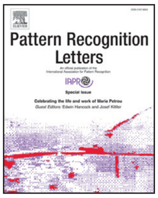
Pattern Recognition Letters 55 (2015) 15–21

Contents lists available at ScienceDirect

Pattern Recognition Letters

journal homepage: www.elsevier.com/locate/patrec



Order preserving pattern matching revisited[☆]

Md. Mahbubul Hasan[a,1], A.S.M. Shohidull Islam[a,b], Mohammad Saifur Rahman[a], M. Sohel Rahman[a,*]

[a] AℓEDA Group, Department of CSE, BUET, Dhaka 1000, Bangladesh
[b] Department of Computational Engineering and Science, McMaster University, Hamilton, Ontario, Canada

ARTICLE INFO

Article history:
Received 18 March 2014
Available online 11 December 2014

Keywords:
Algorithms
Pattern matching
Document processing
Sequence analysis

ABSTRACT

In this paper, we study the order preserving pattern matching (OPPM) problem, which is a very recent variant of the classic pattern matching problem. We revisit this variant, present a new interesting pattern matching algorithm and for the first time consider string regularities from this new perspective.

1. Introduction

Given a string (or text) T and pattern P under an alphabet Σ, the classic string/pattern matching problem asks whether P occurs in T and if yes, then it further reports the occurrences of P in T. This problem has extensive applications in different branches of science and engineering. Due to different types of requirements in different application scenarios, a plethora of variants of the classic string matching problem have been introduced and studied in the literature. The focus of this paper is a very recent variant which is called the order preserving pattern matching (OPPM). In OPPM, like the classic variant, we have a text T and a pattern P as input. However, the underlying alphabet here is an integer alphabet. And, instead of looking for a substring of the text which is identical to the given pattern, we are interested in locating a fragment which is order-isomorphic with the pattern. Two sequences over an integer alphabet are order-isomorphic if the relative order between any two elements at the same positions in both the sequences is the same.

To the best of our knowledge, OPPM was first studied independently by Kim et al. [6] and Kubica et al. [8].[2] Since then, within quite a short period of time, a number of works on OPPM in different directions have been reported in the literature. For example, order preserving suffix trees and index data structures have been devised and different applications thereof have been discussed by Crochemore et al. [2,3]. On the other hand, Cho et al. [1] have presented practically fast algorithms for OPPM. Gawrychowski and Uznanski [4] have considered the approximate version of the problem where k mismatches are allowed.

In this paper, we revisit the order preserving pattern matching problem. Our main contribution in this paper is the study of string regularities from an order preserving point of view. To the best of our knowledge, this is the first attempt to capture the concept regularities form this perspective. In what follows we will conveniently refer to this as *order preserving regularities*. String regularities have been the focus of attention of stringology researchers since long. The following has been articulated by Smyth [10] in a very recent survey on string regularities:

> . . . In the intervening century, certainly thousands of research papers have been written by mathematicians and (over the last half century) also computer scientists that relate in some way to periodicity, or its variants, in strings. A word that has recently been brought into service to describe these variants is “regularities” . . .

Apart from periodicity, the most notable and studied regularities of a string are borders and covers. Hence, in this paper we consider periods, borders and covers of strings from the order preserving point of view (Section 4). We also discuss yet another order preserving pattern matching algorithm (Section 4). Our algorithms are based on the so called Z-algorithm discussed by Gusfield in Chapter 2 of his famous book [5]. In particular, we propose modifications to the Z-algorithm or Z-function of Gusfield to make it useful in the order preserving framework (Section 3). The modified Z-algorithm for a string presented in this paper could be of independent interest.

[☆] This paper has been recommended for acceptance by A. Koleshnikov.

[*] Corresponding author at: Commonwealth Academic Fellow funded by the UK Government. Currently on a sabbatical leave from BUET. Tel.: +8801552389480; fax: +448712475276.

E-mail address: msrahman@cse.buet.ac.bd (M. Sohel Rahman).

[1] Currently working at Google Zurich.

[2] The problem was also presented at the Theo Murphy International Scientific Meeting of the Royal Society on Storage and Indexing of Massive Data.

http://dx.doi.org/10.1016/j.patrec.2014.11.013
0167-8655/© 2014 Published by Elsevier B.V.

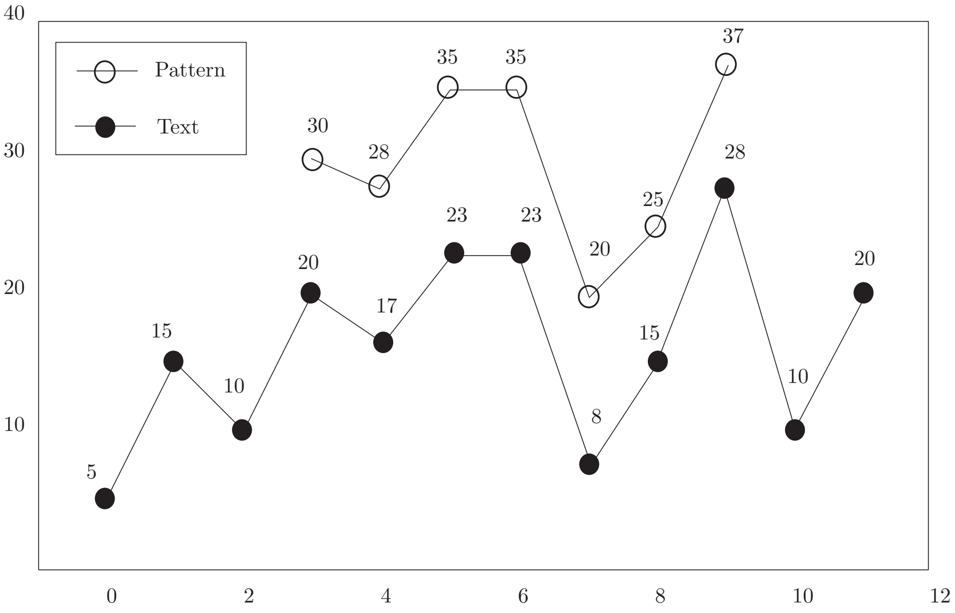


Fig. 1. An example instance of OPPM.

Table 1
An example of Z-function.

S	=	A	C	A	G	G	T	A	C	A	G	T	T	C	C	C	T	C	G	A	C	A	C	C	T	A	C	T	A	C	C	T	A	A	G
$Z(S)$	=	34	0	1	0	0	0	4	0	1	0	0	0	0	0	0	0	0	0	3	0	2	0	0	0	2	0	0	2	0	0	0	1	1	0

Besides the inherent combinatorial and algorithmic beauty of the problem itself, the motivation also comes from a number of interesting practical applications. For example, it can be applied to a time series analysis like share prices on stock markets to recognize specific patterns and to musical melody matching of two musical scores [1,6]. An example is given in Fig. 1. The problem also has some interesting relation to the combinatorial study of patterns in permutations, in particular to the study of pattern avoidance [8].

2. Preliminaries

We use Σ to denote the set of numbers such that the comparison of two numbers can be done in constant time and Σ^* denotes the set of strings over the alphabet Σ. The length of a string $S = (S[1], S[2], \ldots, S[n])$ is denoted by $|S| = n$. For $n = 0$, we say that $S = \varepsilon$, the empty string. If $S = UVW$, then U is said to be a prefix, V a substring (also called a factor) and W a suffix of S. If $VW \neq \varepsilon$ $(UV \neq \varepsilon)$ then U (W) is a proper prefix (proper suffix) of S. Similarly, if $UW \neq \varepsilon$, then V is a proper substring. A substring $(S[i], S[i+1], \ldots, S[j])$ of S is denoted by $S[i, j]$, the ith prefix $S[1, i]$ by prefix$_i(S)$ and the ith suffix $S[i, |S|]$ by suffix$_i(S)$.

The rank of a number c in a string S is defined as follows:

$$\mathrm{rank_S(c) = 1 + |\{i \mid S[i] < c, 1 \leq i \leq |S|\}|}.$$

To be consistent with the previous works [6], we assume that all the numbers in a string are distinct. The order preserving representation (OPR) $\sigma(S)$ of a string S can be defined as follows:

$$\sigma(S) = \mathrm{rank_S(S[1]), rank_S(S[2]), \ldots, rank_S(S[|S|])}.$$

If two strings S_1 and S_2 have identical OPR, i.e., $\sigma(S_1) = \sigma(S_2)$, then we say that the two strings are OPR-equal.

Since we are interested in string regularities, here we discuss some related notions and definitions. Part of the following description is conveniently adopted from a recent survey of Smyth [10]. If $S = S[1 \ldots n]$ has a proper (though possibly empty) prefix U that is also a suffix of S, then U is said to be a border of S. For example, for string $S = (1, 2, 1, 2, 1)$, one border would be $(1, 2, 1)$. The entire string can be considered as a special *border*. If for some $p \in 1 \ldots n$, $S[i] = S[p+i]$ for every $i \in 1 \ldots n-p$, then S is said to have period p. Thus S always has the empty border ε and the trivial period n. It is well-known, and easy to prove, that S has period p if, and only if, it has a border of length $n - p$. The border array β_S of a string S is an array of length n such that $\beta_S[i]$ equals the length of the longest border of $S[1 \ldots i]$ for every $i \in 1 \ldots n$. Since $\beta_S[i] = b > 0$ implies that $\beta_S[b]$ is the next largest border of $S[1 \ldots i]$, it follows that β_S specifies all the borders, hence all the periods, of every prefix of S. A simple $\Theta(n)$-time algorithm can compute the border array of a string having length n [9].

A string S has quasiperiod $q < n$ if and only if there exists a string $U = U[1 \ldots q]$, called a cover of S, such that every position of S lies within an occurrence of U. Thus a cover must also be a border of S. For example, $U = (1, 2, 1)$ is a cover of $S = (1, 2, 1, 2, 1, 1, 2, 1)$. A cover $U \neq S$ is called a *proper cover*.

Since we will be heavily using the Z-function of Gusfield [5], here we briefly review the concept. The Z-function, $Z_i(S)$ is the length of the longest substring of S that starts at position i and matches a prefix of S. We give an example of the Z-function for a string S on the DNA alphabet (i.e., {A, C, G, T}) in Table 1.

3. Modified *Z*-function

3.1. *Definition*

In this section, we propose a modification of the Z-function to make it useful from the order preserving point of view. We start with the following formal definition. For the sake of notational ease, we do not introduce an extended notation to denote Z-function from order preserving point of view; so, in the rest of this paper, unless otherwise specified, we will continue to use the term Z-function to consider it from the order preserving point of view.

Definition 1. Given a string S, the Z-function $Z_i(S)$, $1 \leq i \leq |S|$ is the length of the longest prefix P of suffix$_i(S)$ such that $\sigma(P) = \sigma(S[1, |P|])$.

Table 2
Example of (modified) Z-function.

S	=	(11,	18,	24,	20,	25,	29)
$Z(S)$	=	6	2	1	3	2	1

Algorithm 1 High level overview of Z-function construction.

```
 1: procedure ZFunction($S$)
 2:     $Z(1) := |S|$
 3:     $L := R := 1$
 4:     for $i := 2$ to $|S|$ do
 5:         $Z(i) := 0$
 6:         if $i \le R$ then
 7:             $Z(i) := min(R - i + 1, Z(i - L + 1))$
 8:         end if
 9:         while $i + Z(i) \le |S|$ and $\sigma(S[i, i + Z(i)]) = \sigma(S[1, Z(i) + 1])$ do
10:             $Z(i) := Z(i) + 1$
11:         end while
12:         if $i + Z(i) - 1 > R$ then
13:             $L := i$
14:             $R := i + Z(i) - 1$
15:         end if
16:     end for
17:     return $Z$
18: end procedure
```

In other words, $Z_i(S)$ is the length of the longest substring of S that starts at position i and its OPR is same as the OPR of some prefix of S. An example of a (modified) Z-function is given in Table 2.

3.2. Construction

In this section, we focus on constructing the modified Z-function. In Algorithm 1, we formally present the modified Z-function construction procedure.

3.3. Correctness

In this section we discuss the correctness of the modifed Z-function construction algorithm. We start with the following lemma which will be useful shortly.

Lemma 1. *Let S and T be two strings such that $\sigma(S) \neq \sigma(T)$. Now assume that s and t are any numbers. Then we must also have $\sigma(Ss) \neq \sigma(Tt)$.*

Proof. We prove it by contradiction. Assume for the sake of contradiction that $\sigma(Ss) = \sigma(Tt)$. So the rank of s in S is same as the rank of t in T. If we remove s and t from the corresponding strings, the ranks of the numbers greater than s and t will reduce by 1; ranks of the numbers less than those will remain the same. Hence we must have $\sigma(S) = \sigma(T)$, a contradiction. □

It is very easy to extend Lemma 1 to get the following lemma.

Lemma 2. *Let S and T are two strings such that $|S| = |T|$ and $\sigma(S) \neq \sigma(T)$. Now let U and V are any two strings. Then $\sigma(SU) \neq \sigma(TV)$.*

Algorithm 2 Calculation of OPR of a string S.

```
procedure ComputeOPR($S$)
    $\mathcal{T}$ := Empty Tree
    for $i := 1$ to $|S|$ do
        $\mathcal{T}$.Insert($S[i]$, $i$)
    end for
    for $i := 1$ to $|S|$ do
        OPR($i$) := $\mathcal{T}$.Rank($S[i]$)
    end for
    return OPR
end procedure
```

Now we discuss the correctness of our algorithm. In this algorithm we maintain two pointers L and R such that $\sigma(S[L, R]) = \sigma(S[1, R - L + 1])$ and R is as large as possible for a specific L. The algorithm always maintain the invariant that $L \le i$. This is ensured because the value of L is conditionally updated to i only in line 13. Now, as we proceed from $i = 2$ to $|S|$, for each i we check whether $i \le R$ and if so, we can safely say that $Z(i)$ is at least the minimum of $R - i + 1$ and $Z(i - L + 1)$; otherwise $Z(i)$ is set to 0. The reason for such bound of $Z(i)$ is the fact that we always have $\sigma(S[L, R]) = \sigma(S[1, R - L + 1])$.

In fact, we have a stronger invariant: OPR of any substring of $S[L, R]$ is same as the OPR for the corresponding substring of $S[1, R - L + 1]$. So, OPR of $S[i, R]$ will be same as the OPR of $S[i - L + 1, R - L + 1]$. Since i is increasing, we have already calculated the length of the largest prefix of suffix$_{i-L+1}$ which has the same OPR as some prefix of the main string S and that length is $Z(i - L + 1)$. Therefore $S[i, R]$ has the same OPR as $S[i - L + 1, R - L + 1]$ and we also know $S[i - L + 1, i - L + Z(i - L + 1)]$ has the same OPR as $S[1, Z(i - L + 1)]$. Hence we can say that, $\sigma(S[i, i + M - 1]) = \sigma(S[1, M])$ for $M = \min(\mathrm{R - i + 1, Z(i - L + 1)})$. So M is set as the lower bound in line 7. However, the real $Z(i)$ may be larger. So in the **while** loop at line 9, we check whether $Z(i)$ can be incremented by 1. When we reach the string boundary or $Z(i)$ can not be incremented more maintaining the invariant, we break the loop.

Now, according to Lemma 1, if $\sigma(S[i, i + Z(i)]) \neq \sigma(S[1, Z(i) + 1])$ then for all the values $j > Z(i)$ we can say that $\sigma(S[i, i + j - 1]) \neq \sigma(S[1, j])$. So we will have the correct value in $Z(i)$ as soon as execution of the *while* loop of line 9 is complete. For the efficiency of the algorithm we try to update the value of L and R in the **if** block of line 12 which will be discussed shortly in Section 3.5.

3.4. OPR matching

In the condition of the **while** loop at line 9 of Algorithm 1, we compare the OPRs of two strings $S[i, i + Z(i)]$ and $S[1, Z(i) + 1]$. Using a balanced binary search tree $\mathcal{T}$ that supports the first two operations in Table 3, we can compute the OPR of a string S of length n in $O(n \log n)$ time (Algorithm 2). Note that, all the operations in Table 3 can be implemented in logarithmic time complexity. So it will take $O(n \log n)$ time in the worst case to compute the OPRs in line 9 of Algorithm 1 and an additional $O(n)$ time to check the equality.

However, we can implement this equality checking in constant time with an additional $O(n \log n)$ preprocessing. Later, in Section 3.5, we will see that it will significantly improve the time complexity of our algorithm. However to achieve this improvement, we have to use the tree $\mathcal{T}$ which can perform all the operations of Table 3. The preprocessing phase processes the initial string S and computes two arrays named **Prev** and **Next**.

Table 3
Operations of tree $\mathcal{T}$.

Function	Description
Insert(x, i)	Inserts a (key, value) pair (x, i) in the tree $\mathcal{T}$
Rank(x)	Calculates the rank of the number x among the keys present in $\mathcal{T}$. In other words, it returns number of the keys in $\mathcal{T}$ that are at least x
PreviousIndex(x)	Finds the value of the pair having largest key less than x.
NextIndex(x)	Finds the value of the pair having smallest key greater than x.

Algorithm 3 Preprocessing phase.

```
procedure PREPROCESS($S$)
    $\mathcal{T}$ := Empty Tree
    $\mathcal{T}$.Insert($-\infty, -\infty$)
    $\mathcal{T}$.Insert($\infty, \infty$)
    for $i := 1$ to $|S|$ do
        $\mathcal{T}$.Insert($S[i], i$)
        Prev($i$) := $\mathcal{T}$.PreviousIndex($S[i]$)
        Next($i$) := $\mathcal{T}$.NextIndex($S[i]$)
    end for
    return (Prev, Next)
end procedure
```

Algorithm 4 Improved algorithm for constructing modified Z-function.

```
1:  procedure ZFUNCTION($S$)
2:      (Prev, Next) = Preprocess($S$)
3:      $Z(1) := |S|$
4:      $L := R := 1$
5:      for $i := 2$ to $|S|$ do
6:          $Z(i) := 0$
7:          if $i \le R$ then
8:              $Z(i) := \min(R - i + 1, Z(i - L + 1))$
9:          end if
10:         while $i + Z(i) \le |S|$ and $S[\mathrm{Prev}[Z(i)+1] + i - 1] < S[i + Z(i)] < S[\mathrm{Next}[Z(i)+1] + i - 1]$ do
11:             $Z(i) := Z(i) + 1$
12:         end while
13:         if $i + Z(i) - 1 > R$ then
14:             $L := i$
15:             $R := i + Z(i) - 1$
16:         end if
17:     end for
18:     return $Z$
19: end procedure
```

Here, Prev(i) is the unique j such that $j < i$ and $S[j]$ is the largest value that is less than $S[i]$. Similarly, Next(i) is the unique j such that $j < i$ and $S[j]$ is the smallest value that is greater than $S[i]$. This preprocessing algorithm is formally presented in Algorithm 3.

With the help of **Prev** and **Next** arrays we now can check the equality of two OPRs in constant time. In line 9 of Algorithm 1, we know that $\sigma(S[i, i + Z(i) - 1]) = \sigma(S[1, Z(i)])$ and we would like to increase the value of $Z(i)$ by one. Instead of appending the next number to the strings and evaluating the entire OPRs, we proceed as follows. We simply find the position of the next number in the already calculated OPRs of the strings $S[i, i + Z(i) - 1]$ and $S[1, Z(i)]$. This is where our preprocessing comes handy. From the precalculated **Prev** and **Next** arrays we know that immediate smaller and larger values of $S[Z(i) + 1]$ in $S[1, Z(i)]$ are at $\mathrm{Prev}[Z(i) + 1]$ and $\mathrm{Next}[Z(i) + 1]$ respectively. So if the immediate smaller and larger values of $S[i + Z(i)]$ are at the corresponding places of $S[i, i + Z(i)]$ that is at $\mathrm{Prev}[Z(i) + 1] + i - 1$ and $\mathrm{Next}[Z(i) + 1] + i - 1$ respectively, we can say that the OPRs of the new strings will also be same. Since $\sigma(S[i, i + Z(i) - 1]) = \sigma(S[1, Z(i)])$, it would be enough to check if $S[\mathrm{Prev}[Z(i) + 1] + i - 1] < S[i + Z(i)] < S[\mathrm{Next}[Z(i) + 1] + i - 1]$. This follows readily following a similar line of arguments as discussed in the proof of Lemma 1. Our improved algorithm is presented in Algorithm 4.

3.5. Analysis

The time complexity analysis of Algorithm 4 is a bit tricky. Firstly, the preprocessing phase takes $O(n \log n)$ time. Now, if $Z(i)$ is set to the value of $R - i + 1$ in line 8, then, the value of $Z(i)$ may increase in the following **while** loop and this results in the same amount of increase in the value of R in line 15. However, if $Z(i)$ is not set to $R - i + 1$ then $Z(i)$ will not increase. Condition checking inside the **if** blocks or **while** takes constant time. We can say that the number of iterations of the **while** loop is equal to $R + O(n)$. But $R \le |S|$. Hence the total running time of Algorithm 4 is $O(n \log n)$. The space complexity of the tree $\mathcal{T}$ is $O(n)$. For the arrays Z, Prev and Next we need $O(n)$ memory. So the space complexity of the algorithm is $O(n)$.

Table 4
Example of the difficulty in the order preserving scenario.

S	=	(1,	2,	4,	3,	5,	6,	7,	8,	9)
$S[1, 3]$	=	(1,	2,	4)						
$S[4, 6]$	=				(3,	5,	6)			
$S[7, 9]$	=							(7,	8,	9)
$S[1, 6]$	=	(1,	2,	4	3,	5,	6)			
$S[4, 9]$	=				(3,	5,	6,	7,	8,	9)

4. Regularities

String regularities have been the focus of attention of the pattern matching community since long. And apart from periodicity, the most notable and studied regularities of a string are borders and covers. Since there already exist linear time algorithms for computing *Covers* for normal strings, one might wonder whether those algorithms can be easily extended for the order preserving framework. Notably, these algorithms mostly rely on the *failure function*, also known as the *border array*, used in the famous KMP algorithm for pattern matching by Knuth et al. [7]. However, although we have a modified algorithm for computing the failure function along with a modified KMP algorithm for the order preserving framework [6], as it turns out, it is not straightforward (or even possible) to compute a *Cover* extending the techniques for normal strings. The example in Table 4 sheds some light on why it is difficult to find a order preserving cover of a string.

Consider the string $S = (1, 2, 4, 3, 5, 6, 7, 8, 9)$ in Table 4. We can see that all the substrings $S[1, 3] = (1, 2, 4), S[4, 6] = (3, 5, 6)$ and $S[7, 9] = (7, 8, 9)$ are OPR-equal, i.e., they have the same OPR. Now, note that $S[1, 6]$ ($S[4, 9]$) are concatenation of $S[1, 3]$ and $S[4, 6]$ ($S[4, 6]$ and $S[7, 9]$). But, $S[1, 6]$ and $S[4, 9]$ are not OPR-equal despite that both of those are concatenation of two OPR-equal substrings. So, in general, even if we have three OPR-equal strings A, B and C, it might be possible that AB and BC are not OPR-equal.

In what follows we use the modified Z-function to compute some order preserving regularities. We start with an order preserving matching algorithm in the following subsection and then consider a number of regularities one by one.

4.1. Order preserving pattern matching

Let P be a pattern of length m and T be a text of length n. We want to find all the OPR-occurrences[3] of P in T. In other words, we have to report all the indices $i \le |T| - |P| + 1$ such that $\sigma(P) = \sigma(T[i, i + |P| - 1])$. This problem is already efficiently solved by Kubica et al. [8] and Kim et al. [6]. However we present yet another efficient algorithm for OPPM using the modified Z-function. We concatenate P and T and thus form a new string S. We calculate the modified Z-function for S. Now, if $Z(i) \ge |P|$ for $i > |P|$, then we can say that P occurs at index $i - |P|$ of T, i.e., P matches with the substring $T[i - |P|, i - 1]$. Clearly, assuming that we have the modified Z-function computed, the matching can be done in $O(|S|) = O(\mathrm{n} + \mathrm{m})$ time.

[3] We can easily extend the notion of OPR-equality to get the notion of OPR-occurrences in the context of the order preserving pattern matching.

Algorithm 5 Prints all the periods of T.

```
procedure CalculatePeriod($T$)
    $Z := ZFunction(T)$
    for $i := 1$ to $|T|$ do
        $flag := 1$
        for $j := 1$ to $|T|$ step $i$ do
            if $Z(j) < i$ then
                $flag = 0$
                break
            end if
        end for
        if $flag = 1$ then
            Write $i$
        end if
    end for
end procedure
```

Algorithm 6 Reports maximum border for all the prefixes of S.

```
procedure CalculateBorderArray($S$)
    $Z := ZFunction(S)$
    for $i := 1$ to $|S|$ do
        $BorderArray[i] = 0$
    end for
    for $i := 2$ to $|S|$ do
        if $BorderArray[i + Z(i) - 1] < Z(i)$ then
            $BorderArray[i + Z(i) - 1] = Z(i)$
        end if
    end for
end procedure
```

4.2. Order preserving periods

Let P and T be two strings such that $T = P^k$ for some $k > 0$, i.e., T is k consecutive repetitions of P. Then, P is a period of any prefix of T. However, in case of order preserving period, we can repeat any P' such that $\sigma(P) = \sigma(P')$. We give an example below.

Example 1. Suppose, $P = (1, 3, 2)$ and $k = 3$. Let $T = (1, 3, 2, 4, 10, 9, 5, 11, 7)$. Now $P = (1, 3, 2)$ is an order preserving period of any prefix of T. That is, $(1, 3, 2)$ is an order preserving period of $(1, 3, 2, 4, 10, 9, 5, 11, 7)$, $(1, 3, 2, 4)$, $(1, 3)$, $(1, 3, 2, 4, 10, 9, 5, 11)$ etc.

Now, we are only interested in the lengths of the order preserving periods. Also the period length of greater than the original string T is trivial. So, in Algorithm 5 we present an algorithm that reports all the (order preserving) period lengths less than or equal to $|T|$.

It is easy to see that the running time of this algorithm is:

$$\mathrm{O}(\mathrm{n}\log\mathrm{n}) + \left(\frac{\mathrm{n}}{1} + \frac{\mathrm{n}}{2} + \cdots + \frac{\mathrm{n}}{\mathrm{n}}\right) = \mathrm{O}(\mathrm{n}\log\ \mathrm{n}).$$

4.3. Order preserving borders and the border array

Clearly, for a string S, we have an order preserving border of length $L < |S|$ if and only if $\sigma(S[1, L]) = \sigma(S[|S| - L + 1, |S|])$. Once the modified Z-function is computed, finding an order preserving border for a string is quite simple. For each $i \le |S|$ we just need to check whether $i + Z(i) - 1 = |S|$ and if so, then there is a border of length $Z(i)$.

Similarly, we can easily compute the order preserving border array by $\mathrm{O}(|S|)$ scan of the Z-function. The algorithm is formally presented in Algorithm 6. Notably, we can easily modify the above method to compute the number of borders, the minimum border (length >1), or to compute and report all the borders.

Algorithm 7 Prints all the lengths of cover of S.

```
procedure CalculateCover($S$)
    $Z := ZFunction(S)$
    $\mathcal{C} :=$ an empty balanced binary search tree
    $\mathcal{C}_2 :=$ an empty balanced binary search tree
    Sort $(Z(i), i)$ tuples into array $A$ in descending order of $Z(i)$s.
    insert 1 into $\mathcal{C}$
    insert $|S| + 1$ into $\mathcal{C}$
    insert $|S|$ into $\mathcal{C}_2$
    for $i := 2$ to $|S|$ do
        $(Z(j), j) := (i - 1)$-th element of $A$
        $L :=$ find immediate smaller value than $j$ in $\mathcal{C}$
        $R :=$ find immediate greater value than $j$ in $\mathcal{C}$
        insert $j$ into $\mathcal{C}$
        erase $R - L$ from $\mathcal{C}_2$
        insert $j - L$ into $\mathcal{C}_2$
        insert $R - j$ into $\mathcal{C}_2$
        $M :=$ maximum key in $\mathcal{C}_2$
        if $M \le Z(j)$ and $Z(|S| - Z(j) + 1) = Z(j)$ then
            Write $Z(j)$
        end if
    end for
end procedure
```

4.4. Order preserving covers

A (proper) substring C of a string S is a cover of S, if and only if every number in S is covered by an OPR-occurrence of C in S. For example, suppose, $S = (1, 3, 2, 7, 5, 8, 6)$. Then, $(1, 3, 2)$ is a cover of S, since $\sigma([1, 3, 2]) = \sigma([2, 7, 5]) = \sigma([5, 8, 6])$. In Algorithm 7 we calculate all possible lengths for covers. We are only interested with the covers of length less than $|S|$. We note that Algorithm 7 may print same values more than once. This apparent glitch can be easily fixed by keeping the last printed value in a temporary variable. However, we have tried to keep the algorithm simple for the ease of understanding of the readers.

The basic idea of Algorithm 7 is as follows. We consider each value of the Z-function in decreasing order. We start with $Z_{\max}$, which denotes the maximum value of the Z-function. Suppose $Z(\ell) = Z_{\max}$. Now we need to deal with two intervals, namely, $[1, \ell - 1]$ and $[\ell, n]$. At this point, if the prefix $S[1, Z_{\max}]$ is also a border of S and the substring $S[1, Z_{\max}]$ can cover the above intervals, then, we can definitely report $Z_{\max}$ as (the length of) a cover. Now, consider any value $Z(k) = z$ at some later iterations. So, we have already handled values $z' > z$. At this point we have some more intervals (created by the position k within the intervals that we get from the previous iteration) to check. But clearly if we can check the maximum of the intervals, we are done. This is exactly what is done in Algorithm 7. To handle the values of the Z-function in decreasing order we use a heap and to keep track of the intervals we use two binary search trees.

Now let us analyze the algorithm. Here we have two balanced binary search trees (BST). We go through the values of $Z(j)$ in a decreasing order. In one BST $\mathcal{C}$, we keep some anchor points of the string S (initially, we have only two anchors, namely, at 1 and at $|S| + 1$). In the other BST $\mathcal{C}_2$, we keep the distances between two consecutive anchor points in $\mathcal{C}$ (since initially there are only two anchors in $\mathcal{C}$, we will have $|S| + 1 - 1 = |S|$ in $\mathcal{C}_2$). While processing $Z(j)$, we look for the immediate previous anchor point (L) and immediate next anchor point (R). We now insert our new anchor point j into $\mathcal{C}$ and thus we have to update the distances in $\mathcal{C}_2$. We need to do the following operations on $\mathcal{C}_2$: (1) we remove the distance between L and R (2) we insert the distance between L and j and (3) we insert the distance between j and R. Now we are ready to check if $Z(i)$ can be the length of a cover. To do so, we compare the maximum element in $\mathcal{C}_2$ and $Z(j)$. If all the

Table 5
Finding the order preserving covers of a string.

i	$Z(j)$	j	$\mathcal{C}$	L	R	$\mathcal{C}_2$	M	Cover
2	4	3	1,8	1	8	2,5	5	
3	3	5	1,3,8	3	8	2,3	3	3
4	1	2	1,3,5,8	1	3	1,3	3	3
5	1	4	1,2,3,5,8	3	5	1,3	3	3
6	1	6	1,2,3,4,5,8	5	8	1,2	2	3
7	1	7	1,2,3,4,5,6,8	6	8	1	1	3,1

consecutive distances between already processed indices (anchors) are less than or equal to $Z(j)$ and also the pattern is a suffix of the main string, then $Z(j)$ is the length of a cover.

Sorting the $(Z(i), i)$ tuples takes $O(n \log n)$ time. All the operations of $\mathcal{C}$ and $\mathcal{C}_2$ can be done in logarithmic time. So the time complexity of this algorithm is $O(n \log n)$, where n is the size of the input string. An illustrative example of Algorithm 7 is presented in Table 5 for the input string $S = (1, 3, 2, 7, 5, 8, 6)$. The table shows the values in each step of the iteration of the **For** loop of the algorithm. The lengths of the covers are shown in the last column.

5. Experiments

We have done some experiments to compare the performance of the algorithms we have presented in this paper. All our algorithms, namely the pattern matching and the regularities algorithms are based on the modified Z-function. And in those algorithms the running time of the modified Z-function will remain as the dominant component. So, we have only conducted experiments on the algorithms for the modified Z-function presented in this paper. At this point, recall that our algorithms assume all the numbers in a string to be distinct. However, since the data we have used in our experiments may have repeated values we augment the values with their respective indices in the string to make those distinct.

We have implemented Algorithms 1 and 4 in C programming language and have compiled the code using Microsoft Visual C++ 2010 express. We have implemented two versions of Algorithms 1: in the first version, we have used a naive approach to compute OPR while in the second we use *ComputeOPR* routine shown in Algorithm 2. In our experiments, we have referred to the former as "Algorithm 1" and the latter as "Algorithm 1 using COPR". Our experiments have been run on a Windows 7 64-bit machine with 2.40GHz Intel(R) Core(TM) i3 processor (4 CPUs) and 4GB RAM.

We have collected stock market data (closing prices) from Dhaka Stock Exchange (DSE) for the last 4 years for a particular instrument. However, the number of data points in the above data is too small to obtain a meaningful comparison. As such, from the closing prices data mentioned above, we have synthesized per minute price data applying the following constraints: the price does not vary by more than 10%/min and if the price increases (decreases) in the current tick, then there is 75% probability of price decreasing (increasing) in the next tick. This gives us a series of approximately 450k data points. We will refer to this data as the *semi-synthetic stock market data* henceforth. Using this series, we have run the algorithms for different values of n. For each value of n, we have performed 10 runs and in each run we have used n data points of the series, starting from a random offset. Finally, we have averaged the runtime across these runs. The results are shown in Fig. 2.

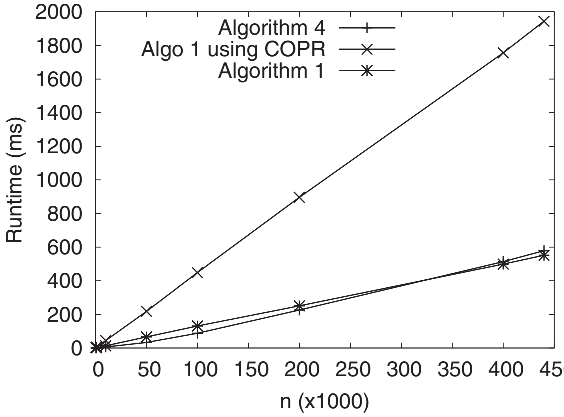


Fig. 2. Runtime comparison of the Z-function algorithms on semi-synthetic stock market data.

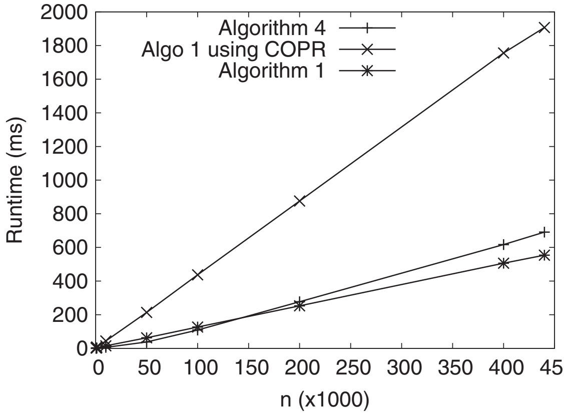


Fig. 3. Runtime comparison of the Z-function algorithms on fully synthetic data.

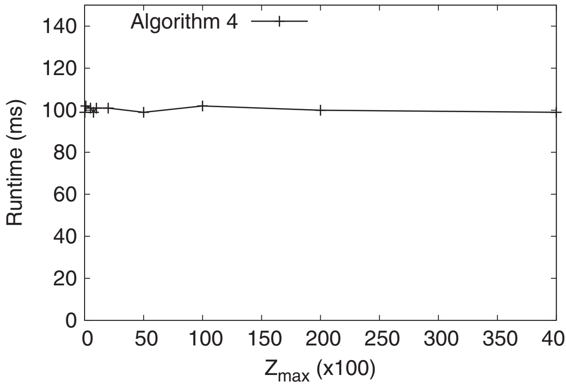


Fig. 4. Impact of $Z_{\max}$ on runtime of Algorithm 4.

From Fig. 2, we find that Algorithm 1 (i.e., the naive approach) is much faster than Algorithm 1 with COPR (i.e., where *ComputeOPR* has been employed), which seems counter-intuitive at first. This trend is also observed in a comparison run on fully synthetic data, as seen in Fig. 3. We further notice that, the runtime of Algorithm 4 is the smallest for smaller values of n; but for higher values of n, it takes longer than Algorithm 1. This cross over point was different in case of the semi-synthetic stock data and the fully synthetic data. Both these phenomena can be explained as follows. The runtime of Algorithm 1 is actually $O(nZ_{\max}^2)$, where $Z_{\max}$ is the maximum value of the Z-function. If we use *ComputeOPR* the runtime becomes $O(nZ_{\max} \log(Z_{\max}))$. But, when $Z_{\max}$ is small, construction and maintenance overhead of the order statics tree is significant. As such, the naive approach to compute OPR turns out to be better. Indeed, this is observed in Figs. 4 and 5, where the algorithms were run on 100k data points with varied values of $Z_{\max}$. Runtime of Algorithm 4 remains constant with variation of $Z_{\max}$, because it is independent of $Z_{\max}$. But, the runtime of Algorithm 1 grows rapidly with the increase in $Z_{\max}$. The growth rate is significantly higher in the naive variation (for $Z_{\max} \geq 5000$, the naive variation did not terminate in more than 2 min).

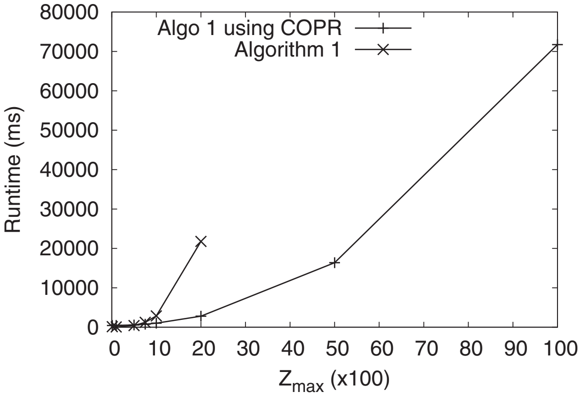


Fig. 5. Impact of $Z_{\max}$ on runtime of Algorithm 1.

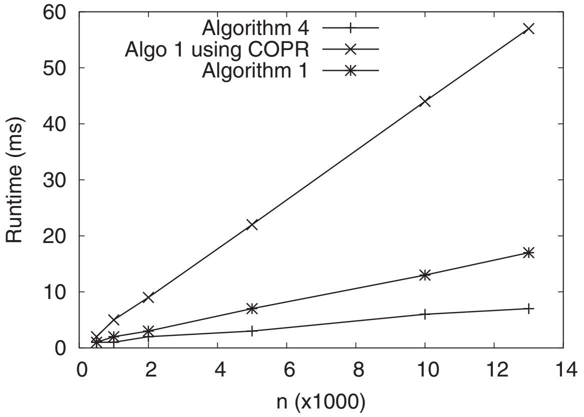


Fig. 6. Runtime comparison of the Z-function algorithms on daily opening price of GE since 1962.

Finally, we managed to obtain the daily opening price of General Electric Company (GE) stock since 1962. The performance of our algorithms on this data has been captured in Fig. 6. The results reported in Fig. 6 is consistent with the results from earlier experiments with semi-synthetic stock market data and fully synthetic data as discussed below. Algorithm 1 with COPR shows inferior performance, compared to the naive algorithm, i.e., Algorithm 1, This is because $Z_{\max}$ is too small in this data set as well. For example, in case of $n = 10,000$, $Z_{\max}$ is only 12 in this dataset.

For further analysis, the data and code used in our experiments have been made available at http://goo.gl/4L5WJA.

6. Discussion

Now that we have presented our algorithms a brief discussion especially with respect to the works already done in the literature is in order. Since our algorithm to solve OPPM problem is based on the modified Z-function, the total running time of our pattern matching algorithm is $O(n\log n + m)$. On the other hand, Kubica et al. [8] claim a linear time algorithm. To this end, we note the following. Cho et al. [1] showed that the method of Kubica et al. [8] may decide incorrectly when there are same characters. For such inaccuracies, the claim of Kubica et al. [8] that the order preserving border array can be computed in linear time and the related results regarding order preserving periods do not hold as well.

The pattern matching algorithm of Kim et al. [6] is slightly better than ours. However, they do not consider the regularities of strings from the order preserving point of view, which we believe is an important contribution of this paper. In fact to the best of our knowledge this is the first attempt to consider the order preserving borders, order preserving border arrays and order preserving covers. Finally, we note that the modified Z-Algorithm presented in this paper is of independent interest.

7. Conclusion

In this paper we have revisited the order preserving pattern matching problem and have presented some interesting algorithms. We have also studied the string regularities from order preserving point of view. The presented algorithms are efficient and interesting. We have also implemented our algorithms and have reported some interesting insight regarding out algorithms. We plan to deploy a full-phased open source software that will be able to do the order preserving pattern matching as well as visually show different order preserving regularities. We believe such a software would be of interest to the people interested in stock market.

References

[1] S. Cho, J.C. Na, K. Park, J.S. Sim, Fast order-preserving pattern matching, in: P. Widmayer, Y. Xu, B. Zhu (Eds.), COCOA, Springer, 2013, pp. 295–305.
[2] M. Crochemore, C.S. Iliopoulos, T. Kociumaka, M. Kubica, A. Langiu, S.P. Pissis, J. Radoszewski, W. Rytter, T. Walen, Order-preserving incomplete suffix trees and order-preserving indexes, in: O. Kurland, M. Lewenstein, E. Porat (Eds.), SPIRE, Springer, 2013, pp. 84–95.
[3] M. Crochemore, C.S. Iliopoulos, T. Kociumaka, M. Kubica, A. Langiu, S.P. Pissis, J. Radoszewski, W. Rytter, T. Walen, Order-preserving suffix trees and their algorithmic applications, CoRR (2013) abs/1303.6872.
[4] P. Gawrychowski, P. Uznanski, Order-preserving pattern matching with k mismatches. CoRR (2013) abs/1309.6453.
[5] D. Gusfield, Algorithms on Strings, Trees, and Sequences – Computer Science and Computational Biology, Cambridge University Press, 1997.
[6] J. Kim, P. Eades, R. Fleischer, S.H. Hong, C.S. Iliopoulos, K. Park, S.J. Puglisi, T. Tokuyama, Order-preserving matching, Theor. Comput. Sci. 525 (2014) 68–79.
[7] D.E. Knuth, J.H. Morris Jr., V.R. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6 (1977) 323–350.
[8] M. Kubica, T. Kulczynski, J. Radoszewski, W. Rytter, T. Walen, A linear time algorithm for consecutive permutation pattern matching, Inf. Process. Lett. 113 (2013) 430–433.
[9] W.F. Smyth, Computing Patterns in Strings, Pearson Addison-Wesley, 2003.
[10] W.F. Smyth, Computing regularities in strings: a survey, Eur. J. Comb. 34 (2013) 3–14.