Towards an Efficient Text Sampling Approach for Exact and Approximate Matching

Simone Faro[1], Francesco Pio Marino[1], Arianna Pavone[2], and Antonio Scardace[1]

[1] Dipartimento di Matematica e Informatica,
Università di Catania, viale A. Doria n. 6, 95125, Catania, Italia
[2] Dipartimento di Scienze Cognitive,
Università di Messina, via Concezione n.6, 98122, Messina, Italia

Abstract. Text-sampling is an efficient approach for the string matching problem recently introduced in order to overcome the prohibitive space requirements of indexed matching, on the one hand, and drastically reduce searching time for the online solutions, on the other hand. Known solutions to sampled string matching are very efficient in practical cases being able to improve standard online string matching algorithms up to 99.6% using less than 1% of the original text size. However at present text sampling is designed to work only in the case of exact string matching.
In this paper we present some preliminary results obtained in the attempt to extend sampled-string matching to the general case of approximate string matching. Specifically we introduce a new sampling approach which turns out to be suitable for both exact and approximate matching and evaluate it in the context of a specific case of approximate matching, the order preserving pattern matching problem.
Our preliminary experimental results show that the new approach is extremely competitive both in terms of space and running time, and for both approximate and exact matching. We also discuss the applicability of the new approach to different approximate string matching problems.

1 Introduction

String matching, in both its *exact* and *approximate* form, is a fundamental problem in computer science and in the wide domain of text processing. It consists in finding all the occurrences of a given pattern x, of length m, in a large text y, of length n, where both sequences are composed by characters drawn from the same alphabet Σ.

As the size of data increases the space needed to store it is constantly increasing too, for this reasons the need for new efficient approaches to the problem capable of significantly improving the performance of existing algorithms by limiting the space used to achieve this as much as possible.

In this paper it is assumed that the text is a sequence of elements taken from a set on which a relation of total order is defined. In general we will try to simplify the discussion by assuming that the text is a sequence of numbers. Such a situation can be assumed for many practical applications since even a character can often be interpreted as a number.

Applications require two kinds of solutions: *online* and *offline* string matching. Solutions based on the first approach assume that the text is not pre-processed and thus they need to scan the input sequence *online*, when searching. Their worst case time complexity is $\Theta(n)$, and was achieved for the first time by the well known Knuth-Morris-Pratt (KMP) algorithm [19], while the optimal average time complexity of the problem is $\Theta(n \log_\sigma m/m)$ [24], achieved for example by the Backward-Dawg-Matching algorithm [9]. Many string matching solutions have been also developed

Simone Faro, Francesco Pio Marino, Arianna Pavone, Antonio Scardace: Towards an Efficient Text Sampling Approach for Exact and Approximate Matching, pp. 75–89.
Proceedings of PSC 2021, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06869-4

in order to obtain sub-linear performance in practice [11]. Among them the Boyer-Moore-Horspool algorithm [3,17] deserves a special mention, since it has inspired much work. Memory requirements of this class of algorithms are very low and generally limited to a precomputed table of size $O(m\sigma)$ or $O(\sigma^2)$ [11]. However their searching time is always proportional to the length of the text and thus their performances may stay poor in many practical cases, especially for huge texts and short patterns.

Solutions based on the second approach try to drastically speed up searching by preprocessing the text and building a data structure that allows searching in time proportional to the length of the pattern. This method is called *indexed searching* [20,16]. However, despite their optimal time performances, space requirements of such data structures are from 4 to 20 times the size of the text, which may be too large for many practical applications.

Leaving aside other different approaches, like those based on *compressed string matching* [21,4], an effective alternative solution to the problem is *sampled string matching*, introduced in 1991 by Vishkin [23]. It consists in the construction of a succinct sampled version of the text (which must be maintained together with the original text) and in the application of an online searching procedure directly on the sampled sequence which acts as a filter method in order to limit the search only on a limited set of candidate occurrences. Although any candidate occurrence of the pattern may be found more efficiently, the drawback of this approach is that any occurrence reported in the sampled-text requires to be verified in the original text. Apart from this point a sampled-text approach may have a lot of good features: it may be easy to implement if compared with other succint matching approaches, it may require very small extra space and may allow fast searching. Additionally it may also allow fast updates of the data structure.

The first practical solution to sampled string matching has been introduced by Claude *et al.* [8] and is based on an alphabet reduction. Their solution has an extra space requirement which is only 14% of text size and turns out to be up to 5 times faster than standard online string matching on English texts. In this paper we refer to this algorithm as Occurrence Text Sampling (OTS).

More recently Faro *et al.* presented a more effective sampling approach based on *character distance sampling* [14,13] (CDS), obtaining in practice a speed up by a factor of up to 9 on English texts, using limited additional space whose amount goes from 11% to 2.8% of the text size, with a gain in searching time up to 50% if compared against the previous solution.

1.1 Our Results and organization of the paper

Known solutions to sampled-string matching prove to work efficiently only in the case of natural language texts or, in general, when searching on input sequences over large alphabets, while their performances degrade when the size of the underlying alphabets decreases. In addition they have been designed to work for solving the exact string matching problem, being inflexible in case they have to be applied to approximate string matching problems.

In this paper we present a new text sampling technique, called Monotonic Run Length Scaling, based on the length of the monotonic sub-sequences formed by the characters of the text when the latter is made up of elements of a finite and totally ordered alphabet. The new technique is original and turns out to be very flexible for its application in both exact and approximate matching. Specifically we also present

a preliminary evaluation of the technique in the case of exact string matching (ESM) and order preserving pattern matching (OPPM), as a case study for the approximate pattern matching.

In the second part of the paper we improve the new approach in practice by proposing a further technique called Monotonic Run Length Sampling and based on the sampling of the lengths of the monotonic sequences in the input text.

From our experimental results it turns out that the new approach, although still in a preliminary phase of formalization and in an early implementation stage, is particularly efficient and flexible in its application, obtaining results that improve up to 12 times standard solutions for the exact string matching problem and up to 40 times known solutions for order preserving pattern matching problem.

The paper is organized as follows. In Section 2 we introduce the Monotonic Run Length Scaling while in Section 3 we introduce the Monotonic Run Length Sampling. In both cases we present a first naïve algorithm for searching a text using the new proposed partial indexes. Then we present our preliminary experimental evaluation in Section 4 testing our proposed sampling approach in terms of space consumption and running times for both exact strung matching and order preserving pattern matching. Finally we draw our conclusions and discuss some future works in Section 5.

2 Monotonic Run Length Scaling

Definition 1 (Monotonic Run). *Let y be a text of length n over a finite and totally ordered alphabet Σ of size σ. A Monotonic Increasing Run of y is a not extendable sub-sequence w of y whose elements are arranged in increasing order. Formally, if $w = y[i..i+k-1]$ is a monotonic increasing run of y of length k, we have:*

- $w[j-1] < w[j]$, *for each* $0 < j < k$;
- $y[i] \leq y[i-1]$;
- $y[i+k-1] \geq y[i+k]$.

Symmetrically a Monotonic Non-Increasing Run of y is a not extendable sub-sequence w of y whose elements are arranged in non-increasing order. Formally, if $w = y[i..i+k-1]$ is a monotonic non-increasing run of y of length k, we have:

- $w[j-1] \geq w[j]$, *for each* $0 < j < k$;
- $y[i] > y[i-1]$;
- $y[i+k-1] < y[i+k]$.

We will indicate with the general term Monotonic Run any sub-sequence of y that can be both monotonic increasing and monotonic non-increasing.

By definition two adjacent monotonic sub-sequences of a string have a single overlapping character, i.e. the rightmost character of the first sub-sequence is also the leftmost character of the second sub-sequence.

Example 2. Let $y = \langle 4, 5, 11, 7, 6, 6, 12, 12, 2, 9, 8, 6, 7, 10, 13\rangle$ be a numeric sequence of length 15. We can identify the following monotonic runs in y: $y[0..2] = \langle 4, 5, 11\rangle$ is a monotonic increasing run; $y[2..5] = \langle 11, 7, 6, 6\rangle$ is a monotonic non-increasing run; $y[5..6] = \langle 6, 12\rangle$ is a monotonic increasing run; $y[6..8] = \langle 12, 12, 2\rangle$ is a monotonic non-increasing run; $y[8..9] = \langle 2, 9\rangle$ is a monotonic increasing run; $y[9..11] = \langle 9, 8, 6\rangle$ is a monotonic non-increasing run; finally, $y[11..14] = \langle 6, 7, 10, 13\rangle$ is a monotonic increasing run.

Function 1: Monotonic-Run-Length-Scaling(x, m)

```
Data: a string x of length m
Result: The Scaled version x̃ of x
x̃ ⟵ ⟨⟩;
μ ⟵ 1;
d ⟵ 1;
if (x[1] − x[0] ≤ 0) then d ⟵ 1 ;
i ⟵ 2;
while (i < n) do
    if ((d = 1 and x[i] − x[i − 1] > 0) or (d = −1 and x[i] − x[i − 1] ≤ 0)) then
        i ⟵ i + 1;
        μ ⟵ μ + 1;
    else
        x̃ ⟵ x̃ + ⟨μ⟩;
        μ ⟵ 0;
        d ⟵ d × −1;
    end
end
x̃ ⟵ x̃ + ⟨μ⟩;
return x̃
```

We now define the process of *Monotonic Run Length Scaling* (MRLX) of a string y, which consists in decomposing the string in a set of adjacent monotonic runs. The resulting sequence, which we call *monotonic run length scaled version* of y, is the numeric sequence of the lengths of the monotonic runs given by the MRLX process.

Definition 3 (Monotonic Run Length Scaling). *Let y be a text of length n over a finite and totally ordered alphabet Σ of size σ. Let $\langle \rho_0, \rho_1, \ldots, \rho_{k-1} \rangle$ the sequence of adjacent monotonic runs of y such that ρ_0 starts at position 0 of y and ρ_{k-1} ends at position $n-1$ of y. The monotonic run length scaled version of y, indicated by $\widetilde{y}$, is a numeric sequence, defined as $\widetilde{y} = \langle |\rho_0|, |\rho_1|, .., |\rho_{k-1}| \rangle$.*

It is straightforward to prove that there exists only a unique monotonic run length scaled version of a given string y. In addition we observe that

$$\left[\sum_{i=0}^{k-1} |\rho_i| \right] - k + 1 = n$$

Example 4. As in Example 2, let $y = \langle 4, 5, 11, 7, 6, 6, 12, 12, 2, 9, 8, 6, 7, 10, 13 \rangle$ be a numeric sequence of length 15. Then the scaled version of y is the sequence $\widetilde{y} = \langle 3, 4, 2, 3, 2, 3, 4 \rangle$ of length 7. We have therefore that $3+4+2+3+2+3+4-7+1 = 15$.

Function 1 depicts the pseudo-code of the algorithm which computes the MRL scaled version of an input string x of length m. It constructs the sequence $\widetilde{x}$ incrementally by scanning the input string x character by character, proceeding from left to right. It is straightforward to prove that its time complexity is $\mathcal{O}(m)$.

Figure 1 shows the average and maximal length of a monotonic run on a random text over an alphabet of size 2^δ, with $2 \leq \delta \leq 8$, and where the ordinates show the length values while the ordinates show the values of δ.

It is easy to observe that the length of each monotonic run (whose value is in any case bounded at the top by the size of the alphabet) never exceeds ten characters.

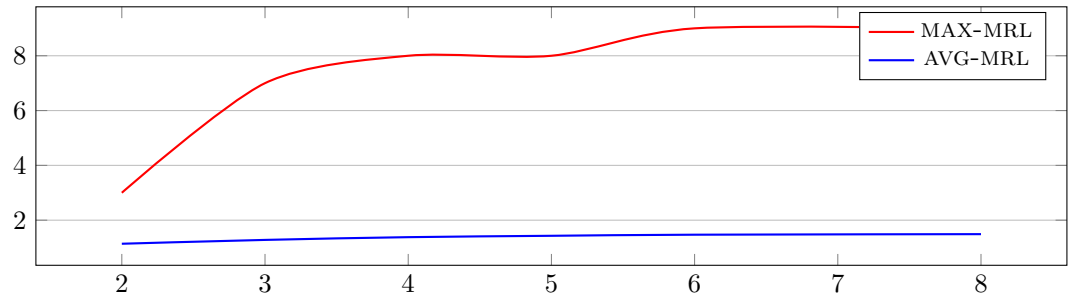

Figure 1. Average and maximal length of a monotonic run on a random text over an alphabet of size 2^{δ}. The ordinates show the length values while the ordinates show the values of δ.

Furthermore, we observe that the average length of the monotonic sub-sequences is much lower being always between 1.10 (for small alphabets) and 1.50 (for large alphabets). This implies that the average length of the monotonic run length scaled version of the text is on average between 66% and 90% of the length of the original sequence, a result not particularly exciting when compared with those obtained by previous sampling techniques such as CDS and OTS.

However, the fact that the length of each monotonic sub-sequence is up to 10 allows us to represent the monotonic run length scaled version of the text using only 4 bits for each element of the sequence, instead of the 8 bits needed in the OTS representation and the 32 bits needed in the CDS representation. Thus the average memory consumption required by the monotonic run length scaled version of the text is on average between 33% and 45% of the memory needed to store the original sequence.

2.1 Searching Using Monotonic Run Length Scaling

In this section we discuss the use of monotonic run length scaling as a sampling approach with application to exact and approximate string matching. In this preliminary work, for the case of approximate string matching, we take the Order Preserving Pattern Matching problem [18,6,7,2,10] as an case study, leaving section 5 with a broader discussion on the applicability of this method to other non-standard string matching problems.

Specifically, we present a naïve searching procedure designed to use the monotonic scaled version of the text as a partial index in order to speed up the search for the occurrences of a given pattern within the original text. Like any other solution based on text-sampling, the solution proposed in this section requires that the partial index is used in a preliminary filter phase and that each candidate occurrence identified in this first phase is then verified within the original text using a simple verification procedure.

The pseudo-code of the naïve searching procedure is depicted in Algorithm 1. The preprocessing phase of the algorithm consists in computing the monotonic run length scaled version $\widetilde{x}$ of the input pattern x. Let k be the length of the sequence $\widetilde{y}$ and let h be the length of the sequence $\widetilde{x}$. In addition let d be the length of the first monotonic run of x, i.e. $d = \widetilde{x}[0]$.

During the searching phase the algorithm naively searches for all occurrences of the sub-sequence $\widetilde{x}[1..h-2]$ along the scaled version of the text. We discard the first

Algorithm 1: Naïve algorithm based on Monotonic Run Length Scaling

```
Data: a pattern x of length m, a text y of length n and its scaled version ỹ of length k
Result: all positions 0 ≤ i < n such that x occurs in y starting at position i
x̃ ⟵ Monotonic-Run-Length-Scaling(x, m);
h ⟵ |x̃|;
d ⟵ x̃[0];
r ⟵ ỹ[0];
for s ⟵ 1 to k − h do
    j ⟵ 1;
    while (j < h − 1 and x̃[j] = ỹ[s + j]) do j ⟵ j + 1;
    if (j = h − 1) and then
        if Verify(y, n, x, m, r − d) then Output(r − d);
    end
    r ⟵ r + ỹ[s] − 1;
end
```

and the last element of $\widetilde{x}$ since they are allowed to match any any element in the text which is greater than or equal.

The main **for** loop of the algorithm iterates over a shift value s, which is initialized to 1 at the first iteration and is incremented by one when passing from one iteration to the next. During each iteration the current window of the text $\widetilde{y}[s..s+h-2]$ is attempted for a candidate occurrence. An additional variable r is maintained, representing the shift position in the original text y corresponding to the current window $\widetilde{y}[s..s+h-2]$. Thus at the beginning of each iteration of the main **for** loop the following invariant holds: $r = \widetilde{y}[0] + \sum_{i=1}^{s-1}(\widetilde{y}[i]) - 1)$.

At each iteration the window of the text $\widetilde{y}[s..s+h-2]$ is compared, character by character, against the sub-sequence $\widetilde{x}[1..h-2]$, proceeding form left to right. If a match is found then a candidate occurrence of the pattern is located at position $r-d$ of the original text and a verification phase is called in order to check if the sub-string $y[r-d...r-d+m-1]$ corresponds to a full occurrence of the pattern x. In all cases at the end of each attempt the value of r is increased by $\widetilde{y}[s]$. and the value of the shift s is increased by one position.

Example 5. As in Example 2, let $y = \langle 4,5,11,7,6,6,12,12,2,9,8,6,7,10,13\rangle$ be a numeric sequence of length 15 (the text) and let $x = \langle 12,2,9,8,6,7,10\rangle$ be a pattern of length 7. Then we have $\widetilde{y} = \langle 3,4,2,3,2,3,4\rangle$ and $\widetilde{x} = \langle 2,2,3,3\rangle$, with $d = 2$.

The algorithm naively searches the text $\widetilde{y}$ for any occurrence of the sub-sequence $\widetilde{x}[1..2] = \langle 2,3\rangle$, finding two candidate occurrences at position 2 and 4, respectively.

We obtain that:

- at the beginning of the third iteration of the **for** main loop, with $s = 2$, we have $r = 3+4-1 = 6$. Thus the verification procedure is run to compare x against the sub-sequence at position $r-d = 4$ in the original text, i.e. $\widetilde{y}[4..10] = \langle 6,6,12,12,2,9,8\rangle$. Unfortunately at position 6 the verification procedure would find neither an exact match nor an order preserving match.
- at the beginning of the fifth iteration of the **for** main loop, with $s = 4$, we have $r = 3+4+2+3-3 = 8$. Thus the verification procedure is run to compare x against the sub-sequence at position $r-d = 6$ in the original text, i.e. $\widetilde{y}[6..12] = \langle 12,12,2,9,8,6,7,10\rangle$. Thus at position 6 the verification phase would find both an exact match and an order preserving match.

Function 2: Monotonic-Run-Length-Sampling($\widetilde{x}, h, q$)

Data: The Monotonic Run Length Scaled version $\widetilde{x}$ of the string x, with length h
Result: The Monotonic Run Length Sampled version $\widetilde{x}^q$ of x
$\widetilde{x}^q \longleftarrow \langle\rangle$;
$r \longleftarrow 0$;
for $i \longleftarrow 0$ **to** $h-1$ **do**
  if $(\widetilde{x}[i] = q)$ **then** $\widetilde{x}^q \longleftarrow \widetilde{x}^q + \langle r \rangle$;
  $r \longleftarrow r + \widetilde{x}[i]$;
end
return $\widetilde{x}^q$

Regarding the complexity issues it is straightforward to observe that, if the verification phase can be run in $O(m)$ time, Algorithm 1 achieves a $\mathcal{O}(nm)$ worst case time complexity and requires only $\mathcal{O}(m)$ additional space for maintaining the scaled version of the pattern $\widetilde{x}$.

3 Monotonic Run Length Sampling

As observed above, the space consumption for representing the partial index obtained by monotonic run length scaling is not particularly satisfactory. This influences also the performance of the search algorithms in practical cases, as we will see in Section 4 which presents a preliminary experimental evaluation.

In this section we propose the application of an approach similar to that used by CDS sampling in order to obtain a partial index requiring a reduced amount of space, on the one hand, and is able to improve the performance of the search procedure, on the other hand.

For what we should present in this section it is useful to introduce some further notions. Given a monotonic run w of y, and assuming $w = y[i..i+k-1]$, we use the symbol $\mu(w)$ to indicate its starting position i in the text y.

The following definition introduces the *Monotonic Run Length Sampling* (MRLS) process which, given an input string y, constructs a partial index $\widetilde{y}^q$, which is the numeric sequence of all (and only) starting positions of any monotonic runs of y with length equal to q, for a given parameter $q > 1$.

Definition 6 (Monotonic Run Length Sampling). *Let y be a text of length n and let $\widetilde{y} = \langle |\rho_0|, |\rho_1|, .., |\rho_{k-1}| \rangle$ be the monotonic run length scaled version of y, with $|\widetilde{y}| = k$. In addition let ℓ be the maximal length of a monotonic run in y, i.e. $\ell = \max(|\rho_i| \ : \ 0 \leq i < k)$. If q is an integer value, with $2 \leq q \leq \ell$, we define the Monotonic Run Length Sampled version of y, with pivot length q, as the numeric sequence $\widetilde{y}^q$, defined as $\widetilde{y}^q = \langle |\rho_{i_0}|, |\rho_{i_1}|, .., |\rho_{i_{h-1}}| \rangle$, where $h \leq k$, $i_{j-1} < i_j$ for each $0 < j < h$, and $|\rho_{i_j}| = q$ for each $0 \leq j < h$.*

Example 7. Again, as in Example 2, let $y = \langle 4, 5, 11, 7, 6, 6, 12, 12, 2, 9, 8, 6, 7, 10, 13 \rangle$ be a numeric sequence of length 15. As already observed $\widetilde{y} = \langle 3, 4, 2, 3, 2, 3, 4 \rangle$. Thus we have: $\widetilde{y}^2 = \langle 5, 8 \rangle$, since $\mu(\langle 6, 12 \rangle) = 5$ and $\mu(\langle 2, 9 \rangle) = 8$; $\widetilde{y}^3 = \langle 0, 6, 9 \rangle$, since $\mu(\langle 4, 5, 11 \rangle) = 0$, $\mu(\langle 12, 12, 2 \rangle) = 6$ and $\mu(\langle 9, 8, 6 \rangle) = 9$; $\widetilde{y}^4 = \langle 2, 11 \rangle$, since $\mu(\langle 11, 7, 6, 6 \rangle) = 2$ and $\mu(\langle 6, 7, 10, 13 \rangle) = 11$.

Function 2 depicts the pseudo-code of the algorithm which computes the MRL sampled version of an input string x of length m. It gets as input the monotonic run

Algorithm 2: Naïve algorithm based on Monotonic Run Length Sampling

```
Data: a pattern x of length m, a text y of length n and its MRLS version ỹ^q of length k
Result: all positions 0 ≤ i < n such that x occurs in y starting at position i
x̃ ⟵ Monotonic-Run-Length-Scaling(x, m);
x̃^q ⟵ Monotonic-Run-Length-Sampling(x, m);
d ⟵ x̃*[0];
for s ⟵ 0 to k − h do
    r ⟵ ỹ*[s];
    if (r − d ≥ 0 and r − d + m − 1 < n) then
        if Verify(y, n, x, m, r − d) then Output(r − d);
    end
end
```

length scaled version $\widetilde{x}$ of the string, its length h and the pivot length q. Then it constructs the sequence $\widetilde{x}^q$ incrementally by scanning the input sequence $\widetilde{x}$ element by element, proceeding from left to right. It is straightforward to prove that, also in this case, the worst case time complexity of the procedure is $\mathcal{O}(m)$.

3.1 Searching Using Monotonic Run Length Sampling

In this section we shortly describe a simple naïve procedure to search for all occurrences (in their exact or approximate version) of a pattern x of length m inside a text y of length n. The pseudo-code of such procedure is depicted in Algorithm 2.

The algorithm takes as input both the text y and its MRLS version $\widetilde{y}^q$ of length k. During the preprocessing phase the algorithm first computes the scaled version $\widetilde{x}$ of the input pattern x and, subsequently, computes its monotonic run length sampled version $\widetilde{x}^q$. Let k be the length of the sequence $\widetilde{y}$ and let h be the length of the sequence $\widetilde{x}^q$. In addition let d be the starting position of the first monotonic run length of x of length q, i.e. $d = \widetilde{x}^q[0]$.

The searching phase of the algorithm consists in a main **for** loop which iterates over the sequence $\widetilde{y}^q$, proceeding form left to right. For each element $\widetilde{y}^q[s]$, for $0 \leq s < h$, the algorithm calls the verification procedure to check an occurrence beginning at position $\widetilde{y}^q[s] - d$ in the original text. Roughly speaking, the algorithm aligns the first monotonic of length q in the pattern with all monotonic runs of length q in the text.

Example 8. As in Example 2, let $y = \langle 4, 5, 11, 7, 6, 6, 12, 12, 2, 9, 8, 6, 7, 10, 13\rangle$ be a numeric sequence of length 15 (the text) and let $x = \langle 12, 2, 9, 8, 6, 7, 10\rangle$ be a pattern of length 7. Then we have $\widetilde{y} = \langle 3, 4, 2, 3, 2, 3, 4\rangle$ and $\widetilde{x} = \langle 2, 2, 3, 3\rangle$. Assuming $q = 3$ we have also $\widetilde{y}^q = \langle 0, 6, 9\rangle$, $\widetilde{x}^q = \langle 2, 4\rangle$ and $d = 2$.

The algorithm considers any position $r \in \widetilde{y}^q$ as a candidate occurrence of the pattern and naively checks the whole pattern against the sub-sequence $y[r - d...r - d + m - 1]$ of the original text. Thus we have

- at the first iteration of the main **for** loop we have $s = 0$ and the algorithm would run a verification for the window starting at $\widetilde{y}^q[0] - d = 0 - 2 = -2$. However, since $-2 < 0$, such window is skipped.
- at the second iteration of the main **for** loop we have $s = 1$ and the algorithm would run a verification for the window starting at $\widetilde{y}^q[1] - d = 6 - 2 = 4$. Thus the pattern x is compared with the sub-sequence $y[4...10] = \langle 6, 6, 12, 12, 2, 9, 8\rangle$. However in

this case neither an exact occurrence nor an order preserving occurrence would be found.
- finally, at the third iteration of the main **for** loop we have $s = 2$ and the algorithm would run a verification for the window starting at $\widetilde{y}^q[9] - d = 9 - 2 = 7$. Thus the pattern x is compared with the sub-sequence $y[7...13] = \langle 12, 2, 9, 8, 6, 7, 10\rangle$. In this case the verification procedure would find both an exact occurrence and an order preserving match.

Regarding the complexity issues it is straightforward to observe that, if the verification phase can be run in $O(m)$ time, also Algorithm 1 achieves a $\mathcal{O}(nm)$ worst case time complexity and requires only $\mathcal{O}(m)$ additional space for maintaining the sampled version of the pattern $\widetilde{x}^q$.

4 Experimental Evaluation

In this section, we present experimental results in order to evaluate the performances of the sampling approaches presented in this paper for both exact string matching and order preserving pattern matching.

The algorithms have been implemented using the C programming language, and have been tested using the SMART tool [12][1] 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.[2] During the compilation we use the -O3 optimization option. .

For both exact and approximate string matching comparisons have been performed in terms of searching times. For our tests, we used six RAND-δ sequences of short integer values (each element of the sequence is an integer in the range [0...256]) varying around a fixed mean equal to 100 with a variability of δ, with $\delta \in \{4, 8, 16, 32, 64, 128, 250\}$. All sequences have a size of 3MB and are provided by the SMART research tool, available online for download. For each sequence in the set, we randomly selected 100 patterns, extracted from the text, and computed the average running time over the 100 runs.

4.1 Space Consumption

The first evaluation we discuss in this section relates to the space used to maintain the partial index. This evaluation is independent of the application for which the index is used and has the same value for both exact and approximate string matching.

Figure 2 shows the space consumption of the new proposed sampling approaches compared against the CDS and OTS methods. Data are reported as percentage values with respect to the size of the original text on which the index is built. Values are computed on six random texts with a uniform distribution and built on an alphabet of size 2^δ (abscissa) with $2 \leq \delta \leq 8$.

From the data shown in the Figure 2 it is possible to observe how the best results in terms of space are obtained by the MRLS and CDS. However, while the latter tends to have good results only for large alphabets, the MRLS approach proves to be

[1] In the case of OPPM experimental evaluations the tool has been properly tuned for testing string matching algorithms based on the OPPM approach

[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`.

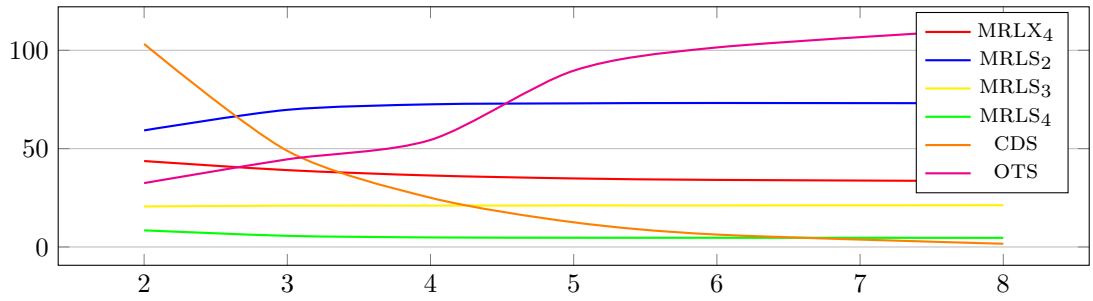

Figure 2. Space Consumption of sampling approaches to text searching. Data are reported as percentage values with respect to the size of the original text on which the index is built. Values are computed on six random texts with a uniform distribution and built on an alphabet of size 2^δ (abscissa) with $2 \leq \delta \leq 8$.

more flexible, obtaining good results also for small alphabets. However, it should be noted that these results were obtained on random texts with a uniform distribution of characters, a condition not favorable to the best performances for CDS and OTS.

4.2 Running Times for the OPPM problem

For the OPPM problem we took the standard NRq algorithm [6] as a reference point for our evaluation, since it is one of the most effective solution known in literature. Specifically we evaluated the following text-sampling solutions:

- (MRLX$_4$) The Monotonic Run Length Scaling approach (Section 2) using a compact representation of the elements (4 bits for each run length) and implemented using the Naïve algorithm.
- (MRLX$_8$) The Monotonic Run Length Scaling approach (Section 2) using a relaxed representation of the elements (an 8-bits char for each run length) and implemented using the Naïve algorithm.
- (MRLS$_q$) The Monotonic Run Length Sampling approach approach (Section 3) using a 32-bits integer value for each text position and implemented by sampling runs of length q, with $2 \leq q \leq 4$.

Table 1 shows the experimental evaluation for the OPPM problem on the Rand-δ short integer sequences where running times are expressed in milliseconds.

To better highlight the improvements obtained by the new proposed solutions, in Table 1 we show the running times only for the reference NRq algorithm, while we show the speed-up obtained against the latter for all the other tested algorithms. In this context a value greater than 1 indicates a speed-up of the running times proportional to the reported value, while a value less than 1 indicates a slowdown in performance. Best and second best results have been enhanced for a better visualization.

From our experimental results it turns out that the best solution in almost all the cases analyzed is the MRLS$_q$ algorithm which is almost always twice as fast as the NRq algorithm and reaches impressive speed-ups for very long patterns, up to 40 times faster than the reference algorithm. The MRLS$_q$ algorithm is second only to the MRLX$_8$ algorithm for short patterns ($m = 8$).

More specifically the MRLX$_8$ algorithm allows speed-ups compared to NRq, however these oscillate between one and a half times and twice as fast as the reference algorithm. In general, its performance is not that impressive. The MRLX$_4$ algorithm

δ	m	NRq	MRLX$_4$	MRLX$_8$	MRLS$_2$	MRLS$_3$	MRLS$_4$
4	8	5.40	0.59	**1.19**	0.91	0.72	0.70
	16	5.21	0.74	1.34	**1.50**	0.76	0.73
	32	5.25	0.78	1.40	**3.65**	1.52	0.71
	64	5.07	0.71	1.25	3.57	**3.67**	1.16
	128	5.21	0.81	1.37	3.59	**9.65**	2.52
	256	5.61	0.77	1.40	3.98	10.58	**14.38**
	512	5.71	0.81	1.46	4.05	10.77	**22.84**
	1024	5.20	0.78	1.42	3.59	9.45	**18.57**
8	8	4.70	0.67	**1.21**	1.06	0.78	0.71
	16	4.89	0.92	1.49	**2.24**	0.82	0.74
	32	4.90	0.93	1.50	**3.29**	1.29	0.85
	64	4.96	1.01	1.57	3.67	**5.51**	1.10
	128	5.03	1.03	1.58	3.70	**11.18**	1.76
	256	5.14	1.01	1.52	3.75	**11.42**	3.67
	512	4.73	0.92	1.44	3.50	10.28	**15.26**
	1024	5.20	0.95	1.57	3.91	11.56	**30.59**
16	8	4.90	0.84	**1.52**	1.19	0.78	0.78
	16	5.24	1.05	1.88	**1.93**	1.15	0.82
	32	4.67	1.00	1.59	**3.38**	1.64	0.80
	64	4.87	0.99	1.61	3.87	**4.35**	1.07
	128	4.98	1.02	1.63	4.02	**9.76**	2.06
	256	4.94	1.01	1.60	3.98	**12.05**	4.57
	512	5.20	1.16	1.95	4.19	12.68	**37.14**
	1024	4.96	1.05	1.68	4.03	12.10	**35.43**
32	8	5.00	0.96	**1.64**	1.10	0.81	0.82
	16	4.66	0.99	1.77	**2.59**	0.92	0.77
	32	4.90	1.14	1.99	**3.38**	1.72	0.87
	64	4.83	1.09	1.90	**3.83**	3.43	1.04
	128	4.88	1.08	1.85	4.17	**12.84**	1.51
	256	4.76	1.05	1.93	4.10	**12.53**	3.81
	512	5.07	1.15	1.96	4.30	13.34	**39.00**
	1024	5.23	1.13	1.99	4.59	13.41	**40.23**
64	8	5.12	1.02	**1.78**	1.32	0.89	0.87
	16	4.93	1.02	1.83	**2.77**	0.89	0.81
	32	4.91	1.10	2.00	**3.48**	1.66	0.83
	64	4.90	1.12	2.04	3.60	**6.53**	0.98
	128	4.84	1.08	1.94	3.87	**11.80**	1.58
	256	4.97	1.13	2.05	4.28	**13.08**	3.88
	512	4.66	1.02	1.83	4.09	12.26	**38.83**
	1024	4.82	1.07	1.84	4.38	12.68	**40.17**
256	8	4.99	0.99	**1.81**	1.23	0.89	0.85
	16	4.83	1.05	1.92	**2.60**	0.93	0.77
	32	5.01	1.12	2.09	**3.48**	1.69	0.83
	64	4.76	1.08	1.91	3.33	**6.52**	0.86
	128	4.96	1.09	1.95	3.59	**8.70**	1.58
	256	4.72	1.07	1.93	3.75	**11.51**	3.87
	512	4.79	1.06	1.94	3.89	**11.97**	10.41
	1024	4.89	1.12	2.08	4.37	12.54	**37.62**

Table 1. Experimental results for the OPPM problem on six Rand-δ short integer sequence, for $4 \leq \delta \leq 256$. Running times of the NRq are expressed in milliseconds. Results for all other algorithm are expressed in terms of speed-up obtained against the reference NRq algorithm. Best results and second best results have been enhanced.

performs worse and almost never brings improvements over NRq, probably due to the trade-off introduced by the compact representation of the partial index.

4.3 Running Times for the ESM problem

For the ESM problem, in accordance with what has been done in previous publications on the subject, we took the standard the Horspool (HOR) algorithm [17] as a reference point for our evaluation. We evaluated the following text-sampling solutions:

- (OTS) The Occurrence Text Sampling approach [8] introduced by Claude *et al.* and implemented by removing the first 8 characters of the alphabet, with the exception of the case $\delta = 4$, for which we removed the first 3 characters, and the case $\delta = 8$, for which we removed the first 7 characters.
- (CDS) The Character Distance Sampling approach [14] introduced by Faro *et al.*, implemented by selecting the 8th character of the alphabet, with the exception of the case $\delta = 4$, for which we selected the 4th character.
- (MRLS$_q$) The Monotonic Run Length Position Sampling approach (Section 3) using a 32-bits integer value for each text position and implemented using the sampling of runs of length q, with $2 \leq q \leq 4$.

Table 2 shows the experimental evaluation for the ESM problem on the Rand-δ short integer sequences where running times are expressed in milliseconds.

Also in this case to better highlight the improvements obtained by the new proposed solutions, in Table 2 we show the running times only for the reference HOR algorithm, while we show the speed-up obtained against the latter for all the other tested algorithms.

Our experimental results show that in the case of medium and large-sized alphabets, the MRLS$_q$ algorithm does not have the same performance as the CDS approach. However, it is very powerful in the case of small alphabets, a case in which the previous solutions suffered particularly and showed not exciting results.

It is also interesting to observe how the MRLS$_q$ algorithm proves to be competitive in the general case, always obtaining the second best results. The speed-up obtained by comparing it with the reference HOR algorithm reaches a factor of 2, in the case of small alphabets, and a factor of 6 for large alphabets.

5 Conclusions and Future Works

This article presents the first results relating to a work in progress. Specifically, we presented a new technique for text sampling called Monotonic Run Length Scaling (MRLX), an approach based on the length of the monotonic runs present within the text, flexible enough to be used both for exact string matching and for approximate string matching. A further improvement was obtained by sampling through the sampling of the lengths of the monotonic runs present in the text, an approach that we have called Monotonic Run Length Sampling (MRLS). In this work we also presented some first experimental tests for the evaluation of the two sampling approaches and implemented using naive search algorithms, focusing on two case studies: exact string matching and order preserving pattern matching.

The first experimental results obtained showed how the approaches are particularly versatile, obtaining considerable speed-ups on execution times, reaching gain

δ	m	HOR	CDS	OTS	$MRLS_2$	$MRLS_3$	$MRLS_4$
4	8	2.43	1.21	1.31	**1.48**	1.23	1.19
	16	2.10	1.19	1.31	**1.98**	1.27	1.25
	32	1.99	1.21	1.33	**2.65**	2.40	1.24
	64	2.16	1.23	1.33	2.96	**4.50**	2.00
	128	2.10	1.20	1.29	2.84	**6.77**	3.96
	256	1.99	1.23	1.33	2.62	6.42	**10.47**
	512	2.01	1.19	1.32	2.75	6.48	**12.56**
	1024	1.97	1.22	1.31	2.70	5.97	**12.31**
8	8	1.40	**2.19**	1.44	1.65	1.33	1.19
	16	1.11	**2.92**	1.52	1.88	1.50	1.39
	32	0.99	**3.67**	1.62	1.83	2.02	1.60
	64	1.01	**6.73**	1.58	1.84	4.04	1.87
	128	1.02	**9.27**	1.62	1.89	4.64	2.83
	256	1.04	**11.56**	1.58	1.96	4.73	4.52
	512	1.01	**14.43**	1.55	1.87	4.59	9.18
	1024	1.01	**16.83**	1.63	1.87	5.05	11.22
16	8	1.04	**1.65**	0.94	1.35	1.41	1.39
	16	0.79	**2.26**	1.27	1.76	1.98	1.52
	32	0.68	**4.00**	1.70	1.58	2.52	1.79
	64	0.62	**6.20**	1.72	1.48	3.26	2.07
	128	0.61	**10.17**	1.85	1.45	3.59	3.21
	256	0.61	**12.20**	1.69	1.49	3.81	4.69
	512	0.63	**15.75**	1.66	1.50	3.94	7.88
	1024	0.61	**20.33**	1.74	1.45	3.81	7.62
32	8	0.92	**1.53**	1.39	1.39	1.46	1.48
	16	0.66	**1.94**	1.83	1.74	1.94	1.74
	32	0.55	**3.06**	2.20	1.49	2.75	2.04
	64	0.53	**6.62**	2.52	1.43	3.31	2.41
	128	0.50	**8.33**	2.63	1.32	3.57	3.12
	256	0.50	**12.50**	2.78	1.35	3.57	4.55
	512	0.47	**15.67**	2.94	1.27	3.62	6.71
	1024	0.50	**25.00**	2.78	1.39	3.33	7.14
64	8	0.85	**1.60**	1.52	1.47	1.52	1.49
	16	0.60	**2.07**	2.00	1.76	1.88	1.71
	32	0.50	2.78	2.50	1.28	**2.94**	2.27
	64	0.46	**5.11**	3.07	1.39	3.29	2.71
	128	0.44	**11.00**	3.38	1.29	3.38	4.00
	256	0.43	**14.33**	3.31	1.30	3.31	4.78
	512	0.43	**21.50**	3.91	1.30	3.31	6.14
	1024	0.42	**21.00**	3.50	1.24	3.23	6.00
256	8	0.77	**1.54**	1.54	1.71	1.57	1.57
	16	0.58	**2.23**	2.07	1.81	2.07	1.93
	32	0.46	**3.29**	2.56	1.39	2.88	2.19
	64	0.45	**5.00**	3.45	1.36	3.46	3.21
	128	0.44	**11.00**	3.37	1.33	3.38	4.40
	256	0.45	**15.00**	4.90	1.32	3.21	5.00
	512	0.47	**23.50**	5.85	1.42	3.62	5.87
	1024	0.47	**23.50**	5.86	1.38	3.36	5.87

Table 2. Experimental results for the ESM problem on six Rand-δ short integer sequence, for $4 \leq \delta \leq 256$. Running times of the HOR are expressed in milliseconds. Results for all other algorithm are expressed in terms of speed-up obtained against the reference HOR algorithm. Best results and second best results have been enhanced.

factors of 40, in particularly favorable conditions. The new techniques presented therefore serve as good starting points for improvements in future investigations. The first aspect of the research that can be carried out for future improvements is the choice of the underlying search algorithms used for the implementation of the partial index filtering procedure. The MRLX approach requires algorithms that scan the text, reading every single character, in order not to lose the information relating to the alignment with the position of the occurrences in the original text. This can be done using more efficient string matching algorithms such as the KMP [19] or the Shift-And [1] algorithms. The MRLS approach, using text positions as the primary information of the numerical sequence, can afford the use of more efficient string matching algorithms that skip portions of the text during the search. In this case, we expect more significant improvements in execution times.

However, one of the most interesting aspects to be analyzed in a future work is the applicability of the new sampling approach to other approximate string matching problems. We can now say that the MRLX technique is well suited to solve other problems such as Cartesian-tree pattern matching [22] or shape preserving pattern matching [6], which have a strong relationship with the OPPM problem. Furthermore we argue that an approximate search within the partial index can also allow to obtain solutions for problems in which the occurrence of the pattern is found within the text in the form of some kind of permutation of its characters. And many non-standard string matching problems belong to this category, such as swap matching [15], string matching with inversions and/or moves [5] as well as the jumbled matching itself.

References

1. R. Baeza-Yates and G. H. Gonnet: *A new approach to text searching.* Commun. ACM, 35(10) Oct. 1992, p. 74–82.
2. D. Belazzougui, A. Pierrot, M. Raffinot, and S. Vialette: *Single and multiple consecutive permutation motif search*, in Algorithms and Computation - 24th International Symposium, ISAAC 2013, Hong Kong, China, December 16-18, 2013, Proceedings, vol. 8283 of Lecture Notes in Computer Science, Springer, 2013, pp. 66–77.
3. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Commun. ACM, 20(10) 1977, pp. 762–772.
4. D. Cantone, S. Faro, and E. Giaquinta: *Adapting boyer-moore-like algorithms for searching huffman encoded texts.* Int. J. Found. Comput. Sci., 23(2) 2012, pp. 343–356.
5. D. Cantone, S. Faro, and E. Giaquinta: *Text searching allowing for inversions and translocations of factors.* Discret. Appl. Math., 163 2014, pp. 247–257.
6. D. Cantone, S. Faro, and M. O. Külekci: *Shape-preserving pattern matching*, in Proceedings of the 21st Italian Conference on Theoretical Computer Science 2020, vol. 2756 of CEUR Workshop Proceedings, CEUR-WS.org, 2020, pp. 137–148.
7. S. Cho, J. C. Na, K. Park, and J. S. Sim: *Fast order-preserving pattern matching*, in Combinatorial Optimization and Applications - 7th International Conference, COCOA 2013, Proceedings, vol. 8287 of Lecture Notes in Computer Science, Springer, 2013, pp. 295–305.
8. F. Claude, G. Navarro, H. Peltola, L. Salmela, and J. Tarhio: *String matching with alphabet sampling.* J. Discrete Algorithms, 11 2012, pp. 37–50.
9. M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter: *Speeding up two string-matching algorithms.* Algorithmica, 12(4/5) 1994, pp. 247–267.
10. S. Faro and M. O. Külekci: *Efficient algorithms for the order preserving pattern matching problem*, in Algorithmic Aspects in Information and Management - 11th International Conference, AAIM 2016, Proceedings, vol. 9778 of Lecture Notes in Computer Science, Springer, 2016, pp. 185–196.

11. S. Faro and T. Lecroq: *The exact online string matching problem: A review of the most recent results.* ACM Comput. Surv., 45(2) 2013, pp. 13:1–13:42.
12. S. Faro, T. Lecroq, S. Borzi, S. D. Mauro, and A. Maggio: *The string matching algorithms research tool*, in Proceedings of the Prague Stringology Conference 2016, Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2016, pp. 99–111.
13. S. Faro and F. P. Marino: *Reducing time and space in indexed string matching by characters distance text sampling*, in Prague Stringology Conference 2020, Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2020, pp. 148–159.
14. S. Faro, F. P. Marino, and A. Pavone: *Efficient online string matching based on characters distance text sampling.* Algorithmica, 82(11) 2020, pp. 3390–3412.
15. S. Faro and A. Pavone: *An efficient skip-search approach to swap matching.* Comput. J., 61(9) 2018, pp. 1351–1360.
16. P. Ferragina and G. Manzini: *Indexing compressed text.* J. ACM, 52(4) 2005, pp. 552–581.
17. R. N. Horspool: *Practical fast searching in strings.* Softw. Pract. Exp., 10(6) 1980, pp. 501–506.
18. J. Kim, P. Eades, R. Fleischer, S. Hong, C. S. Iliopoulos, K. Park, S. J. Puglisi, and T. Tokuyama: *Order-preserving matching.* Theor. Comput. Sci., 525 2014, pp. 68–79.
19. D. E. Knuth, J. H. M. Jr., and V. R. Pratt: *Fast pattern matching in strings.* SIAM J. Comput., 6(2) 1977, pp. 323–350.
20. U. Manber and E. W. Myers: *Suffix arrays: A new method for on-line string searches.* SIAM J. Comput., 22(5) 1993, pp. 935–948.
21. G. Navarro and J. Tarhio: *Lzgrep: a boyer-moore string matching tool for ziv-lempel compressed text.* Softw. Pract. Exp., 35(12) 2005, pp. 1107–1130.
22. S. Song, G. Gu, C. Ryu, S. Faro, T. Lecroq, and K. Park: *Fast algorithms for single and multiple pattern cartesian tree matching.* Theor. Comput. Sci., 849 2021, pp. 47–63.
23. U. Vishkin: *Deterministic sampling - A new technique for fast pattern matching.* SIAM J. Comput., 20(1) 1991, pp. 22–40.
24. A. C. Yao: *The complexity of pattern matching for a random string.* SIAM J. Comput., 8(3) 1979, pp. 368–387.