Order-Preserving Matching with Filtration*

Tamanna Chhabra and Jorma Tarhio

Department of Computer Science and Engineering
Aalto University
P.O. Box 15400, 00076 Aalto, Finland
{firstname.lastname}@aalto.fi

Abstract. The problem of order-preserving matching has gained attention lately. The text and the pattern consist of numbers. The task is to find all substrings in the text which have the same relative order as the pattern. The problem has applications in analysis of time series like stock market or weather data. Solutions based on the KMP and BMH algorithms have been presented earlier. We present a new sublinear solution based on filtration. Any algorithm for exact string matching can be used as a filtering method. If the filtration algorithm is sublinear, the total method is sublinear on average. We show by practical experiments that the new solution is more efficient than earlier algorithms.

1 Introduction

String matching [11] is a widely known problem in Computer Science. Given a text T of length n and a pattern P of length m, both being strings over a finite alphabet Σ, the task of string matching is to find all the occurrences of P in T.

The problem of order-preserving matching [2,3,8,9] has gained attention lately. It considers strings of numbers. The relative order of the positions in the pattern P is matched with the relative order of a substring u of T, where $|u| = |P|$. Suppose $P = (10, 22, 15, 30, 20, 18, 27)$ and $T = (22, 85, 79, 24, 42, 27, 62, 40, 32, 47, 69, 55, 25)$, then the relative order of P matches the substring $u = (24, 42, 27, 62, 40, 32, 47)$ of T, see Fig. 1. In the pattern P the relative order of the positions is: $1, 5, 2, 7, 4, 3, 6$. This means 10 is the smallest number in the string, 15 is the second smallest, 18 the third smallest and so on. In order-preserving matching, the task is to find a substring in T which has the same relative order as P. In the given example, u has the same relative order as P. So u matches the pattern P.

At least three solutions [2,8,9] have been proposed for on-line order-preserving matching of strings. Kubica et al. [9] and Kim et al. [8] have presented solutions based on the Knuth–Morris–Pratt algorithm (KMP) [10]. Kim et al. doubted the applicability of the Boyer–Moore approach [1] to order-preserving matching. However, Cho et al. [2] demonstrated that the bad character heuristic works. We will present a new practical solution based on approximate string matching.

* Supported by the Academy of Finland (grant 134287).

J. Gudmundsson and J. Katajainen (Eds.): SEA 2014, LNCS 8504, pp. 307–314, 2014.

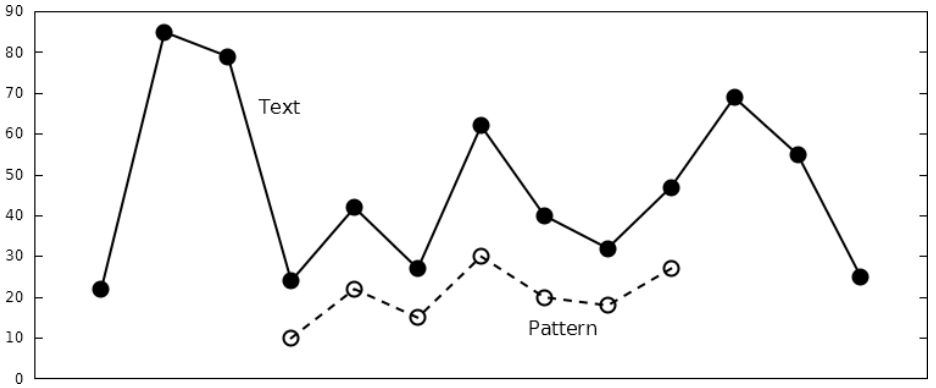

Fig. 1. Example of order-preserving matching

We form a modified pattern and use an algorithm for exact string matching as a filtration method. Our approach is simpler and in practice more efficient than earlier solutions. We transform the original pattern P into a binary string P' expressing increases (1), equalities (0), and decreases (0) between subsequent pattern positions. Then we search for P' in the similarly transformed text T'. For example, $P' = 101001$ corresponds to $P = (10, 22, 15, 30, 20, 18, 27)$ and $T' = 100101001100$ to T above. Each occurrence is a match candidate which is verified following the numerical order of the positions of the original pattern P. Note that in this approach any algorithm for exact string matching can be used as a filtration method. If the filtration algorithm is sublinear and the text is transformed on line, the total method is sublinear on average.

We made experiments with two sublinear string matching algorithms and two linear string matching algorithms as the filtering method. Our approach with sublinear filters was considerably faster than the algorithm by Cho et al. [2], which is the first sublinear solution of the problem and which was the previous winner.

The paper is organized as follows. Section 2 describes the previous solutions for the order-preserving matching, Section 3 presents our solution based on filtration, Section 4 analyses the new approach, Section 5 presents and discusses the results of practical experiments, and Section 6 concludes the article.

2 Previous solutions

There are three solutions presented so far, two algorithms are based on the KMP algorithm [10] and the third one is based on the Boyer–Moore–Horspool algorithm (BMH) [1,6].

In the first KMP approach presented by Kubica et al. [9], the fail function in the KMP algorithm is modified to compute the order-borders table. This can be achieved in linear time. The KMP algorithm is mutated such that it determines

if the text contains substring with the same relative order as that of the pattern using the order-borders table B, which is defined for the pattern P as follows:

$$B[1] = 0$$
$$B[i] = \max\{j < i : P[1\,..\,j] \approx P[i-j+1\,..\,j]\} \text{ for } i \geq 2$$

This computation can also be done in linear time. Hence, the time complexity of the algorithm is linear.

The second KMP approach by Kim et al. [8] is based on the prefix representation and it is further optimized according to the nearest neighbor representation.

The prefix representation is based on finding the rank of each integer in the prefix. It can be computed easily by inserting each character to the dynamic order statistic tree and then computing the rank of each character in the prefix. The functions insert and rank both take $O(\log m)$ time and since there are $O(m)$ operations, the time complexity of computing prefix representation is $O(m \log m)$. The failure function is computed as in the KMP algorithm using the previous values in $O(m \log m)$ time. Finally the text is searched using the function KMP-order-matcher. It is different from the KMP algorithm as it is based on the ranks of the prefix. The functions involved in it are rank, insert and delete, and all of them take $O(\log m)$ time, and since the number of operations can be at most n, these functions take $O(n \log m)$ time. The failure function takes $O(m \log m)$ time and computing the prefix representation takes $O(m \log m)$ time. Thus the total time complexity is $O(n \log m)$.

This approach is optimized to overcome the overhead involved in computing the rank function using the nearest neighbor representation. It checks if the order of each character in the text matches the corresponding character in the pattern. In this, the characters itself are compared and there is no need to compute the rank. It takes $O(m \log m)$ time to compute the nearest neighbor representation of the pattern, $O(m)$ time for the computation of the failure function and $O(n)$ time for searching the text. So the total time complexity is $O(n + m \log m)$.

The BMH approach [2] is based on the bad character rule applied to q-grams, i.e. strings of q characters. A q-gram is treated as a single character to make shifts longer. In this way, a large amount of text can be skipped for long patterns, and the algorithm is obviously sublinear on the average. This algorithm is the first sublinear solution for order-preserving matching.

3 Our Solution

In Section 1 we gave an informal description of order-preserving matching. That definition causes problems in handling equal values which can appear in real data. The concept of order-isomorphism removes these problems.

Problem definition. Two strings u and v of the same length over Σ are called *order-isomorphic* [9], 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*, the task is to find all the substrings of T which are order-isomorphic with P.

Our solution for order-preserving matching consists of two parts: filtration and verification. First the text is filtered with some exact string matching algorithm and then the match candidates are verified using a checking routine.

Filtration. For filtration, the consecutive numbers in the pattern $P = p_1 p_2 ... p_m$ are compared pairwise in the preprocessing phase and the result is encoded as a string $P' = b_1 b_2 ... b_{m-1}$ of binary numbers: b_i is 1 if $p_i < p_{i+1}$ holds, otherwise b_i is 0. In the search phase, some algorithm for exact string matching (let us call it A) is applied to filter out the text. When Algorithm A reads an alignment window of the original text, the text is encoded on line in the same way as the pattern. Algorithm A is run, as if the whole text would have been encoded. Because Algorithm A may recognize an occurrence of P' which does not correspond to an actual match of P in T, each occurrence of P' is only a match candidate which should be verified. It is clear that this filtration method cannot skip any occurrence of P in T.

Verification. In the preprocessing phase, the positions of the pattern $P = p_1 p_2 ... p_m$ are sorted. The result is an auxiliary table r: $p_{r[i]} \leq p_{r[j]}$ holds for each pair $i < j$ and $p_{r[1]}$ is the smallest number in P. In addition, we need a binary vector E representing the equalities: $E[i] = 1$ denotes that $p_{r[i]} = p_{r[i+1]}$ holds. The match candidates found by Algorithm A are traversed in accordance with the table r. If the candidate starts from t_j in T, the first comparison is done between $t_{j-1+r[1]}$ and $t_{j-1+r[2]}$. There is a mismatch when

$$\begin{aligned} & t_{j-1+r[i]} > t_{j-1+r[i+1]} \text{ or} \\ & (t_{j-1+r[i]} = t_{j-1+r[i+1]} \text{ and } E[i] = 0) \text{ or} \\ & (t_{j-1+r[i]} < t_{j-1+r[i+1]} \text{ and } E[i] = 1) \end{aligned}$$

is satisfied. The candidate is discarded when a mismatch is encountered. Verification is efficient because sorting is done only once during preprocessing.

Remark. We use binary characters in encoding. We also tried encoding of three characters 0, 1, and 2 corresponding to '<', '=', and '>', but the binary approach was faster in practice, because testing of one condition is faster than testing of two conditions. Also the frequency of consecutive equalities is low in real data.

4 Analysis

We will prove that our approach is sublinear in the average case, if the filtration algorithm is sublinear. Sublinearity means that on average all the characters in the text are not examined.

Let us assume that the numbers in P and T are integers and they are statistically independent of each other and the distribution of numbers is discrete

uniform. Let P' and T' be the transformed pattern and text. Let c be the count of the integer range (i.e. the alphabet size). The probability of one in a position of P' or T' (as a result of a comparison) is $p = (c^2/2 - c/2)/c^2 = (c-1)/2c$, because there are c^2 integer pairs and c equalities. So the probability of a character match q is

$$p^2 + (1-p)^2 = 2p(p-1) + 1 = 1 - \frac{c-1}{c} \cdot \frac{c+1}{2c} = 1 - \frac{c^2-1}{2c^2} = \frac{1}{2} + \frac{1}{2c^2}.$$

The probability of a match of P' (i.e. a match candidate of P) at a certain position of T' is q^{m-1}, which approaches to zero, when m grows. This is true even for $c = 2$. This means that the verification time approaches zero when m grows, and the filtration time dominates. If the filtration method is sublinear, the total algorithm is sublinear.

In the worst case, the total algorithm requires $O(nm)$ time, if for example P is 1^m and T is 1^n.

The preprocessing phase requires $O(m \log m)$ due to sorting of the pattern positions. The space requirement is $O(m)$.

5 Experiments

We tested four string matching algorithms as filtration methods for order-preserving matching. Two of them, SBNDM2 and SBNDM4 [4] are based on the Backward Nondeterministic DAWG Matching (BNDM) algorithm [11]. In BNDM, each alignment window is processed from right to left like in the Boyer–Moore algorithm [1]. BNDM simulates the nondeterministic automaton of the reversed pattern with bitparallelism. SBNDMq starts the processing of each alignment window by reading a q-gram. The third algorithm is Fast Shift-Or (FSO) [5]. We utilized a version of FSO coded by B. Ďurian [4]. FSO was selected because it is fast on short binary patterns [4]. The fourth algorithm is the KMP algorithm [10]; together with verification it was supposed to approximate the two earlier methods [8,9] based on KMP. Of the algorithms, SBNDM2 and SBNDM4 are sublinear, whereas FSO and KMP are linear.

The tests were run on Intel 2.70 GHz i7 processor with 16 GB of memory running Ubuntu 12.10. All the algorithms were implemented in C in the 64 bit mode and run in the testing framework of Hume and Sunday [7]. Our solution based on filtration was compared with the earlier BMH approach by Cho et al. [2] (the authors generously let us use their implementation).

For testing we used three texts: a random text and two real texts, which were time series of the Dow Jones index and Helsinki temperatures. The random data contains 1,000,000 random integers between 0 and 2^{30}. The Dow Jones data contains 15,248 integers pertaining to the daily values of the stock index in the years 1950–2011 and the Helsinki temperature data contains 6,818 integers referring to the daily mean temperatures in Fahrenheit (multiplied by ten) in Helsinki in the years 1995–2005. From each text we picked random patterns of length $5, 8, 10, 12, 15$, and 20. Each set contains 1,000 patterns for the random

Table 1. Execution time of algorithms (random: in seconds, others: in 10 milliseconds)

Data	Algorithm	5	8	10	15	20	30	50
DOW	KOPM	2.02	1.94	1.94	2.00	1.94	1.96	1.95
	BMOPM-3	1.64	1.06	0.91	0.81	0.79	0.76	0.78
	BMOPM-4	2.16	0.96	0.72	0.50	0.41	0.34	0.31
	BMOPM-5	4.34	1.13	0.78	0.47	0.35	0.27	0.22
	FSO-OPM	1.01	0.48	0.46	0.46	0.44	0.46	0.46
	S2OPM	1.05	0.58	0.41	0.24	0.16	0.09	0.06
	S4OPM	1.03	0.31	0.20	0.31	0.10	0.08	0.06
Hel temp	KOPM	0.85	0.81	0.75	0.77	0.76	0.76	0.76
	BMOPM-3	0.70	0.46	0.46	0.40	0.39	0.39	0.44
	BMOPM-4	0.91	0.40	0.42	0.28	0.24	0.21	0.21
	BMOPM-5	1.87	0.49	0.52	0.31	0.23	0.18	0.17
	FSO-OPM	0.34	0.21	0.22	0.22	0.21	0.21	0.21
	S2OPM	0.40	0.18	0.13	0.08	0.06	0.03	0.03
	S4OPM	0.43	0.12	0.09	0.06	0.04	0.04	0.03
Random	KOPM	6.90	4.91	6.66	6.06	4.94	6.72	6.70
	BMOPM-3	8.08	4.01	4.69	4.17	4.08	4.10	4.07
	BMOPM-4	11.48	3.89	3.47	1.77	1.39	1.47	1.18
	BMOPM-5	22.83	5.97	4.70	2.67	1.88	1.22	0.79
	FSO-OPM	5.36	1.93	1.64	1.68	1.65	1.69	1.68
	S2OPM	3.90	2.47	1.89	1.17	1.26	0.79	0.35
	S4OPM	3.90	2.01	1.76	1.15	0.85	0.57	0.35

text and 200 patterns for the real texts. Table 1 shows the average execution times of all the algorithms. In addition, a graph on the times for the Dow Jones data is shown in Fig. 2. The real texts were tested with 180 repeated runs and the random text was tested with 60 repeated runs. In Table 1, S2OPM represents the algorithm based on SBNDM2 filtration, S4OPM represents the algorithm based on SBNDM4 filtration, BMOPM-q represents the BMH approach for $q = 3, 4, 5$, KOPM represents the algorithm based on KMP filtration and FSO-OPM represents the algorithm based on Fast Shift-Or.

From Table 1, it can be clearly seen that in case of real data, S4OPM is faster than all the other algorithms for all tested values of m except for $m = 5$. For $m = 5$, FSO-OPM is the fastest. In all the three cases, S2OPM is slower than S4OPM but its execution time approaches the execution time of S4OPM as the value of m increases. And for $m = 50$, the execution times of S2OPM and S4OPM are almost equal. Relatively, S2OPM and S4OPM performed better on the real data than on the random data. In the case of Helsinki daily temperatures, S2OPM and S4OPM both give almost the same performance. In the case of random data, for $m = 5$ S2OPM and S4OPM are the best and for $m = 8$ and for $m = 10$, FSO-OPM is the best.

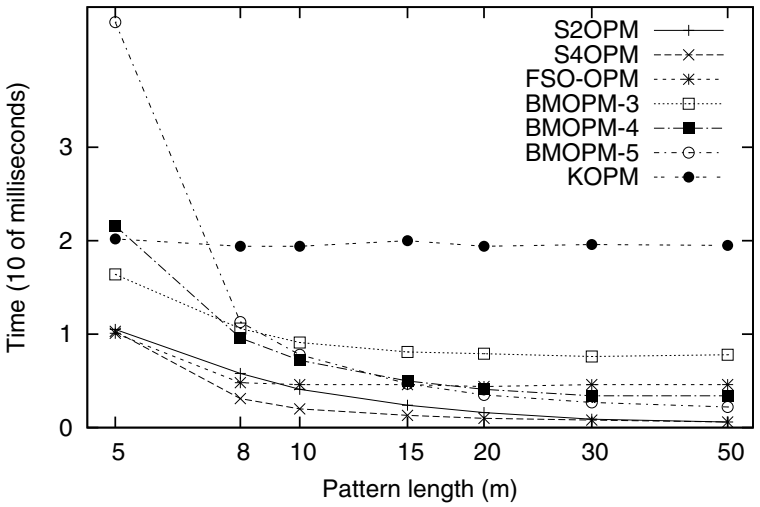

Fig. 2. Execution times of algorithms for the Dow Jones data

In case of real data like the Dow Jones stock index, it is also interesting to consider the number of matches. For $m = 5$, the number of matches is 56,048, for $m = 8$ number of matches is 2,133, and for $m = 10$ number of matches is 360.

6 Concluding Remarks

We introduced a new practical solution based on filtration for order-preserving matching. Any exact string matching algorithm can be used as the filtration algorithm. In this paper we have used SBNDM2, SBNDM4, FSO, and KMP as the filtration method of our solution. The results of our practical experiments prove that our SBNDM2 and SBNDM4 based solutions are much faster than the earlier BMOPM algorithm. Moreover, our FSO based solution is the fastest algorithm for certain short pattern lengths.

Acknowledgement. We thank Hannu Peltola for his help.

References

1. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Communications of the ACM 20(10), 762–772 (1977)
2. Cho, S., Na, J.C., Park, K., Sim, J.S.: Fast order-preserving pattern matching. In: Widmayer, P., Xu, Y., Zhu, B. (eds.) COCOA 2013. LNCS, vol. 8287, pp. 295–305. Springer, Heidelberg (2013)

3. Crochemore, M., Iliopoulos, C.S., Kociumaka, T., Kubica, M., Langiu, A., Pissis, S.P., Radoszewski, J., Rytter, W., Waleń, T.: Order-preserving incomplete suffix trees and order-preserving indexes. In: Kurland, O., Lewenstein, M., Porat, E. (eds.) SPIRE 2013. LNCS, vol. 8214, pp. 84–95. Springer, Heidelberg (2013)
4. Ďurian, B., Holub, J., Peltola, H., Tarhio, J.: Improving practical exact string matching. Information Processing Letters 110(4), 148–152 (2010)
5. Fredriksson, K., Grabowski, S.: Practical and optimal string matching. In: Consens, M.P., Navarro, G. (eds.) SPIRE 2005. LNCS, vol. 3772, pp. 376–387. Springer, Heidelberg (2005)
6. Horspool, R.N.: Practical fast searching in strings. Software–Practice and Experience 10(6), 501–506 (1980)
7. Hume, A., Sunday, D.: Fast string searching. Software–Practice and Experience 21(11), 1221–1248 (1991)
8. Kim, J., Eades, P., Fleischer, R., Hong, S.-H., Iliopoulos, C.S., Park, K., Puglisi, S.J., Tokuyama, T.: Order preserving matching. Theoretical Computer Science 525, 68–79 (2014)
9. Kubica, M., Kulczynski, T., Radoszewski, J., Rytter, W., Walen, T.: A linear time algorithm for consecutive permutation pattern matching. Information Processing Letters 113(12), 430–433 (2013)
10. Knuth, D.E., Morris, J.M., Pratt, V.R.: Fast pattern matching in strings. SIAM Journal on Computing 6(2), 323–350 (1977)
11. Navarro, G., Raffinot, M.: Flexible pattern matching in strings. Practical on-line search algorithms for texts and biological sequences. Cambridge University Press, New York (2002)