An Efficient Skip-Search Approach to the Order-Preserving Pattern Matching Problem⋆

Domenico Cantone[1], Simone Faro[1], and M. Oğuzhan Külekci[2]

[1] Università di Catania, Department of Mathematics and Computer Science, Italy
[2] ERLAB Software Co. ITU ARI Teknokent, İstanbul, Turkey
{cantone, faro}@dmi.unict.it, oguzhankulekci@gmail.com

Abstract. Given a pattern and text, both over a common ordered alphabet, the *order-preserving pattern matching* problem consists in finding all substrings of the text with the same relative order as the pattern. This problem, an approximate variant of the well-known *exact pattern matching* problem, finds applications in such fields as time series analysis (e.g., share prices on stock markets), weather data analysis, musical melody matching, etc., and has gained increasing attention in recent years. In this paper we present a new efficient approach to this problem inspired to the well-known Skip Search algorithm for the exact string matching problem. It makes use of efficient SIMD SSE instructions in order to speed up the searching phase. Experimental results show that our proposed algorithm is up to twice as faster than previous solutions.

1 Introduction

Given a pattern x of length m and a text y of length n, both over a common alphabet Σ, the *exact string matching problem* consists in finding all occurrences of the string x in y. String matching is a very important source of challenging problems in the wider domain of text processing. String matching algorithms are often basic components in practical softwares existing under most operating systems They also emphasize programming methods that serve as paradigms in other fields of computer science.

The worst-case lower bound $\mathcal{O}(n)$ for the string matching problem has been achieved for the first time by the well-known Knuth-Morris-Pratt algorithm [13] (KMP, for short). However, several string matching algorithms with a sublinear $\mathcal{O}(n \log m/m)$ performance on average have also been developed over the years. Among them, the Boyer-Moore algorithm [2] deserves a special mention, since it has been particularly successful and has inspired much work.

The *order-preserving pattern matching problem* [2,3,8,9] (OPPM, in short) is an approximate variant of the exact pattern matching problem in which the pattern x and text y are drawn from a totally ordered alphabet Σ and one is searching for all the substrings of y with the same relative order as x. For instance, when the alphabet is the set $\mathbb{N}$ of natural numbers with the standard order relation, the relative order of the sequence $x = \langle 6, 5, 8, 4, 7\rangle$ is the sequence $\langle 2, 1, 4, 0, 3\rangle$ since 6 has rank 2, 5 has rank 1, and so on. Thus x has an order-preserving occurrence in the string

$$y = \langle 8, 11, 10, 16, 15, 20, 13, 17, 14, 18, 20, 18, 25, 17, 20, 25, 26\rangle$$

at position 3, since x and the subsequence $\langle 16, 15, 20, 13, 17\rangle$ share the same relative order. Another order-preserving occurrence of x in y is at position 10 (see Fig. 1).

⋆ This work has been supported by the Scientific & Technological Research Council Of Turkey (TUBITAK), the Department Of Science Fellowships & Grant Programs (BIDEB), 2221 Fellowship Program, and by G.N.C.S., Istituto Nazionale di Alta Matematica "Francesco Severi".

Domenico Cantone, Simone Faro, M. Oğuzhan Külekci: An Efficient Skip-Search Approach to the Order-Preserving Pattern Matching Problem, pp. 22–35.
Proceedings of PSC 2015, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05787-2

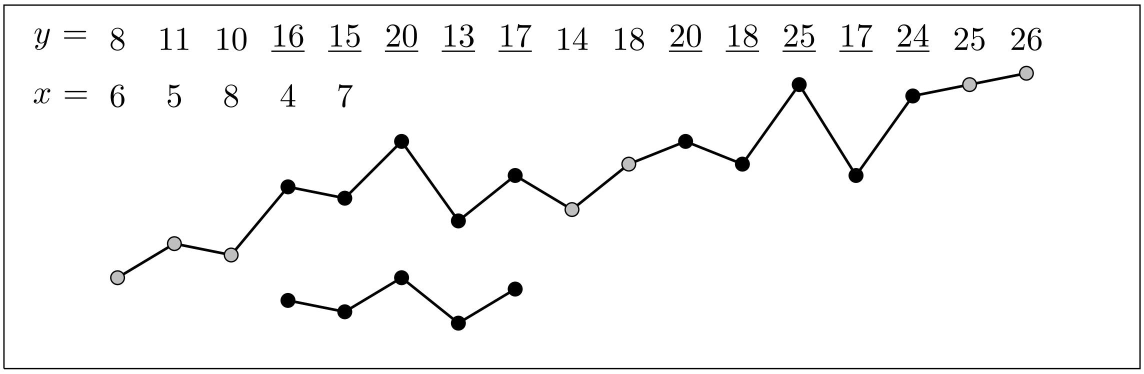

Figure 1. Example of a pattern x of length 5 over an integer alphabet with two order preserving occurrences in a text y of length 17, at positions 3 and 10.

The OPPM problem finds applications in all situations in which one is interested only in the "shape" of the pattern (intended as the relative order of its characters) rather than in the pattern itself. For instance, it can be applied successfully to time series analysis like share prices on stock markets and weather data, or to melody matching of musical scores.

In the last few years some solutions have been proposed for the OPPM problem. The first solution was presented by Kubica *et al.* [12] in 2013. They proposed a $\mathcal{O}(n+m\log m)$ solution over generic ordered alphabets based on the KMP algorithm [13] and a $\mathcal{O}(n+m)$ solution in the case of integer alphabets. A few months later, Kim *et al.* presented in [11] a similar solution running in $\mathcal{O}(n+m\log m)$ time, still based on the KMP approach. Although Kim *et al.* stressed some doubts about the applicability of the Boyer-Moore approach [2] to the OPPM problem, in 2013 Cho *et al.* [5] presented a method for deciding the order-isomorphism between two sequences showing that the Boyer-Moore approach can be applied also to the order-preserving variant of the pattern matching problem. More recently, Chhabra and Tarhio [4] presented a more practical solution based on approximate string matching techniques. Specifically, their solution consists in converting the input sequences into binary sequences and then applying any standard algorithm for exact string matching as a filtration method.

In this paper we present a new algorithm for the OPPM problem which turns out to be more effective in practice than currently available solutions. Our proposed algorithm is based on the well-known Skip Search algorithm [3] for the exact string matching problem, which consists in processing separately chunks of the text for any occurrence of the pattern. For all substrings of a given length of the pattern, a fingerprint is computed and indexed. Then, by using this information, candidate occurrences of the pattern are located in the text. We propose to use efficient SIMD SSE instructions [10] for computing the fingerprints of the pattern substrings. Experimental results show that our proposed approach leads to algorithmic variants that are up to twice as faster than previous solutions present in the literature.

The paper is organized as follows. In Section 2 we review some preliminary notions and properties relative to the OPPM problem and give an overview of the Skip Search algorithm and its searching approach. We present our proposed algorithm for the OPPM problem in Section 3 and then compare its performance with previous known algorithms in Section 4. Finally, we draw our conclusions in Section 5.

2 Preliminaries

A string x over an ordered alphabet Σ, of size σ, is defined as a sequence of elements in Σ. We shall assume that a total order relation "$\preceq$" is defined on it.

By $|x|$ we denote the length of a string x. We refer to the i-th element in x as $x[i]$ and use the notation $x[i\,..\,j]$ to denote the subsequence of x from the element at position i to the element at position j (including the extremes), where $0 \leq i \leq j < |x|$.

2.1 Order-Isomorphism and Related Properties

We say that two (nonnull) sequences x, y over Σ are order-isomorphic if the relative order of their elements is the same. More formally:

Definition 1 (Order-isomorphism). *Two nonnull sequences x, y of the same length, over a totally ordered alphabet $(\Sigma, \preceq)$, are said to be* order-isomorphic, *and we write $x \approx y$, if the following condition holds*

$$\text{for } 0 \leq i, j < |x|\,, \qquad x[i] \preceq x[j] \quad \Longleftrightarrow \quad y[i] \preceq y[j].$$

The following lemma states some elementary properties of order-isomorphism which follow directly from the definition.

Lemma 2. *Let x and y be two nonnull sequences of the same length, over a totally ordered alphabet $(\Sigma, \preceq)$, such that $x \approx y$. Then*

(a) $x[j] \prec x[i]$ iff $y[j] \prec y[i]$, for $0 \leq i, j < |x|$;
(b) $x[j] = x[i]$ iff $y[j] = y[i]$, for $0 \leq i, j < |x|$. □

From a computational point of view, it is convenient to characterize the order of a sequence by means of two functions: the *rank* and the *equality* functions. These are defined below, together with some of their elementary properties.

Definition 3 (Rank function). *Let x be a nonnull sequence over a totally ordered alphabet $(\Sigma, \preceq)$. The* rank function of x *is the bijection from $\{0, 1, \ldots, |x| - 1\}$ onto itself defined, for $0 \leq i < |x|$, by*

$$rk_x(i) \;=_{Def} \left|\{k : x[k] \prec x[i] \text{ or } (x[k] = x[i] \text{ and } k < i)\}\right|.$$

The following properties are easy consequences of Definition 3.

Lemma 4. *Given a nonnull sequence x over a totally ordered alphabet $(\Sigma, \preceq)$, we have:*

(a) if $x[j] \prec x[i]$, then $rk_x(j) < rk_x(i)$, for $0 \leq i, j < |x|$;
(b) if $x[j] = x[i]$ and $0 \leq j < i < |x|$, then $rk_x(j) < rk_x(i)$. □

Corollary 5. *Let x be a nonnull sequence over a totally ordered alphabet $(\Sigma, \preceq)$. Then we have $x[rk_x^{-1}(i)] \preceq x[rk_x^{-1}(i+1)]$, for $0 \leq i < |x| - 1$.* □

For any nonnull sequence x, we shall refer to the sequence

$$\langle rk_x^{-1}(0), rk_x^{-1}(1), \ldots, rk_x^{-1}(|x| - 1)\rangle$$

as the *relative order* of x (see Example 8).

From Corollary 5, it follows that the relative order of x can be computed in time proportional to the time required to (stably) sort x.

The rank function alone allows one to characterize order-isomorphic sequences only when characters are pairwise distinct. To handle the more general case in which multiple occurrences of the same character are permitted, we also need the *equality function*.

Definition 6 (Equality function). *Let x be a sequence of length $m \geq 2$ over a totally ordered alphabet $(\Sigma, \preceq)$. The* equality function *of x is the binary map $eq_x \colon \{0, 1, \ldots, m-2\} \rightarrow \{0, 1\}$ where, for $0 \leq i \leq m-2$,*

$$eq_x(i) =_{Def} \begin{cases} 1 & \text{if } x[rk_x^{-1}(i)] = x[rk_x^{-1}(i+1)] \\ 0 & \text{otherwise.} \end{cases}$$

The rank and equality functions allow to fully characterize order-isomorphism, as stated in the following lemma, whose proof can be found in the Appendix.

Lemma 7. *For any two sequences x and y of the same length $m \geq 2$, over a totally ordered alphabet, we have*

$$x \approx y \quad \text{iff} \quad rk_x = rk_y \text{ and } eq_x = eq_y\,. \qquad \square$$

Example 8. Consider the following three sequences of length 7:

$$x = \langle 6, 3, 8, 3, 10, 7, 10 \rangle, \quad y = \langle 2, 1, 4, 1, 5, 3, 5 \rangle, \quad z = \langle 6, 3, 8, 4, 9, 7, 10 \rangle.$$

They have the same rank function $\langle 2, 0, 4, 1, 5, 3, 6 \rangle$ and, therefore, the same relative order $\langle 1, 3, 0, 5, 2, 4, 6 \rangle$. However, x and y are order-isomorphic, whereas x and z (as well as y and z) are not. Notice that, in agreement with Lemma 7, we have $eq_x = eq_y = \langle 1, 0, 0, 0, 0, 1 \rangle$ and $eq_z = \langle 0, 0, 0, 0, 0, 0 \rangle$.

Based on the preceding lemma, in order to establish whether two given sequences of the same length m are order-isomorphic, it is enough to compute their rank and equality functions, and then compare them. The cost of such a test is dominated by the cost $\mathcal{O}(m \log m)$ of sorting the two sequences. However, if one needs to find all the sequences from a set $\mathcal{S}$ that are order-isomorphic to a fixed sequence (all the sequences having the same size m), the simple iteration of the previous test would lead to an overall complexity of $\mathcal{O}(|\mathcal{S}| \cdot m \log m)$. In this case a better approach is possible, based on the following characterization of order-isomorphism which requires the computation of the rank and equality functions of the fixed sequence only, yielding an overall complexity of $\mathcal{O}\big((|\mathcal{S}| + \log m) \cdot m\big)$.

Lemma 9. *Let x and y be two sequences of the same length $m \geq 2$, over a totally ordered alphabet. Then $x \approx y$ iff the following conditions hold:*

(i) $y[rk_x^{-1}(i)] \preceq y[rk_x^{-1}(i+1)]$, *for* $0 \leq i < m-1$
(ii) $y[rk_x^{-1}(i)] = y[rk_x^{-1}(i+1)]$ *if and only if* $eq_x(i) = 1$, *for* $0 \leq i < m-1$. $\square$

Based on Lemma 9, the procedure Order-Isomorphic in Fig. 2 correctly verifies whether a sequence y is order-isomorphic to a sequence x of the same length as y. It receives as input the functions rk_x and eq_x and the sequence y, and returns true if $x \approx y$, false otherwise. A mismatch occurs when one of the three conditions of lines 2, 3, or 4 holds. Notice that the time complexity of the procedure Order-Isomorphic is linear in the size of its input sequence y.

The OPPM problem consists in finding all the substrings of the text with the same relative order as the pattern. More precisely,

Definition 10 (Order-preserving pattern matching). *Let x and y be two sequences of length m and n, respectively, with $n > m$, both over an ordered alphabet $(\Sigma, \preceq)$. The* order-preserving pattern matching problem *consists in finding all positions i, with $0 \leq i \leq n-m$, such that $y[i \mathinner{..} i+m-1] \approx x$.*

```
Order-Isomorphic(inv-rk, eq, y)
for i ← 0 to |y| − 2 do
    if (y[inv-rk(i)] ≻ y[inv-rk(i + 1)]) then return false
    if (y[inv-rk(i)] ≺ y[inv-rk(i + 1)] and eq(i) = 1) then return false
    if (y[inv-rk(i)] = y[inv-rk(i + 1)] and eq(i) = 0) then return false
return true
```

Figure 2. The procedure to verify whether a sequence y is order-isomorphic to a sequence of length $|y|$, whose inverse rank and equality functions are the parameters *inv-rk* and *eq*, respectively.

If $y[i\,..\,i+m-1] \approx x$, we say that x has an *order-preserving occurrence in* y *at position* i.

In Section 3, we shall present an algorithm for the OPPM problem, based on the Alpha Skip Search algorithm. For convenience, we briefly review it next.

2.2 The Skip Search Algorithm and its Alpha Variant

The Skip Search algorithm is an elegant and efficient solution to the exact pattern matching problem, firstly presented in [3] and subsequently adapted to many other problems and variants of the exact pattern matching problem.

Let x and y be a pattern and a text of length m and n, respectively, over a common alphabet Σ of size σ. For each character c of the alphabet, the Skip Search algorithm collects in a bucket $B[c]$ all the positions of that character in the pattern x, so that for each $c \in \Sigma$ we have:

$$B[c] =_{\mathrm{Def}} \{i \ : \ 0 \leq i \leq m-1 \text{ and } x[i] = c\}.$$

Plainly, the space and time complexity needed for the construction of the array B of buckets is $\mathcal{O}(m+\sigma)$. Notice that when the pattern is shorter than the alphabet size, buckets are empty.

The search phase of the Skip Search algorithm examines all the characters $y[j]$ in the text at positions $j = km - 1$, for $k = 1, 2, \ldots, \lfloor n/m \rfloor$. For each such character $y[j]$, the bucket $B[y[j]]$ allows one to compute the possible positions h of the text in the neighborhood of j at which the pattern could occur.

By performing a character-by-character comparison between x and the subsequence $y[h\,..\,h+m-1]$ until either a mismatch is found, or all the characters in the pattern x have been considered, it can be tested whether x actually occurs at position h of the text.

The Skip Search algorithm has a quadratic worst-case time complexity, however, as shown in [3], the expected number of text character inspections is $\mathcal{O}(n)$.

Among the variants of the Skip Search algorithm, the most relevant one for our purposes is the Alpha Skip Search algorithm [3], which collects buckets for substrings of the pattern rather than for its single characters.

During the preprocessing phase of the Alpha Skip Search algorithm, all the factors of length $\ell = \lfloor \log_\sigma m \rfloor$ occurring in the pattern x are arranged in a trie T_x, for fast retrieval. In addition, for each leaf ν of T_x a bucket is maintained which stores the positions in x of the factor corresponding to ν. Provided that the alphabet size is considered as a constant, the worst-case running time of the preprocessing phase is linear.

The searching phase consists in looking into the buckets of the text factors $y[j \mathinner{..} j+\ell-1]$, for all $j = k(m-\ell+1)-1$ such that $1 \leq k \leq \lfloor (n-\ell)/m \rfloor$, and then test, as in the previous case, whether there is an occurrence of the pattern at the indicated positions of the text.

The worst-case time complexity of the searching phase is quadratic, though the expected number of text character comparisons is $\mathcal{O}(n \log_\sigma m/(m - \log_\sigma m))$.

3 A New Order-Preserving Pattern Matching Algorithm

In this section we present a new algorithm, called Order-Preserving-Skip-Search, for the Order-Preserving Pattern Matching problem. However, for brevity, in the following we shall often refer to it as SkSop algorithm.

Our algorithm combines the same approach of the Skip Search algorithm with the power of the SIMD (Single Instruction Multiple Data) instruction set, and specifically the Intel SSE (Streaming SIMD Extensions) instruction set, as discussed below.

In the last two decades a lot of effort has been spent exploiting the power of the word-RAM model of computation in order to speed-up string matching algorithms for a single pattern.

In this model, the computer operates on words of length w, so that usual arithmetic and logic operations on words all take one unit of time. Most of the solutions which exploit the word-RAM model are based on the *bit-parallelism* technique [1] or on the *packed string matching* technique [8,9]. In the packed string matching technique, multiple characters can be packed into a single word, so that the characters can be compared in bulk rather than individually.

Next we discuss our model in details.

3.1 The Model

In the design of our algorithm, we use specialized word-size packed string matching instructions, based on the Intel streaming SIMD extensions (SSE) technology. SIMD instructions exist in many recent microprocessors supporting parallel execution of some operations on multiple data via a set of special instructions working on a limited number of special registers.

In our model of computation we assume that w is the number of bits in a word and σ is the size of the alphabet. The *packing factor* $\alpha = w/\log\sigma$ (or, rather, its floor) is the number of characters which fit in a single computer word, whereas the number of bits used to encode an alphabet character is $\gamma = \log\sigma$.

In most practical applications we have $\sigma = 256$ (ASCII code). Moreover SSE specialized instructions allow one to work on 128-bit registers, so that blocks of sixteen 8-bit characters can be read and processed in a single time unit ($\alpha = 16$). In particular, our algorithm makes use of specialized word-size packed instructions which we call wsrv (*word-size rank vector*) and wsrp (*word-size relative position*). These are reviewed next.

The instruction wsrv

The instruction $\mathsf{wsrv}(B, i)$ computes an α-bit fingerprint from a w-bit register B handled as a block of α small integers values. Assuming that $B[0 \mathinner{..} \alpha-1]$ is a w-bit integer parameter, wsrv(B, i) returns an α-bit value $r[0 \mathinner{..} \alpha-1]$, where $r[j] = 1$ iff $B[i] \geq B[j]$, and $r[j] = 0$ otherwise.

The wsrv(B, i) specialized instruction can be emulated in constant time by the following sequence of specialized SIMD instructions:

wsrv(B, i)
  $D \leftarrow$ _mm_set1_epi8$(B[i])$
  $C \leftarrow$ _mm_cmpgt_epi8(B, D)
  $r \leftarrow$ _mm_movemask_epi8(C)
  return r

Specifically the _mm_set1_epi8$(B[i])$ instruction creates a w-bit register D handled as a block of α small integers values, where $D[j] = B[i]$ for $0 \leq j < \alpha$. The _mm_cmpgt_epi8(B, D) instruction compares the α integers in B and the α integers in D for "greater than". It creates a w-bit register C handled as a block of α small integers where $C[j] = 1^\gamma$ if $B[j] \geq D[j]$, and $C[j] = 0^\gamma$ otherwise, and where we remember that $\gamma = \log\sigma$ is the number of bits to encode an alphabet character. Finally, the _mm_movemask_epi8(D) instruction gets a 128 bit parameter D, handled as sixteen 8-bit integers, and creates a 16-bit mask from the most significant bits of the 16 integers in D, and zero extends the upper bits.

The instruction wsrp

The instruction wsrp(B) computes an α-bit fingerprint from a w-bit register B handled as a block of α small integers values. Assuming that $B[0 \mathinner{..} \alpha-1]$ is a w-bit integer parameter, wsrp(B) returns an α-bit value $r[0 \mathinner{..} \alpha-1]$, where $r[j] = 1$ iff $B[j] \geq B[j+1]$, and $r[j] = 0$ otherwise (we put $r[\alpha-1] = 0$).

The wsrp(B) specialized instruction can be emulated in constant time by the following sequence of specialized SIMD instructions

wsrp(B)
  $D \leftarrow$ _mm_slli_si128$(B, 1)$
  $C \leftarrow$ _mm_cmpgt_epi8(B, D)
  $r \leftarrow$ _mm_movemask_epi8(C)
  return r

where the _mm_slli_si128$(B, 1)$ instruction shifts the w-bit register in B to the left by one position (α bits) while shifting in zeros and the _mm_cmpgt_epi8 and the _mm_movemask_epi8 instructions are as described above.

3.2 The Fingerprint Functions

The preprocessing phase of the algorithm indexes the subsequences of the pattern (of length q) in order to locate them during the searching phase. For efficiency reasons, each numeric sequence of length q is converted into a numeric value, called *fingerprint*, which is used to index the substring. A fingerprint value ranges in the interval $\{0 \mathinner{..} \tau-1\}$, for a given bound τ. The value τ is set to 2^{16}, so that a fingerprint can fit into a single 16-bit register.

The procedure FNG for computing the fingerprints is shown in Fig. 3 (on the left). Given a sequence x of length m, an index i such that $0 \leq i < m-q$, and two integers k and q such that $k \leq q \leq m$, the procedure FNG combines k different values computed

FNG(x, i, q, k)
1. $B \leftarrow 0^{\alpha-q}.x[i \mathinner{..} i+q-1]$
2. $v \leftarrow \mathsf{wsrp}(B)$
3. for $j \leftarrow 0$ to $k-2$ do
4. $\quad v \leftarrow (v \ll 1) +$ $\mathsf{wsrv}(B, \alpha - q + j)$
5. return v

EXAMPLE ($q = 5$, $k = 3$, and $\alpha = 8$)

$x[i \mathinner{..} i+q-1] = \langle 3, 6, 2, 4, 7\rangle$

$B = [0, 0, 0, 3, 6, 2, 4, 7]$

$\mathsf{wsrp}(B) = [0, 0, 0, 1, 0, 1, 1, 0] = 22_{10}$

$\mathsf{wsrv}(B, 3) = [0, 0, 0, 1, 1, 0, 1, 1] = 27_{10}$

$\mathsf{wsrv}(B, 4) = [0, 0, 0, 0, 1, 0, 0, 1] = 9_{10}$

$v = 22 \times 2^2 + 27 \times 2^1 + 9 = 151_{10}$

Figure 3. On the left: the pseudo-code of the procedure FNG for the computation of the fingerprint of a substring a length q combining k distinct fingerprints. On the right: an example of a computation of a fingerprint by the procedure FNG.

on the substring $x[i \mathinner{..} i+q-1]$ in order to compute the fingerprint v. Preliminarily, the substring $x[i \mathinner{..} i+q-1]$ is inserted in the rightmost portion of a w-bit register B. Then the fingerprint v is computed as

$$v = \mathsf{wsrp}(B) \times 2^{k-1} + \sum_{j=0}^{k-2} \left(\mathsf{wsrv}(B, \alpha - q + j) \times 2^{k-2-j}\right) .$$

Plainly, the time complexity of the procedure FNG is $\mathcal{O}(k)$.

Fig. 3 (on the right) shows an example of how the procedure FNG works on a subsequence $x[i \mathinner{..} i+q-1] = \langle 3, 6, 2, 4, 7\rangle$ of length $q = 5$, combining $k = 3$ different values.

3.3 The Order-Preserving Skip Search algorithm

We are now ready to briefly describe our algorithm for the OPPM problem, based on the Alpha variant of the Skip Search algorithm. We distinguish in it a preprocessing and a searching phase.

The preprocessing phase of the SkSop algorithm, which is reported in Fig. 4 (on the left), consists in compiling the fingerprints of all possible substrings of length q contained in the pattern x. Thus a fingerprint value v, with $0 \leq v < 2^\alpha$, is computed for each subsequence $x[i \mathinner{..} i+q-1]$, for $0 \leq i < m - q$.

To this purpose a table F of size 2^α is maintained for storing, for any possible fingerprint value v, the set of positions i such that FNG$(x, i, q, k) = v$. More precisely, for $0 \leq v < 2^\alpha$, we have

$$F[v] = \Big\{ i \mid 0 \leq i < m - q \text{ and } \textsc{fng}(x, i, q, k) = v \Big\}.$$

The preprocessing phase of the SkSop algorithm requires some additional space to store the $(m - q)$ possible alignments in the 2^α locations of the table F. Thus, the space requirement of the algorithm is $\mathcal{O}(m-q+2^\alpha)$ that approximates to $\mathcal{O}(m)$, since α is constant. The first loop of the preprocessing phase just initializes the table F, while the second loop is run $(m - q)$ times, which makes the overall time complexity of the preprocessing phase $\mathcal{O}(m + 2^\alpha)$ that, again, approximates to $\mathcal{O}(m)$.

The basic idea of the searching phase is to compute a fingerprint value every $(m-q)$ positions of the text y and to check whether the pattern appears in y, involving the

```
PREPROCESSING(x, q, m, k)
1. for v ← 0 to 2^α − 1 do
2.     F[v] ← ∅
3. for i ← 0 to m − q do
4.     v ← FNG(x, i, q, k)
5.     F[v] ← F[v] ∪ {(i + q − 1)}
6. return F
```

```
SKSOP(x, r, y, n, q, k)
1. F ← Preprocessing(x, q, m, k)
2. for j ← m − 1 to n step m − q + 1 do
3.     v ← FNG(y, j, q, k)
4.     for each i ∈ F[v] do
5.         z ← y[j − i .. j − i + m − 1]
6.         if ORDER-ISOMORPHIC(rk_x^{-1}, eq_x, z)
7.             then output (j)
```

Figure 4. The pseudo-code of the SkSop algorithm for the OPPM problem.

block $y[j .. j+q-1]$. If the fingerprint value indicates that some of the alignments are possible, then the candidate positions are checked naively for matching.

The pseudo-code provided in Fig. 4 (on the right) reports the skeleton of the SkSop algorithm. The main loop investigates the blocks of the text y in steps of $(m-q+1)$ blocks. If the fingerprint v computed on $y[j .. j+q-1]$ points to a nonempty bucket of the table F, then the positions listed in $F[v]$ are verified accordingly.

In particular $F[v]$ contains a linked list of the values i marking the pattern x and the beginning position of the pattern in the text. While looking for occurrences on $y[j .. j+q-1]$, if $F[v]$ contains the value i, this indicates the pattern x may potentially begin at position $(j-i)$ of the text. In that case, a matching test is to be performed between x and $y[j-i .. j-i+m-1]$ via a character-by-character inspection.

The total number of filtering operations is exactly $n/(m-q)$. At each attempt, the maximum number of verification requests is $(m-q)$, since the filter provides information about that number of appropriate alignments of the patterns. On the other hand, if the computed fingerprint points to an empty location in F, then there is obviously no need for verification. The verification cost for a pattern x of length m is assumed to be $\mathcal{O}(m)$, with the brute-force checking approach. Hence, in the worst case the time complexity of the verification is $\mathcal{O}(m(m-q))$, which happens when all alignments in x must be verified at any possible beginning position. Hence, the best case complexity is $\mathcal{O}(n/(m-q))$, while the worst case complexity is $\mathcal{O}(nm)$.

4 Experimental Evaluations

In this section we present experimental results in order to evaluate the performance of our Skip-Search based algorithm SkSop. In particular we tested our algorithm against the filter approach of Chhabra and Tarhio [4], which is, to the best of our knowledge, the most effective solution to the OPPM problem in practical cases. In the experimental evaluations reported in [4], the SBNDM2 and SBNDM4 algorithms [7] turned out to be the most effective exact string matching algorithms which can be used in combination with the filter technique. In our experimental evaluations, we used the SBNDM2 algorithm to test the filter approach by Chhabra and Tarhio. In our dataset we use the following short names to identify the algorithms that we have tested:

- Fct: the SBNDM2 algorithm based on the filter approach by Chhabra and Tarhio presented in [4];

– SkSop(k, q): our Skip Search-based algorithm presented in Section 3, which combines k different fingerprint values on subsequences of length q.

More specifically, in our tests, we considered the SkSop(k, q) algorithm, for $k \in \{1, 2, 3, 4, 5\}$ and $q \in \{3, 4, 5, 6, 7, 8\}$.

The Boyer-Moore approach by Cho *et al.* [5] has not been included in our evaluations, as it was shown to be less efficient than the algorithm by Chhabra and Tarhio in all cases.

All algorithms have been evaluated in terms of efficiency, i.e. running times, and accuracy, i.e., number of verifications performed during the searching phase. In particular, they have been tested on sequences of small integer values (i.e., in the range $[0 \,..\, 256]$), big integer values (i.e., in the range $[0 \,..\, 10.000]$) and real numbers (i.e., in the range $[0,0 \,..\, 10.000,99]$). However, we did not observe any significant difference in the results; thus, for brevity, in the following table we report only the results relative to small integer sequences. Each text consists in a sequence of 1 million elements. In particular we tested our algorithm on the following set of small integer sequences:

– Rand-δ: a sequence of random integer values ranging around a fixed mean μ with a variability of δ and a uniform distribution, i.e. each value is uniformly distributed in the range $\{\mu - \delta \,..\, \mu + \delta\}$;
– Periodic-ρ: a sequence of random integer values uniformly ranging around a cyclic function with a period of ρ elements.

For each text in the set, we randomly selected 100 patterns extracted from the text and computed the average running time over 100 runs. We also computed the average number of false positives detected by the algorithms during the search. Algorithms have been implemented using the `C` programming language and have been compiled using the `gcc` compiler Apple LLVM version 5.1 (based on LLVM 3.4svn) with 8Gb Ram. Compilation has been performed with the `-O3` optimization option.

For each value of k, we have also reported in round parentheses the value of q which led to the best performance.

Average Number of Verifications

We evaluated the accuracy of our solutions in terms of the number of verifications performed during the search. Specifically, we counted the average number of verifications that each algorithm performs every 2^{10} text characters and computed the mean of such value over 100 runs. Table 1 and Table 2 show, respectively, the results obtained on Rand-δ sequences, with $\delta = 5, 20, 40$, and on Periodic-ρ sequences, with $\rho = 8, 16, 32$. Best results have been underlined.

Results on Rand-δ sequences (Table 1) show that the difference in the number of verifications performed during the search is sensible when different fingerprints are used. Using a single fingerprint (algorithm SkSop$(1, q)$) leads to a quite high number of verifications, up to 50 every 2^{20} characters. The best results are obtained by combining 4 different fingerprint values (algorithm SkSop$(4, q)$), in which case the number of verifications decreases to 0.25 every 2^{20} characters.

When we combine more than 4 fingerprint values (algorithm SkSop$(5, q)$), the number of verifications sensibly increases to 0.5. This behavior is due to the combination process which uses a final hash value of 16 bits and which causes loss of information.

δ	m	SkSop$(1,q)$	SkSop$(2,q)$	SkSop$(3,q)$	SkSop$(4,q)$	SkSop$(5,q)$
	8	52.64 $^{(8)}$	3.87 $^{(8)}$	1.20 $^{(8)}$	0.25 $^{(8)}$	0.43 $^{(8)}$
	12	46.22 $^{(8)}$	4.27 $^{(8)}$	1.02 $^{(8)}$	0.25 $^{(8)}$	0.41 $^{(8)}$
	16	45.65 $^{(8)}$	4.01 $^{(8)}$	1.06 $^{(8)}$	0.24 $^{(8)}$	0.41 $^{(8)}$
5	20	48.13 $^{(8)}$	4.18 $^{(8)}$	1.09 $^{(8)}$	0.24 $^{(8)}$	0.42 $^{(8)}$
	24	45.02 $^{(8)}$	4.13 $^{(8)}$	1.05 $^{(8)}$	0.24 $^{(8)}$	0.42 $^{(8)}$
	28	44.92 $^{(8)}$	4.05 $^{(8)}$	1.03 $^{(8)}$	0.24 $^{(8)}$	0.41 $^{(8)}$
	32	46.99 $^{(8)}$	4.23 $^{(8)}$	1.04 $^{(8)}$	0.23 $^{(8)}$	0.44 $^{(8)}$
	8	37.78 $^{(8)}$	3.96 $^{(8)}$	1.02 $^{(8)}$	0.23 $^{(8)}$	0.51 $^{(8)}$
	12	40.65 $^{(8)}$	4.17 $^{(8)}$	1.04 $^{(8)}$	0.25 $^{(8)}$	0.52 $^{(8)}$
	16	39.78 $^{(8)}$	4.65 $^{(8)}$	1.00 $^{(8)}$	0.25 $^{(8)}$	0.52 $^{(8)}$
20	20	39.05 $^{(8)}$	4.12 $^{(8)}$	1.02 $^{(8)}$	0.24 $^{(8)}$	0.49 $^{(8)}$
	24	39.24 $^{(8)}$	4.35 $^{(8)}$	1.02 $^{(8)}$	0.25 $^{(8)}$	0.50 $^{(8)}$
	28	40.15 $^{(8)}$	4.34 $^{(8)}$	1.00 $^{(8)}$	0.24 $^{(8)}$	0.49 $^{(8)}$
	32	40.00 $^{(8)}$	4.39 $^{(8)}$	1.01 $^{(8)}$	0.25 $^{(8)}$	0.51 $^{(8)}$
	8	42.34 $^{(8)}$	4.37 $^{(8)}$	1.03 $^{(8)}$	0.27 $^{(8)}$	0.54 $^{(8)}$
	12	35.64 $^{(8)}$	4.50 $^{(8)}$	0.99 $^{(8)}$	0.25 $^{(8)}$	0.49 $^{(8)}$
	16	41.08 $^{(8)}$	4.40 $^{(8)}$	1.01 $^{(8)}$	0.26 $^{(8)}$	0.54 $^{(8)}$
40	20	40.71 $^{(8)}$	4.29 $^{(8)}$	1.05 $^{(8)}$	0.26 $^{(8)}$	0.54 $^{(8)}$
	24	37.77 $^{(8)}$	4.33 $^{(8)}$	0.96 $^{(8)}$	0.25 $^{(8)}$	0.52 $^{(8)}$
	28	39.98 $^{(8)}$	4.51 $^{(8)}$	1.02 $^{(8)}$	0.25 $^{(8)}$	0.53 $^{(8)}$
	32	38.26 $^{(8)}$	4.46 $^{(8)}$	0.99 $^{(8)}$	0.26 $^{(8)}$	0.54 $^{(8)}$

Table 1. Average number of verifications performed every 2^{10} characters, computed on a Rand-δ small integer sequence, with $\delta = 5, 20$, and 40.

ρ	m	SkSop$(1,q)$	SkSop$(2,q)$	SkSop$(3,q)$	SkSop$(4,q)$	SkSop$(5,q)$
	8	92.22 $^{(8)}$	37.40 $^{(8)}$	14.22 $^{(8)}$	8.01 $^{(8)}$	8.83 $^{(8)}$
	12	98.48 $^{(8)}$	35.46 $^{(8)}$	14.72 $^{(8)}$	8.27 $^{(8)}$	10.38 $^{(8)}$
	16	98.27 $^{(8)}$	36.46 $^{(8)}$	15.71 $^{(8)}$	8.77 $^{(8)}$	10.46 $^{(8)}$
8	20	96.95 $^{(8)}$	35.91 $^{(8)}$	15.14 $^{(8)}$	8.47 $^{(8)}$	10.12 $^{(8)}$
	24	96.88 $^{(8)}$	36.06 $^{(8)}$	14.87 $^{(8)}$	8.34 $^{(8)}$	10.18 $^{(8)}$
	28	97.63 $^{(8)}$	35.79 $^{(8)}$	14.60 $^{(8)}$	7.94 $^{(8)}$	9.67 $^{(8)}$
	32	97.65 $^{(8)}$	35.93 $^{(8)}$	15.09 $^{(8)}$	8.31 $^{(8)}$	10.23 $^{(8)}$
	8	173.85 $^{(8)}$	40.23 $^{(8)}$	5.19 $^{(8)}$	3.74 $^{(8)}$	7.03 $^{(8)}$
	12	179.84 $^{(8)}$	46.64 $^{(8)}$	5.35 $^{(8)}$	4.25 $^{(8)}$	7.62 $^{(8)}$
	16	179.20 $^{(8)}$	46.94 $^{(8)}$	5.59 $^{(8)}$	4.35 $^{(8)}$	7.57 $^{(8)}$
16	20	176.24 $^{(8)}$	45.61 $^{(8)}$	5.40 $^{(8)}$	4.20 $^{(8)}$	7.07 $^{(8)}$
	24	181.67 $^{(8)}$	46.50 $^{(8)}$	5.53 $^{(8)}$	4.24 $^{(8)}$	7.31 $^{(8)}$
	28	176.67 $^{(8)}$	46.27 $^{(8)}$	5.47 $^{(8)}$	4.18 $^{(8)}$	7.09 $^{(8)}$
	32	179.96 $^{(8)}$	46.12 $^{(8)}$	5.55 $^{(8)}$	4.34 $^{(8)}$	7.52 $^{(8)}$
	8	125.55 $^{(8)}$	35.52 $^{(8)}$	3.23 $^{(8)}$	2.26 $^{(8)}$	3.96 $^{(8)}$
	12	134.48 $^{(8)}$	35.41 $^{(8)}$	3.19 $^{(8)}$	2.13 $^{(8)}$	3.84 $^{(8)}$
	16	136.69 $^{(8)}$	39.27 $^{(8)}$	3.34 $^{(8)}$	2.31 $^{(8)}$	4.07 $^{(8)}$
32	20	140.14 $^{(8)}$	40.58 $^{(8)}$	3.51 $^{(8)}$	2.33 $^{(8)}$	3.99 $^{(8)}$
	24	138.36 $^{(8)}$	39.70 $^{(8)}$	3.52 $^{(8)}$	2.39 $^{(8)}$	4.15 $^{(8)}$
	28	139.04 $^{(8)}$	37.90 $^{(8)}$	3.44 $^{(8)}$	2.35 $^{(8)}$	4.05 $^{(8)}$
	32	136.39 $^{(8)}$	39.09 $^{(8)}$	3.44 $^{(8)}$	2.33 $^{(8)}$	4.06 $^{(8)}$

Table 2. Average number of verifications performed every 2^{10} characters, computed on a Periodic-ρ small integer sequence, with $\rho = 8, 16$, and 32.

Results on Periodic-ρ sequences show how the number of verifications performed by the algorithms is affected by the value of ρ. Specifically the number of verifications

δ	m	Fct	SkSop$(1,q)$	SkSop$(2,q)$	SkSop$(3,q)$	SkSop$(4,q)$	SkSop$(5,q)$
	8	42.32	0.81 $^{(5)}$	1.14 $^{(4)}$	1.22 $^{(4)}$	1.27 $^{(4)}$	$\underline{1.27}$ $^{(4)}$
	12	27.09	0.80 $^{(7)}$	1.21 $^{(5)}$	1.35 $^{(5)}$	$\underline{1.37}$ $^{(5)}$	1.34 $^{(5)}$
	16	20.38	0.83 $^{(8)}$	1.33 $^{(6)}$	1.44 $^{(5)}$	$\underline{1.52}$ $^{(5)}$	1.50 $^{(5)}$
5	20	16.59	0.88 $^{(8)}$	1.39 $^{(7)}$	1.54 $^{(6)}$	$\underline{1.58}$ $^{(5)}$	1.56 $^{(6)}$
	24	13.56	0.89 $^{(8)}$	1.44 $^{(7)}$	1.60 $^{(6)}$	1.61 $^{(6)}$	$\underline{1.63}$ $^{(6)}$
	28	11.50	0.85 $^{(8)}$	1.47 $^{(7)}$	1.56 $^{(7)}$	1.59 $^{(5)}$	$\underline{1.62}$ $^{(6)}$
	32	9.97	0.81 $^{(8)}$	1.47 $^{(7)}$	1.57 $^{(7)}$	1.59 $^{(6)}$	$\underline{1.60}$ $^{(6)}$
	8	42.13	0.81 $^{(4)}$	1.12 $^{(4)}$	$\underline{1.22}$ $^{(4)}$	1.19 $^{(4)}$	1.14 $^{(4)}$
	12	27.41	0.84 $^{(6)}$	1.24 $^{(5)}$	1.40 $^{(5)}$	$\underline{1.40}$ $^{(5)}$	1.35 $^{(5)}$
	16	19.78	0.85 $^{(7)}$	1.28 $^{(6)}$	1.43 $^{(6)}$	$\underline{1.46}$ $^{(5)}$	1.40 $^{(5)}$
20	20	15.73	0.90 $^{(8)}$	1.33 $^{(7)}$	1.49 $^{(6)}$	$\underline{1.51}$ $^{(5)}$	1.50 $^{(6)}$
	24	13.24	0.89 $^{(8)}$	1.40 $^{(7)}$	1.51 $^{(6)}$	1.55 $^{(6)}$	$\underline{1.55}$ $^{(6)}$
	28	11.37	0.86 $^{(8)}$	1.45 $^{(7)}$	1.57 $^{(6)}$	1.57 $^{(6)}$	$\underline{1.58}$ $^{(6)}$
	32	9.89	0.85 $^{(8)}$	1.42 $^{(7)}$	$\underline{1.58}$ $^{(7)}$	1.56 $^{(6)}$	1.54 $^{(7)}$
	8	41.32	0.81 $^{(4)}$	1.11 $^{(4)}$	$\underline{1.19}$ $^{(4)}$	1.16 $^{(4)}$	1.11 $^{(4)}$
	12	27.36	0.83 $^{(6)}$	1.22 $^{(5)}$	1.38 $^{(5)}$	$\underline{1.39}$ $^{(5)}$	1.34 $^{(5)}$
	16	19.78	0.84 $^{(7)}$	1.27 $^{(6)}$	1.42 $^{(6)}$	$\underline{1.43}$ $^{(5)}$	1.40 $^{(6)}$
40	20	16.21	0.90 $^{(8)}$	1.34 $^{(7)}$	1.51 $^{(6)}$	1.52 $^{(6)}$	$\underline{1.52}$ $^{(6)}$
	24	13.26	0.90 $^{(8)}$	1.40 $^{(7)}$	1.51 $^{(7)}$	1.54 $^{(6)}$	$\underline{1.57}$ $^{(6)}$
	28	11.38	0.86 $^{(8)}$	1.43 $^{(7)}$	1.56 $^{(7)}$	1.56 $^{(6)}$	$\underline{1.57}$ $^{(6)}$
	32	9.93	0.84 $^{(8)}$	1.43 $^{(8)}$	1.56 $^{(7)}$	1.58 $^{(6)}$	$\underline{1.58}$ $^{(7)}$

Table 3. Running times on a Rand-δ small integer sequence, with $\delta = 5, 20$ and 40. Running times (in milliseconds) are reported for the Fct algorithm, while speed-up values are reported for the SkSop(k,q) algorithms.

increases when the period of the function decreases. This is due to the high presence of similar patterns in the text. In addition, in this case, the best results are obtained by the SkSop(k,q) algorithm. Good results are obtained also by combining 3 or 5 fingerprint values.

We observe also that in all cases the number of verifications for each text character is less than 0.1 and is not affected by the length of the pattern. Thus we can observe that the SkSop(k,q) algorithm has a linear behavior on average, as will become apparent in the following evaluation of the running times.

Running Times

The performance of the algorithms presented has been evaluated in terms of their running times. We compared the time required by each algorithm while searching the text for the set of 100 patterns. Table 3 and Table 4 show, respectively, the experimental results obtained on Rand-δ sequences, with $\delta = 5, 20, 40$, and on Periodic-ρ sequences, with $\rho = 8, 16, 32$. Running times are expressed in milliseconds. Best results have been underlined.

In particular, for the Fct algorithm we reported the average running times obtained as the mean of 100 runs. Instead, in the case of the execution of SkSop(k,q) algorithms, we reported the speed up of the running times obtained when compared with the time taken by the fct algorithm. Specifically, if *time*(fct) is the running time of the fct algorithm and t is the running time of our algorithm, then the speed up is computed as *time*(fct)$/t$.

Experimental results on Rand-δ sequences (Table 3) show that the performances of all algorithms are not affected by the value of δ. The Fct algorithm is dominated

ρ	m	Fct	SkSop$(1,q)$	SkSop$(2,q)$	SkSop$(3,q)$	SkSop$(4,q)$	SkSop$(5,q)$
8	8	39.90	0.79 $^{(3)}$	0.87 $^{(4)}$	0.89 $^{(4)}$	0.87 $^{(4)}$	0.87 $^{(4)}$
	12	32.94	0.87 $^{(5)}$	1.09 $^{(6)}$	1.19 $^{(6)}$	1.24 $^{(6)}$	1.17 $^{(6)}$
	16	27.21	0.90 $^{(7)}$	1.24 $^{(7)}$	1.44 $^{(7)}$	1.55 $^{(7)}$	1.46 $^{(7)}$
	20	21.47	0.90 $^{(8)}$	1.21 $^{(7)}$	1.44 $^{(7)}$	1.60 $^{(7)}$	1.50 $^{(7)}$
	24	19.29	0.94 $^{(8)}$	1.27 $^{(7)}$	1.59 $^{(8)}$	1.77 $^{(7)}$	1.68 $^{(8)}$
	28	16.90	0.91 $^{(8)}$	1.28 $^{(7)}$	1.65 $^{(8)}$	1.81 $^{(7)}$	1.77 $^{(8)}$
	32	15.58	0.89 $^{(8)}$	1.28 $^{(7)}$	1.68 $^{(8)}$	1.87 $^{(8)}$	1.83 $^{(8)}$
16	8	37.40	0.68 $^{(4)}$	0.83 $^{(4)}$	0.93 $^{(4)}$	0.93 $^{(4)}$	0.90 $^{(4)}$
	12	25.03	0.60 $^{(5)}$	0.79 $^{(4)}$	1.03 $^{(5)}$	1.04 $^{(5)}$	0.99 $^{(5)}$
	16	18.63	0.60 $^{(6)}$	0.80 $^{(7)}$	1.15 $^{(6)}$	1.15 $^{(6)}$	1.10 $^{(6)}$
	20	15.22	0.53 $^{(8)}$	0.81 $^{(8)}$	1.22 $^{(7)}$	1.19 $^{(7)}$	1.15 $^{(7)}$
	24	12.75	0.50 $^{(7)}$	0.78 $^{(8)}$	1.26 $^{(7)}$	1.24 $^{(7)}$	1.19 $^{(7)}$
	28	10.52	0.45 $^{(7)}$	0.73 $^{(8)}$	1.21 $^{(7)}$	1.20 $^{(7)}$	1.14 $^{(7)}$
	32	10.01	0.43 $^{(8)}$	0.78 $^{(8)}$	1.31 $^{(8)}$	1.29 $^{(7)}$	1.23 $^{(8)}$
32	8	38.82	0.76 $^{(4)}$	1.00 $^{(4)}$	1.11 $^{(4)}$	1.09 $^{(4)}$	1.05 $^{(4)}$
	12	24.86	0.65 $^{(6)}$	0.91 $^{(4)}$	1.16 $^{(5)}$	1.18 $^{(5)}$	1.15 $^{(5)}$
	16	18.85	0.61 $^{(6)}$	0.89 $^{(5)}$	1.24 $^{(6)}$	1.27 $^{(5)}$	1.24 $^{(6)}$
	20	15.02	0.58 $^{(6)}$	0.86 $^{(6)}$	1.31 $^{(6)}$	1.34 $^{(6)}$	1.30 $^{(6)}$
	24	12.32	0.52 $^{(7)}$	0.83 $^{(7)}$	1.31 $^{(7)}$	1.32 $^{(6)}$	1.28 $^{(6)}$
	28	10.89	0.50 $^{(8)}$	0.85 $^{(8)}$	1.38 $^{(7)}$	1.38 $^{(6)}$	1.34 $^{(6)}$
	32	9.50	0.48 $^{(8)}$	0.81 $^{(8)}$	1.37 $^{(7)}$	1.38 $^{(6)}$	1.34 $^{(7)}$

Table 4. Running times on a Periodic-δ integer sequence, with $\delta = 5, 20$, and 40. Running times (in milliseconds) are reported for the Fct algorithm, while speed-up values are reported for the SkSop(k,q) algorithms.

by our Skip Search-based algorithms in all cases, especially for long patterns. When the length of the pattern is between 8 and 20, the SkSop$(4,q)$ algorithm obtains the best results, whereas the SkSop$(5,q)$ algorithm is the best solution when the length of the pattern is greater or equal to 20. In this latter case, the SkSop$(5,q)$ algorithm obtains a significant speed up of 1.60 if compared with the Fct algorithm.

Experimental results on Periodic-ρ sequences (Table 4) show that the algorithm SkSop$(4,q)$ obtains the best results in most of the cases, with the best speed up (up to 1.9) most of the times, especially for long patterns.

The Fct algorithm is still the fastest solution only when the pattern is very short ($m = 8$), while the SkSop$(3,q)$ algorithm obtains very good results when the length of the pattern is between 8 and 20.

5 Conclusions

In this paper we discussed the Order-Preserving Pattern Matching Problem and presented a new algorithm to solve such problem, based on the well-known Skip Search approach. It turns out that our solution is much more effective in practice than existing algorithms. Our algorithm uses SIMD SSE instructions to speed up the searching process. Experimental results show that our solution is up to twice as faster than previous solutions, while exhibiting a linear behavior on average.

References

1. R. Baeza-Yates and G. H. Gonnet: *A new approach to text searching*. Comm. of the ACM, 35(10), 1992, pp. 74–82.
2. R. S. Boyer and J. S. Moore: *A fast string searching algorithm*. Communications of the ACM 20(10), 1977, pp. 762–772.
3. C. Charras, T. Lecroq, and J. D. Pehoushek: *A very fast string matching algorithm for small alphabets and long patterns*, in Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching, M. Farach-Colton, ed., Piscataway, New Jersey, vol. 1448 of Lecture Notes in Computer Science, Springer-Verlag, Berlin 1998, pp. 55–64.
4. T. Chhabra and J. Tarhio: *Order-preserving matching with filtration*, in Proc. SEA '14, 13th International Symposium on Experimental Algorithms, vol. 8504 of Lecture Notes in Computer Science, Springer 2014, pp. 307–314.
5. S. Cho, J. C. Na, K. Park, and J. S. Sim: *Fast order-preserving pattern matching*, in Widmayer, P., Xu, Y., Zhu, B., eds., COCOA 2013, vol. 8287 of Lecture Notes in Computer Science, Springer, Chengdu 2013, pp. 295–305.
6. M. Crochemore, C. S. Iliopoulos, T. Kociumaka, M. Kubica, A. Langiu, S. P. Pissis, J. Radoszewski, W. Rytter, and T. Walen: *Order-preserving incomplete suffix trees and order-preserving indexes*, in Proc. SPIRE 2013, 20th International Symposium, vol. 8214 of Lecture Notes in Computer Science, Springer, Jerusalem 2013, pp. 84–95.
7. B. Durian, J. Holub, H. Peltola, and J. Tarhio: *Improving practical exact string matching*. Information Processing Letters 110(4), 2010, pp. 148–152.
8. S. Faro and M. O. Külekci: *Fast Packed String Matching for Short Patterns*, in Proc. of the 15th Meeting on Algorithm Engineering and Experiments, 2013, pp. 113–121.
9. S. Faro and M. O. Külekci: *Fast and flexible packed string matching*. J. Discrete Algorithms vol. 28, 2014, pp. 61–72.
10. Intel Corporation: *Intel (R) 64 and IA-32 Architectures Optimization Reference Manual*, 2011.
11. J. Kim, P. Eades, R. Fleischer, S.-H. Hong, C. S. Iliopoulos, K. Park, S. J. Puglisi, and T. Tokuyama: *Order preserving matching*. Theoretical Computer Science 525, 2014, pp. 68–79.
12. M. Kubica, T. Kulczynski, J. Radoszewski, W. Rytter, and T. Walen: *A linear time algorithm for consecutive permutation pattern matching*. Information Processing Letters 113(12), 2013, pp. 430–433.
13. D. E. Knuth, J. M. Morris, and V. R. Pratt: *Fast pattern matching in strings*. SIAM Journal on Computing 6(2), 1977, pp. 323–350.
14. G. Navarro and M. Raffinot: *Flexible pattern matching in strings: Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, New York, NY 2002.