samples	Fake Samples
CDS	10.14 MB	-	2536790	-
FCDS	2.53 MB	-	2537048	258
FCDS+	2.53 MB	1.26 MB	2537048	258

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

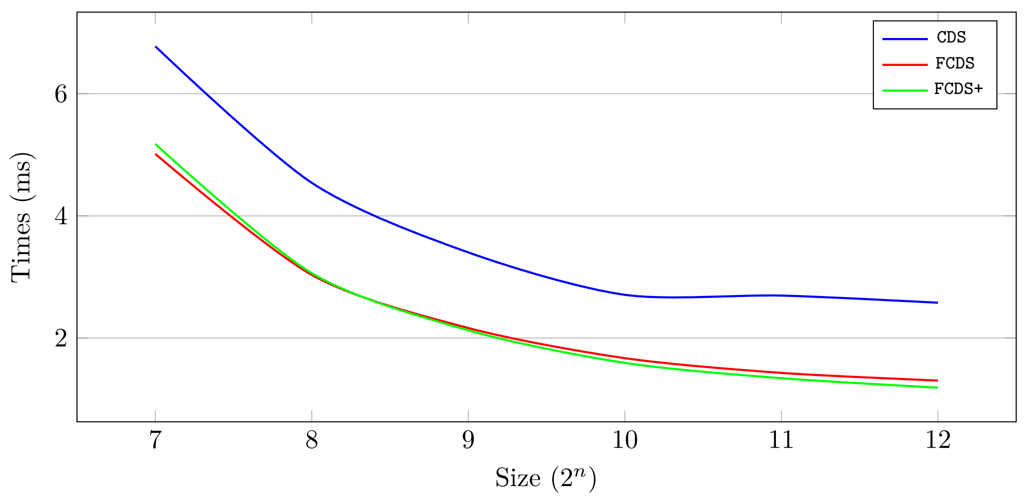

Figure 3 Running times of three sampled string matching algorithms for searching a 100 MB English text and averaging over 1000 different runs.

For each sequence in the dataset, we randomly selected 1000 patterns extracted from the text and computed the average running time over these 1000 runs.

As illustrated in Figure 3, despite utilizing less space, as demonstrated in previous sections, the *Fake Decomposition* method achieves superior performance in practice, particularly in terms of searching time.

According to experimental results, it turns out that the newly proposed methods (`FCDS` and `FCDS+`) allows a speed-up ranging between 33.7% and 55.1%. Despite the increase in the size of the partial-index with the new method, several factors contribute to the observed improvement in performance. Firstly, the enlarged size of the sampled pattern, a known accelerator in classical string matching, likely plays a role, as exact string matching algorithms generally perform better with larger patterns. Secondly, the Character Distance Sampling algorithms, while susceptible to performance degradation in scenarios with fewer than 2 distances represented in the sampled pattern, benefit from the increased number of characters required for the representation, thereby averting their worst-case scenario. Another notable speed-up is observed in the searching phase, which no longer requires the computation of distances by subtracting consecutive positions; rather, each item in our sampled text already represents a distance itself.

While FCDS consistently demonstrates a space reduction of approximately 75%, `FCDS+` utilizes an additional data structure to accommodate the position mapping, resulting in a space reduction ranging from 62.5% (if the position mapping is stored every 8 different characters) to 72% (with a mapping value of 32).

It is noteworthy that while both the `FCDS` and `FCDS+` approaches demonstrate comparable performances, nuanced disparities are discernible. Notably, `FCDS` exhibits marginally superior efficacy for short patterns, whereas `FCDS+` surpasses as pattern length extends. The rationale underlying this phenomenon lies in the examination of candidate occurrences identified by the search within the partial index.

For short patterns, the proliferation of candidate occurrences escalates, rendering the verification times in `FCDS` easily justifiable. The proximity of consecutive checks minimizes the impact of backtracking on search efficiency. Consequently, the inclusion of an additional mapping table mildly penalizes the `FCDS+` methodology.

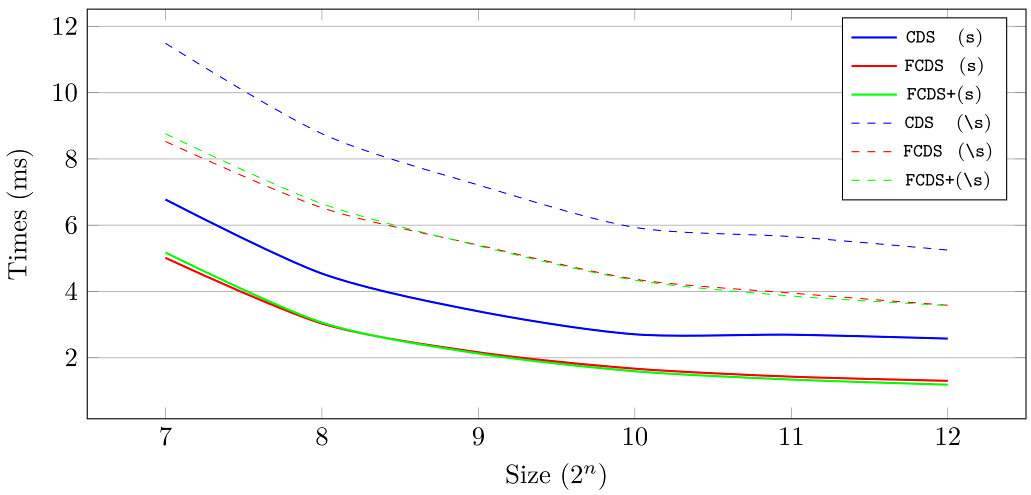

Figure 4 Performances of the algorithms with a 100 MB English text averaging over 1000 different executions. Solid lines indicate running times obtained with a partial index constructed using the 8-th ranked pivot character, specifically representing the character 's'. Dashed lines indicate running times obtained with a partial index constructed using the the 14-th pivot character, representing the blank space character '\s', which generates no fake samples.

Conversely, with longer patterns, the candidate occurrences diminish significantly, exacerbating the performance degradation induced by backtracking in the FCDS approach. Herein lies the challenge: reconstructing the occurrence's position from distant points within the text. In contrast, FCDS+ capitalizes on the mapping table, curtailing backtracking distances substantially.

Lastly, we venture into a comparative analysis of the two methodologies delineated in this study, particularly examining scenarios where distinct pivot characters are employed. A shared observation emerges regarding the potential rationale for adopting the "blank space" character (ranked 14 within the text) as a pivot. Notably, its utilization within the FCDS approach obviates the generation of fake samples. One might conjecture that leveraging this character as a pivot could be pragmatically advantageous when performance metrics are comparable, owing to its minimal impact on partial index construction.

Regrettably, the time performances are far from commensurate, as elucidated in Figure 4. It becomes apparent that sampling predicated on the rank 8 character yields a notable 40% to 50% reduction in processing times compared to its "blank space" counterpart.

6 Conclusions

We introduced a novel method for storing sampled versions of text and patterns based on character distance sampling. Our approach involved an initial analysis of a new decomposition method, which significantly reduces the space requirements for each occurrence of any pivot character. Subsequently, we developed two distinct searching processes aimed at outperforming the original *Character Distance Sampling* (CDS) algorithm. Our algorithms demonstrate improved efficiency compared to the previous method by a factor ranging between 33.7% and 55.1%, while achieving a space reduction ranging from 63% to 75%.

Given the promising results achieved with this new technique, it would be interesting to apply it to other derived versions of the *CDS*, such as the offline and condensed variants. It would also be intriguing to investigate how the *CDS* algorithm performs when combined with other online string matching algorithms, to explore the possibility of achieving new and interesting performance improvements. We intend to explore these directions in our future work.

References

1 Alberto Apostolico. The myriad virtues of subword trees. In Alberto Apostolico and Zvi Galil, editors, *Combinatorial Algorithms on Words*, pages 85–96, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.

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

3 Francisco Claude Faust, Gonzalo Navarro, Hannu Peltola, Leena Salmela, and Jorma Tarhio. String matching with alphabet sampling. *Journal of Discrete Algorithms*, 11, December 2010. `doi:10.1016/j.jda.2010.09.004`.

4 M. Crochemore. Speeding up two string-matching algorithms. *Algorithmica*, 12(4):247–267, 1994. `doi:10.1007/BF01185427`.

5 Simone Faro and Thierry Lecroq. The exact online string matching problem: A review of the most recent results. *ACM Comput. Surv.*, 45(2), March 2013. `doi:10.1145/2431211.2431212`.

6 Simone Faro, Thierry Lecroq, Stefano Borzi, Simone Di Mauro, and Alessandro Maggio. The string matching algorithms research tool. In *Proceedings of the Prague Stringology Conference 2016*, pages 99–111. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2016. URL: `http://www.stringology.org/event/2016/p09.html`.

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

8 Simone Faro, Francesco Pio Marino, and Arianna Pavone. Efficient online string matching based on characters distance text sampling. *Algorithmica*, 82(11):3390–3412, 2020. `doi:10.1007/S00453-020-00732-4`.

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

10 Simone Faro, Francesco Pio Marino, and Arianna Pavone. Improved characters distance sampling for online and offline text searching. *Theor. Comput. Sci.*, 946:113684, 2023. `doi:10.1016/J.TCS.2022.12.034`.

11 Simone Faro, Francesco Pio Marino, Arianna Pavone, and Antonio Scardace. Towards an efficient text sampling approach for exact and approximate matching. In Jan Holub and Jan Zdárek, editors, *Prague Stringology Conference 2021, Prague, Czech Republic, August 30-31, 2021*, pages 75–89. Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2021. URL: `http://www.stringology.org/event/2021/p07.html`.

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

13 R. Nigel Horspool. Practical fast searching in strings. *Software: Practice and Experience*, 10(6):501–506, 1980. `doi:10.1002/spe.4380100608`.

14 Juha Kärkkäinen and Esko Ukkonen. Sparse suffix trees. In Jin-Yi Cai and Chak Kuen Wong, editors, *Computing and Combinatorics*, pages 219–230, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

15 Jinil Kim, Peter Eades, Rudolf Fleischer, Seok-Hee Hong, Costas S. Iliopoulos, Kunsoo Park, Simon J. Puglisi, and Takeshi Tokuyama. Order-preserving matching. *Theoretical Computer Science*, 525:68–79, 2014. Advances in Stringology. doi:10.1016/j.tcs.2013.10.006.

16 Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977. doi:10.1137/0206024.

17 Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. doi:10.1137/0222058.

18 Gonzalo Navarro Paolo Ferrigna. *Pizza&Chili*. Available online: pizzachili.dcc.uchile.cl/, 2005.

19 Uzi Vishkin. Deterministic sampling–a new technique for fast pattern matching. *SIAM Journal on Computing*, 20(1):22–40, 1991. doi:10.1137/0220002.

20 Andrew Chi-Chih Yao. The complexity of pattern matching for a random string. *SIAM Journal on Computing*, 8(3):368–387, 1979. doi:10.1137/0208029.