Alternative Algorithms for Order-Preserving Matching

Tamanna Chhabra[1], M. Oğuzhan Külekci[2], and Jorma Tarhio[1]

[1] Department of Computer Science, Aalto University
P.O. Box 15400, FI-00076 Aalto, Finland
firstname.lastname@aalto.fi

[2] ERLAB Software Co.
ITU Ari2 Teknokent, Istanbul, Turkey
okulekci@medipol.edu.tr

Abstract. The problem of order-preserving matching is to find all substrings in the text which have the same relative order and length as the pattern. Several online and one offline solution were earlier proposed for the problem. In this paper, we introduce three new solutions based on filtration. The two online solutions rest on the SIMD (Single Instruction Multiple Data) architecture and the offline solution is based on the FM-index scheme. The online solutions are implemented using two different SIMD instruction sets, SSE (streaming SIMD extensions) and AVX (Advanced Vector Extensions). Our main emphasis is on the practical efficiency of algorithms. Therefore, we show with practical experiments that our new solutions are faster than the previous solutions.

Keywords: order-preserving matching, string searching, FM-index, SIMD, SSE, AVX

1 Introduction

The string matching problem [26] of finding all occurrences of a pattern string P of length m in a text string T of length n is one of the classical problems in computer science. Over the last few decades, there has been active development in the field of string matching. One string problem is to locate all the substrings in the text T which have the same relative order and length as the pattern P. This problem is known as *order-preserving matching* [1,4,7,8,21,23]. It has applications in time series studies [20] such as the analysis of development of share prices in a stock market.

In classical string matching, the text T and the pattern P are strings of characters. In order-preserving matching, T and P are strings of numbers. The term relative order means the numerical order of the numbers in the string. In $P = (35, 42, 29, 24, 32, 40)$, number 24 is the smallest number of the pattern, 29 is the second smallest and so on. Therefore, the relative order of P is $4, 6, 2, 1, 3, 5$. The aim of order-preserving matching is to find all the substrings in T which have the same length and relative order as P. It can be observed that the relative order of the substring at location 4 of text $T = (10, 18, 22, 30, 39, 15, 12, 20, 35, 24, 32)$, matches that of the pattern P as shown in Fig. 1, where indexing of T starts from zero.

Several online [1,4,8,21,23] and one offline solution [6] have been proposed for order-preserving matching. Kubica et al. [23] proposed the first online solution based on the Knuth–Morris–Pratt algorithm (KMP) [22]. The second solution was put forward by Kim et al. [21] which also rested on the KMP algorithm. Both the solutions were linear. Later, Cho et al. [4] introduced a solution based on the Boyer–Moore–Horspool (BMH) algorithm [17] and it was the first practical sublinear average-case

Tamanna Chhabra, M. Oğuzhan Külekci, Jorma Tarhio: Alternative Algorithms for Order-Preserving Matching, pp. 36–46.
Proceedings of PSC 2015, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05787-2

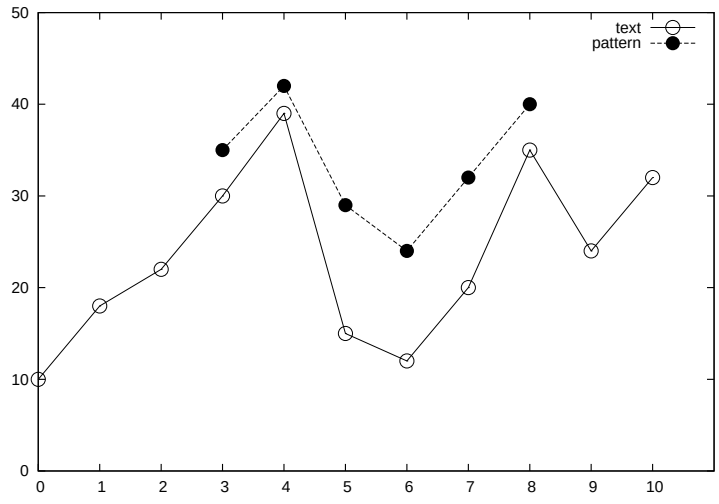

Figure 1. Example of order preserving matching.

solution of the problem. At the same time, Belazzougui et al. [1] derived an optimal algorithm which is sublinear on average. Independently, Chhabra and Tarhio [8] presented another sublinear average-case solution based on filtration and it was shown to be faster in practice than the previous solutions. In the upcoming sections of the paper, we will refer to this solution as OPMF, short for order-preserving matching with filtration. In addition, Crochemore et al. [6] proposed an offline solution based on indexing.

In this paper, we will introduce two new online solutions utilizing the *SIMD (single instruction, multiple data)* architecture [19] and one offline solution based on the FM-index [14]. The online solutions use specialized packed string instructions with a low latency and throughput and turned out to be clearly faster than the previous online solutions. The OPMF algorithm is based on computing a transformed pattern and text by creating their respective bitmaps where a 1 bit means the successive element is greater than the current one and a 0 bit means the opposite. In the online solutions, we aim to perform this transformation quickly with SSE4.2 (streaming SIMD extensions) and AVX (Advanced Vector Extensions) instructions. In the offline solution, the computed bitmap of the text is stored in the compressed form via the FM-index scheme. The transformed pattern is then searched in the FM-index to get potential matches which are then verified. Our main emphasis is on the practical efficiency of algorithms. Therefore, we compared our new solutions with OPMF which was proven to be the fastest practical solution so far. Our experiments show that at least one of our new online solutions is in most cases faster than the original OPMF. And the indexing solution was the most efficient as one may expect.

The paper is organized as follows. Section 2 presents the background, Section 3 describes the problem definition, Section 4 outlines the previous solutions for the order-preserving matching, Section 5 introduces our solution based on filtration, Section 6 interprets the analysis of the solution, Section 7 presents the results of practical experiments, and Section 8 concludes the article.

2 Background

Our new online solutions apply filtration and the SIMD instruction set architecture [19]. These instructions were originally developed for multimedia but are recently employed for pattern matching. The general trend in the last decades for speeding up string matching algorithms has been based on the word-RAM model, where in practice several operations on items occupying a single *word* are assumed to be achieved in constant time. In that context, the advance of the SIMD technology gave rise to packed string matching [2], where one can assume several consecutive symbols of the underlying text are packed into a single register, and there exist special instructions on those special registers to operate on those items individually. The SIMD instructions were used to create a filter while searching for single long patterns in [24]. The filtration code was listed among the best performing 11 pattern matching algorithms in a recent survey [13]. The same idea was deployed for multiple string matching [11], and then extended to also cover short patterns [12,10]. Ladra et al. [16] investigated the benefits of using SIMD instructions on compressed data structures, mainly on rank/select operations, and analyzed the BMH algorithm [17] as a case study. Our results in this paper show that SIMD instructions can also be very efficient in order-preserving pattern matching as well.

The SIMD architecture [19] allows the execution of multiple data on single instruction. SSE (streaming SIMD extensions) [19,29] is a family of SIMD instruction sets supported by modern processors. Intel added sixteen new 128-bit registers known as XMM0 through XMM15. However, the registers XMM7–XMM15 are only accessible in the 64-bit operating mode. As the registers are 128 bits long, four floating point numbers could be handled at the same time (a single precision floating point number is considered 32 bits long), thereby providing important speedups in algorithms. Furthermore, the functionality provided by SSE instructions was extended by Intel AVX (Advanced Vector Extensions) [29]. It provides support for 256-bit registers known as YMM0 through YMM15. As with SSE, in AVX also the registers YMM7–YMM15 are only accessible in the 64-bit operating mode. As the registers have been extended from 128 bits to 256 bits, hence eight floating point numbers could be managed simultaneously, thereby rendering substantial performance gain. Dedicated data types are utilized in SIMD programming. In SSE4.2, we have the following data types:

- _m128: four 32-bit floating point values
- _m128d: two 64-bit floating point values
- _m128i: 16/8/4/2 integer values, depending on the size of the integers

The similar data types are available in AVX but the length of registers is 256. These vector data types are defined in separate header files depending on the type of instruction set architecture. To perform a task, we have different intrinsic functions. The name of the function starts with _mm. After that follows the name describing the operation. The next character specifies whether the operation is on a packed vector or on a scalar value: p stands for packed and s for scalar operation. The last character relates to single precision or double precision floating point values. For example, _mm_cmpgt_ps is a function for comparing two values.

The offline solution rests on the FM-index scheme [14]. Ferragina and Manzini [14] proposed that if the Burrows-Wheeler transform [3] is coupled with another data structure, namely Suffix Arrays (SA) [25], we get a space efficient index which is a sort of compressed suffix array called the FM-index. It can be used to count efficiently

the occurrences of a pattern in the compressed text and to determine the locations of each pattern in the text.

3 Problem definition

Problem definition. Two strings u and v of the same length over Σ are called *order-isomorphic* [23], written $u \approx v$, if

$$u_i \leq u_j \Leftrightarrow v_i \leq v_j \text{ for } 1 \leq i, j \leq |u|.$$

In the *order-preserving pattern matching problem*, we want to locate all the substrings in the text T which are order-isomorphic with the pattern P.

In the example of Section 1, the substring $w = (30, 39, 15, 12, 20, 35)$ of T is order-isomorphic with pattern $P = (35, 42, 29, 24, 32, 40)$ as can also be seen in Fig. 1.

4 Previous solutions

This section describes the previous solutions formulated for the order-preserving matching problem. First of all we explain the online solutions given so far. The first solution was presented by Kubica et al. [23] based on Knuth–Morris–Pratt algorithm (KMP) [22]. In this approach the fail function in the KMP algorithm was modified to compute the order-borders table. With this table one can find out in linear time if the text contains substring with the same relative order as that of the pattern.

The second solution to the problem was given by Kim et al. [21] and was grounded on the prefix representation. It is based on finding the rank of each number in the prefix. They used the dynamic order statistic tree as the data structure and the text is searched using the KMP-order-matcher function. The total time complexity is $O(n \log m)$. This approach was then enhanced using the nearest neighbor representation. Thereafter, the total time complexity is $O(n + m \log m)$.

However, Cho et al. [4] provided a solution based on the BMH approach [17]. They applied the variant of BMH algorithm built on q-gram. This was the first practical sublinear solution of the problem. The time complexity in the worst case is $O(mn)$. Later on, they also developed a version which is linear in the worst case [5], but that is in practice a bit slower than the original one.

The algorithm by Belazzougui et al. [1] is optimal sublinear. They viewed the problem in a slightly different way: T is a permutation of $1, \ldots, n$ and P consists of m distinct integers of $[1, n]$. They constructed a forward search automaton working in $O(m^2 \log\log m + n)$ time which is too large for long patterns. With a Morris-Pratt representation of the forward automaton, they accieved $O(m \log\log m + n)$ search time. Furthermore, the automaton was extended to accept a set of patterns. Besides these linear solutions, they presented a sublinear average case algorithm. Firstly, a tree is constructed of all isomorphic order factors of P by inserting factors one at a time. Thereafter search is performed along the text through a window of size m. The construction time of the tree is $O(\frac{m \log m}{\log\log m})$ and average-case time complexity is $O(\frac{n \log m}{m \log\log m})$. However, there exists no implementation of this algorithm so far.

Another sublinear average-case solution is OPMF [8]. The solution consisted of two phases: filtration and verification. In filtration, the pattern P and the text T are transformed to P' and T' by creating their respective bitmaps such that a 1 bit

means the successive element is greater than the current one, and a 0 bit means otherwise. The text is transformed incrementally online in order to be able to skip characters. Any (sublinear) exact string matching is then applied to filter out the text. As a result, we get match candidates, which are then verified using a checking function. In addition to exact order-preserving matching, the same filtration method can also be applied to approximate order-preserving matching [7] and to multiple order-preserving matching [28].

Lastly, let us consider the offline solution by Crochemore et al. [6]. This approach is grounded on the construction of an index that handles the queries in linear time with respect to the length of the pattern. The index is based on the incomplete suffix tree and its construction takes $O(n \log \log n)$ time. They extended their work to complete order-preserving suffix trees and showed how these can be constructed in $O(n \log n / \log \log n)$ time. There exists no practical implementation of this algorithm.

5 Our solutions

We propose two online and one offline solutions for order preserving matching. The online solutions utilize the SIMD architecture [19]. The first online solution employs the SSE4.2 instruction set architecture and the second solution utilizes the AVX instruction set architecture.

Online solution using SSE4.2

In the OPMF algorithm, the pattern P is transformed into P' and the text T is transformed incrementally to T'. We aim to perform the online transformation faster than in OPMF. This solution for order-preserving matching consists of two parts: filtration and verification. First the text is filtered and then the match candidates are verified using a checking routine.

Filtration using SSE4.2. Assume that we have 32 bits long floating point numbers and the processor has SSE4.2 support. The preprocessing of the pattern consists of two parts. First a bit mask, which is the reverse of P', is formed and after that a shift table is constructed based on the mask. For the bit mask, the consecutive numbers in the pattern $P = p_0 p_1 \cdots p_{m-1}$ are compared pairwise, $(p_0 > p_1)(p_1 > p_2)(p_2 > p_3) \cdots (p_{m-2} > p_{m-1})$. This can be achieved by creating $\_mm128$ type pointers $ptr1$ and $ptr2$ pointing to p_0 and p_1 respectively. Furthermore, we use the PCMPGT instruction (_mm_cmpgt_ps()) to compare $ptr1$ with $ptr2$ to compute $(p_0 > p_1)(p_1 > p_2)(p_2 > p_3)(p_3 > p_4)$ in parallel. It compares the packed single-precision floating-point values in the source operand (the second operand) and the destination operand (the first operand) and returns the results of the comparison to the destination operand. The result of this instruction is 128 bits long. Additionally, we use the MOVMSK instruction (_mm128_movemask_ps()) which extracts the most significant bits from the packed single-precision floating-point value in the source operand. The reverse of the result is stored in the four low-order bits of the destination operand. The upper bits of the destination operand are filled with zeros. The result will be the bit mask *mask*. Alg. 1 shows how the transformation of the pattern P into *mask* can be carried out rapidly.

Since SSE4.2 allows four numbers to be compared in parallel, we apply binary 4-grams and set the size of the shift table *delta* to 16 ($=2^4$). The construction algorithm

```
PREPROCESSING(mask)
for (i = 0; i < 16; i++)
   delta[i]=m-1
   k = (mask<<3) & 0xf;
for (i = 0; i < 8 ; i++)
   delta[k+i]=m-2
   k = (mask<<2) & 0xf;
for (i = 0; i < 4 ; i++)
   delta[k+i]=m-3
   k = (mask<<1) & 0xf;
for (i = 0; i < 2 ; i++)
   delta[k+i]=m-4
for (i = 0; i < m-4 ; i++)
   delta[(mask>>i) & 0xf] = m-i-5
```

```
SEARCH(Text, delta)
i=m-5;
while i<n do
  k=1
  while k>0 do
    k = delta[simd-comp(t_i,t_{i+1},4)]
    i = i+k
  for (j=i-m+5; j<i; j+=4)
    z = simd-comp(t_j,t_{j+1},4)
    if (z != ((mask>>(j-i+m-5)) & 0xf))
     then goto out
  verify occurrence
  out: i = i+1
```

Figure 2. The PREPROCESSING and SEARCH phases.

for *delta* is shown in the left-hand side of Fig. 2. The computation of the parameter *mask* is explained above. The entry $delta[x]$ is zero if x is the reverse of the last 4-gram of P'. The entries of the table are initialized to $m-1$. Thereafter, the entries are updated according to the preprocessing algorithm of Fig. 2. Fig. 3 shows how the shift table is formed for the pattern P of Fig. 1. At the end, entry 12 is zero. This means that $12 = 1100$ is the reverse of the last 4-gram of P'.

Algorithm 1 (Transformation of pattern into bitmap)

```
mask = 0
for (i = 0; i < (m-1); i=i+4)
   x_ptr = _mm_loadu_ps(pattern+i+1)
   y_ptr = _mm_loadu_ps(pattern+i)
   mask = mask | _mm_movemask_ps(_mm_cmpgt_ps(x_ptr, y_ptr)) << i
```

The search algorithm shown in the right-hand side of Fig. 2 is a variation of the BMH algorithm [17,27] utilizing 4-grams. Inside the main loop there are two loops. The first loop searches for occurrences of the last 4-gram of P' by using the shift table *delta*. The tested 4-gram is formed online with SIMD instructions in the same way as used for the pattern. The numbers are compared in parallel using PCMPGT instruction explained above (simd-comp in Fig. 2). The second loop checks whether a complete occurrence of P' is found. If an occurrence of P' is found, the corresponding part of T is verified. The search algorithm uses a copy of the pattern as a sentinel (not shown in Fig. 2) to be able to recognize the end of input.

As each occurrence of P' in T' is only a match candidate, it should be verified. In simple words, for instance, if $P = (15, 18, 20, 16)$ and $T = (2, 4, 6, 1, 5, 3)$ then the transformed pattern P' and T' are 110 and 110101 respectively where 1 indicates increase and 0 indicates the opposite. The match candidate of P' at location 0 of T' needs to be verified because though P' appears in T', the relative order of the numbers is 0,2,3,1 in the pattern and 1,2,3,0 in the text. Therefore P' is only a match candidate.

Verification. The verification process is the same as in OPMF. In the preprocessing phase, the numbers of the pattern $P = p_0 p_1 \cdots p_{m-1}$ are sorted. As a result, we obtain an auxiliary table r: $p_{r[i]} \leq p_{r[j]}$ holds for each pair $i < j$ and $p_{r[0]}$ is the smallest number in P. The potential candidates obtained from the filtration phase are traversed in accordance with the table r. If the candidate starts from t_j in T, the first comparison is done between $t_{j+r[0]}$ and $t_{j+r[1]}$.

$delta[0000]$	$\leftarrow 5$				
$delta[0001]$	$\leftarrow 5$				
$delta[0010]$	$\leftarrow 5$			$\leftarrow 2$	
$delta[0011]$	$\leftarrow 5$			$\leftarrow 2$	
$delta[0100]$	$\leftarrow 5$		$\leftarrow 3$		
$delta[0101]$	$\leftarrow 5$		$\leftarrow 3$		
$delta[0110]$	$\leftarrow 5$		$\leftarrow 3$		
$delta[0111]$	$\leftarrow 5$		$\leftarrow 3$		
$delta[1000]$	$\leftarrow 5$	$\leftarrow 4$			
$delta[1001]$	$\leftarrow 5$	$\leftarrow 4$			$\leftarrow 1$
$delta[1010]$	$\leftarrow 5$	$\leftarrow 4$			
$delta[1011]$	$\leftarrow 5$	$\leftarrow 4$			
$delta[1100]$	$\leftarrow 5$	$\leftarrow 4$			$\leftarrow 0$
$delta[1101]$	$\leftarrow 5$	$\leftarrow 4$			
$delta[1110]$	$\leftarrow 5$	$\leftarrow 4$			
$delta[1111]$	$\leftarrow 5$	$\leftarrow 4$			

Figure 3. Computation of the shift table for $mask = 11001$ for $P' = 10011$.

Online solution using AVX

This is similar to the above solution with a few exceptions. The difference is in the comparison of numbers and in computation of the shift function. Instead of four numbers, eight floating point numbers are compared at a stretch. The comparison instruction is _mm256_cmp_ps(), which requires three operands. The predicate operand (the third operand) specifies the type of comparison to be performed on each of the pairs of packed values.

Offline Solution

This solution also enumerates the bitmaps but they are stored in the compressed form via the FM-index. In this case, when a pattern is queried, we just extract the possible candidate positions from the index, and then apply naive check. It also consists of two parts: filtration and verification.

Filtration. In the preprocessing phase, the consecutive numbers in the pattern $P = p_0 p_1 \cdots p_{m-1}$ are compared pairwise and the pattern P is transformed into a bitmap P' in the same way as in OPMF. The text is also encoded and an FM-index is created of the encoded text. Alg. 2 below shows how the encoded text is stored in the form of FM-index. Thereafter, the occurrences of transformed pattern P' are found within the compressed text. As an occurrence of P' is only a potential match candidate, it should be verified with a checking routine.

Note. It was thought that there might be an inefficiency in the FM-index for a bit string. It is because the FM-index uses a wavelet tree, and it would be useless in the case of a binary text. So a modified FM-index without a wavelet tree might be more efficient. Therefore we implemented another FM-index without a wavelet tree. To keep the FM-index compressed, the Burrows-Wheeler transform of the bit-string was computed and was compressed via rank and select dictionaries, and then the backward search on the compressed bit string was implemented via rank/select queries. But we observed that this approach was slower than the standard one.

Verification. The verification process is the same as in the online solution because once we get the potential matches they are verified using the same checking function.

Algorithm 2 (FM-index)

```
std::string str((char *) & text[0], n);
construct_im(fm_index, str.c_str(), 1);
matches=count(fm_index, (const char*)P');
auto locations=locate(fm_index, (const char*)P');
```

6 Analysis

Let us assume that the numbers in $P = p_0 p_1 \cdots p_{m-1}$ and $T = t_0 t_1 \cdots t_{n-1}$ are integers and they are statistically independent of each other and the distribution of numbers is discrete uniform. Let P' and T' be the corresponding bitmaps for filtration. In case of online solutions using SIMD, the analysis is similar to the analysis of the original OPMF algorithm. It is obvious that our SIMD search algorithms are sublinear on average, because the search algorithm based on BMH is sublinear on average. The verification time approaches zero when m grows, and the filtration time dominates. When filtering the text T, it is encoded incrementally online in order to skip characters, and the solution as a whole becomes sublinear on average. In the worst case, the total algorithm requires $O(nm)$ time, if for example P' is 1^{m-1} and T' is 1^{n-1}. The preprocessing phase requires $O(m \log m)$ due to sorting of the pattern positions. See the analysis of OPMF [8] for more details.

In the case of the offline solution using the FM-index, the verification time also approaches zero when m grows and the filtration time dominates. During the preprocessing phase, the text T' is compressed and stored via the FM-index. The operation *count* takes a pattern P' and returns the number of occurrences of that pattern in the text T'. It can count all matching positions in $O(m)$ time. The operation *locate* finds the locations of all the occurrences (occ) of the pattern P' in T' in time $O(m + occ \log^{\epsilon} n)$. However, in the worst case, this solution also requires $O(nm)$ time because checking a match candidate takes $O(m)$ time.

7 Experiments

The tests were run on Intel 2.70 GHz i7 processor with 16 GB of memory. All the algorithms were implemented in C and run in the testing framework of Hume and Sunday [18]. In case of offline solution, the FM-index was implemented using the sdsl library [15].

We tested our algorithms on ten different data sets. Of all the results, we present the results on three texts: a random text and two real texts. The random data [20] is 320 MB long. The real data comprised of time series of the Dow Jones index and feature data [20]. The Dow Jones data consisted of 15,128 integers and feature data is 198 MB long. The patterns were randomly picked from the text. We had eight sets of 300 patterns in case of random and feature data set with lengths $5, 9, 10, 12, 15, 18, 20$ and 25 and five sets of 300 patterns with lengths $5, 10, 15, 20$ and 25 in case of Dow Jones data. Each test was repeated nine times.

We compared our new solutions with our earlier OPMF solutions [9] based on the SBNDM2 and SBNDM4 algorithms. Because the latter solutions were faster than the other old solutions in the tests of [9], we do not present results for other methods. Tables 1 and 2 show the average execution times of the algorithms for a set of 300 patterns for random and feature data in seconds, respectively, whereas table 3 depicts the average execution times of the algorithms for a set of 300 patterns for Dow Jones

data in 10 of milliseconds. In addition, graphs on times for random data are shown in Fig. 4 respectively. In the tables given below, SBNDM2 represents the OPM algorithm based on SBNDM2 filtration, SBNDM4 represents the OPM algorithm based on SBNDM4 filtration, SSE represents the online solution based on SSE4.2 instruction set, AVX represents the online solution based on AVX instruction set and FM-INDEX represents the offline solution based on the FM index.

m	SBNDM2	SBNDM4	SSE	AVX	FM-INDEX
5	18.44	22.56	12.26	—	405.53
9	15.96	13.34	8.71	9.06	31.53
10	13.99	12.21	7.98	7.37	14.01
12	11.56	10.66	7.07	5.69	3.14
15	9.07	8.80	6.35	4.59	0.39
18	7.52	7.34	5.92	4.10	0.05
20	6.85	6.69	5.82	3.87	0.01
25	5.59	5.40	5.44	3.59	0.00

Table 1. Execution times of algorithms in seconds for random data

m	SBNDM2	SBNDM4	SSE	AVX	FM-INDEX
5	16.92	20.28	12.33	—	306.26
9	9.41	8.02	5.37	5.63	21.29
10	8.26	7.32	4.86	4.58	8.52
12	6.93	6.53	4.29	3.48	3.16
15	5.42	5.27	3.80	2.81	0.32
18	4.52	4.43	3.62	2.52	0.11
20	4.06	4.30	3.45	2.36	0.04
25	3.28	3.29	3.29	2.17	0.02

Table 2. Execution time of algorithms in seconds for feature data

m	SBNDM2	SBNDM4	SSE	AVX	FM-INDEX
5	1.14	1.49	0.83	—	14.17
10	0.49	0.31	0.36	0.45	0.41
15	0.29	0.16	0.34	0.22	0.04
20	0.18	0.14	0.28	0.21	0.03
25	0.12	0.12	0.23	0.10	0.04

Table 3. Execution times of algorithms in 10 of milliseconds for Dow Jones data

From Tables 1, 2, and 3, it can be clearly seen that our solutions based on the FM-index, SSE4.2 and AVX are the fastest depending on the value of m. Irrespective of the data, the solution based on SSE4.2 is the fastest for $m = 5$. In case of random and feature data, as the value of m reaches 10, the AVX solution becomes the fastest. However, when m is greater than or equal to 12, the FM-index based solution is the fastest. And as the value of m reaches 25, the execution time of FM-index based solution approaches zero. However, in the case of Dow Jones data, the FM-index based solution is the fastest as the value of m reaches 10.

The construction times of the FM-index for our Dow Jones and random texts were 0.07 and 3.2 seconds, respectively.

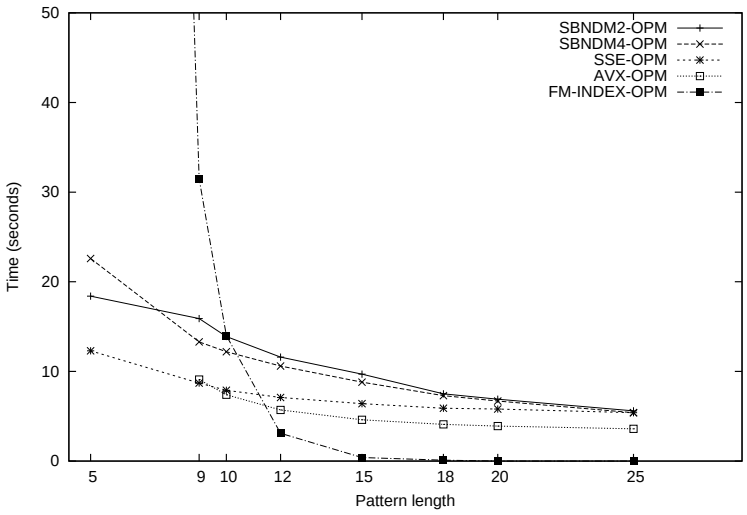

Figure 4. Execution times of algorithms for random data

8 Concluding remarks

We pioneered two online and one offline solution for the order-preserving matching problem. The online solutions are based on SIMD which improves the execution time substantially in most cases. The SIMD architecture requires careful redesigning of an algorithm, and the outcome is not necessarily efficient for an arbitrary string matching problem. The offline solution, which is based on the FM-index, is superior for long patterns. However, the search algorithm of the offline solution was slower than we expected for short patterns. We have proved with practical experiments that our solutions are competitive with the previous solutions.

References

1. D. Belazzougui, A. Pierrot, M. Raffinot, and S. Vialette: *Single and multiple consecutive permutation motif search*, in Proceedings of 24th International Symposium on Algorithms and Computation, ISAAC 2013, Hong Kong, China, December 16–18, 2013, pp. 66–77.
2. O. Ben-Kiki, P. Bille, D. Breslauer, L. Gasieniec, R. Grossi, and O. Weimann: *Optimal packed string matching*, in Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS vol. 13, 2011, pp. 423–432.
3. M. Burrows and D.J. Wheeler: *A block sorting lossless data compression algorithm*. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.
4. S. Cho, J. C. Na, K. Park, and J. S. Sim: *Fast order-preserving pattern matching*, in Widmayer, P., Xu, Y., Zhu, B., eds., 7th International Conference on Combinatorial Optimization and Applications 2013, vol. 8287 of Lecture Notes in Computer Science, 2013, pp. 295–305.
5. S. Cho, J. C. Na, K. Park, and J. S. Sim: *A fast algorithm for order-preserving pattern matching*. Inf. Process. Lett., 115(2) 2015, pp. 397–402.
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 Kurland, O., Lewenstein, M., Porat, E., eds., 20th String Processing and Information Retrieval Symposium 2013, vol. 8214 of Lecture Notes in Computer Science, 2013, pp. 84–95.
7. T. Chhabra, E. Giaquinta, and J. Tarhio: *Filtration algorithms for approximate order-preserving matching*. Submitted, 2015.

8. T. Chhabra and J.Tarhio: *Order-preserving matching with filtration*, in Gudmundsso, J., Katajainen, J., eds., 13th International Symposium on Experimental Algorithms 2014, vol. 8504 of Lecture Notes in Computer Science, 2014, pp. 307–314.
9. B. Ďurian, J. Holub, H. Peltola, and J. Tarhio: *Improving practical exact string matching.* Inf. Process. Lett., 110(4) 2010, pp. 148–152.
10. S. Faro and M. O. Külekci: *Fast and flexible packed string matching.* Journal of Discrete Algorithms, 28 (2014), pp. 61–72.
11. S. Faro and M. O. Külekci: *Fast multiple string matching using streaming SIMD extensions technology*, in L. Calderón-Benavides et al., eds., 19th International Symposium on String Processing and Information Retrieval 2012, vol. 7608 of Lecture Notes in Computer Science, pp. 217–228.
12. S. Faro and M. O. Külekci: *Fast packed string matching for short patterns*, in Proceedings of the 15th Meeting on Algorithm Engineering and Experiments 2013, pp. 113–121.
13. S. Faro and T. Lecroq: *The exact online string matching problem: A review of the most recent results.* ACM Computing Surveys (CSUR), 45(2) 2013, p. 13.
14. P. Ferragina and G. Manzini: *Opportunistic data structures with applications*, in 41st Annual Symposium on Foundations of Computer Science, IEEE 2000, pp. 390–398.
15. S. Gog: *Succinct Data Structure Library 2.0*, `https://github.com/simongog/sdsl-lite`.
16. S. Ladra, O. Pedreira, J. Duato, and N.R. Brisaboa: *Exploiting SIMD instructions in current processors to improve classical string algorithms*, in 16th East European Conference on Advances in Databases and Information Systems 2012, T. Morzy, T. Haerder, and R. Wrembel, eds., vol. 7503 of Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg 2012, pp. 254–267.
17. R. N. Horspool: *Practical fast searching in strings.* Software–Practice and Experience, 10(6) 1980, pp. 501–506.
18. A. Hume and D. Sunday: *Fast string searching.* Software–Practice and Experience, 21(11) 1991, pp. 1221–1248.
19. H. Jeong, S. Kim, W. Lee, and S.-H. Myung: *Performance of SSE and AVX instruction sets.* CoRR abs/1211.0820, 2012.
20. E. Keogh, Q. Zhu, B. Hu, Y. Hao., X. Xi, L. Wei, and C. A. Ratanamahatana: *The UCR Time Series Classification/Clustering Homepage*, `http://www.cs.ucr.edu/~eamonn/UCRsuite.html`
21. J. Kim, P. Eades, R. Fleischer, S.-H. Hong, C.S. Iliopoulos, K. Park, S. J. Puglisi, and T. Tokuyama: *Order preserving matching.* Theor. Comp. Sci., 525 (2014), pp. 68–79.
22. D. E. Knuth, J. H. Morris Jr., and V. R. Pratt: *Fast pattern matching in strings.* SIAM Journal on Computing, 6(2) 1977, pp. 323–350.
23. M. Kubica, T. Kulczynski, J. Radoszewski, W. Rytter, and T. Walen: *A linear time algorithm for consecutive permutation pattern matching.* Inf. Process. Lett., 113(12) 2013, pp. 430–433.
24. M. O. Külekci: *Filter based fast matching of long patterns by using SIMD instructions*, in J. Holub and J. Žďárek, eds., Prague Stringology Conference 2009, pp. 118–128.
25. U. Manber and G. Myers: *Suffix arrays. A new method for on-line string searches*, in 1st Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM 1990, pp. 319–327.
26. 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.
27. J. Tarhio and H. Peltola: *String matching in the DNA alphabet.* Software–Practice and Experience, 27(7) 1997, pp. 851–861.
28. B. Watson: *Personal Communication*, 2015.
29. Intel Corporation: *Intel Architecture Instruction Set Extensions Programming Reference*, `https://software.intel.com/sites/default/files/m/9/2/3/41604`.