Theoretical Computer Science 525 (2014) 68–79

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Order-preserving matching

Jinil Kim[a], Peter Eades[b], Rudolf Fleischer[c,d], Seok-Hee Hong[b], Costas S. Iliopoulos[e,f], Kunsoo Park[a,*], Simon J. Puglisi[g], Takeshi Tokuyama[h]

[a] *Department of Computer Science and Engineering, Institute of Computer Technology, Seoul National University, Seoul, Republic of Korea*
[b] *School of Information Technologies, University of Sydney, Australia*
[c] *SCS and IIPL, Fudan University, Shanghai, China*
[d] *Department of Applied Information Technology, German University of Technology in Oman, Muscat, Oman*
[e] *Department of Informatics, King's College London, London, United Kingdom*
[f] *Digital Ecosystems and Business Intelligence Institute, Curtin University, Australia*
[g] *Department of Computer Science, University of Helsinki, Finland*
[h] *Graduate School of Information Sciences, Tohoku University, Japan*

ARTICLE INFO

Keywords:
String matching
Numeric string
Order relation
Multiple pattern matching
KMP algorithm
Aho–Corasick algorithm

ABSTRACT

We introduce a new string matching problem called *order-preserving matching* on numeric strings, where a pattern matches a text if the text contains a substring of values whose relative orders coincide with those of the pattern. Order-preserving matching is applicable to many scenarios such as stock price analysis and musical melody matching in which the order relations should be matched instead of the strings themselves. Solving order-preserving matching is closely related to the *representation of order relations* of a numeric string. We define the *prefix representation* and the *nearest neighbor representation* of the pattern, both of which lead to efficient algorithms for order-preserving matching. We present efficient algorithms for single and multiple pattern cases. For the single pattern case, we give an $O(n \log m)$ time algorithm and optimize it further to obtain $O(n + m \log m)$ time. For the multiple pattern case, we give an $O(n \log m)$ time algorithm.

1. Introduction

String matching is a fundamental problem in computer science and has been extensively studied. Sometimes a string consists of numeric values instead of characters in an alphabet, and we are interested in some *trends* in the text rather than specific patterns. For example, in a stock market, analysts may wonder whether there is a period when the share price of a company dropped consecutively for 10 days and then went up for the next 5 days. In such cases, the changing patterns of share prices are more meaningful than the absolute prices themselves. Another example is melody matching between two musical scores. A musician may be interested in whether her new song has a melody similar to well-known songs. As many variations are possible in a melody where the relative heights of pitches are preserved but the absolute pitches can be changed, it would be reasonable to match relative pitches instead of absolute pitches to find similar musical phrases.

An *order-preserving matching* can be helpful in both examples, because a pattern is matched with the text if the text contains a substring of values whose relative orders coincide with those of the pattern. For example, in Fig. 1, pattern

* Corresponding author. Tel.: +82 2 880 1828; fax: +82 2 885 3141.
E-mail addresses: jikim@theory.snu.ac.kr (J. Kim), peter.eades@sydney.edu.au (P. Eades), rudolf@fudan.edu.cn (R. Fleischer), seokhee.hong@sydney.edu.au (S.-H. Hong), c.iliopoulos@kcl.ac.uk (C.S. Iliopoulos), kpark@theory.snu.ac.kr (K. Park), puglisi@cs.helsinki.fi (S.J. Puglisi), tokuyama@dais.is.tohoku.ac.jp (T. Tokuyama).

0304-3975/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.tcs.2013.10.006

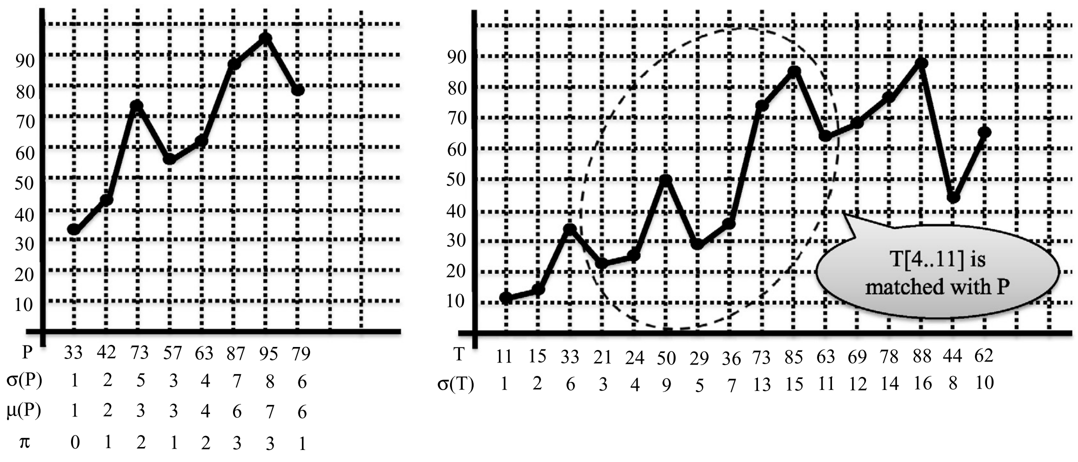

Fig. 1. Example of pattern and text.

$P = (33, 42, 73, 57, 63, 87, 95, 79)$ is matched with text T since the substring $(21, 24, 50, 29, 36, 73, 85, 63)$ in the text has the same relative orders as the pattern. In both strings, the first characters 33 and 21 are the smallest, the second characters 42 and 24 are the second smallest, the third characters 73 and 50 are the 5-th smallest, and so on. If we regard prices of shares, or absolute pitches of musical notes, as numeric characters of the strings, both examples above can be modeled as order-preserving matching.

Solving order-preserving matching is closely related to *representations of order relations* of a numeric string. If we replace each character in a numeric string by its rank in the string, then we can obtain a (natural) representation of order relations. But this natural representation is not amenable to developing efficient algorithms because the rank of a character depends on the substring in which the rank is computed. Hence, we define the *prefix representation* of order relations, which leads to an $O(n \log m)$ time algorithm for order-preserving matching, where n and m are the lengths of the text and the pattern, respectively. Surprisingly, however, there is an even better representation, called the *nearest neighbor representation*, with which we were able to develop an $O(n + m \log m)$ time algorithm.

In this paper, we define a new class of string matching problem, called *order-preserving matching*, and present efficient algorithms for single and multiple pattern cases. For the single pattern case, we propose an $O(n \log m)$ algorithm based on the Knuth–Morris–Pratt (KMP) algorithm [18,20], and optimize it further to obtain $O(n + m \log m)$ time. For the multiple pattern case, we present an $O(n \log m)$ algorithm based on the Aho–Corasick algorithm [1].

Related work: Norm matching and (δ, γ)-matching have been studied to search for similar patterns of numeric strings. In norm matching [8,25,2,28], each text substring and the pattern is matched if the L_p distance is less than the predefined value for some given p. In (δ, γ)-matching [12,19,16,15,23,24,26], two parameters δ and γ are given, and two numeric strings of the same length are matched if the maximum difference of the corresponding characters is at most δ and the total sum of differences is at most γ. Several variants were studied to allow for *don't care* symbols [17], transposition-invariant [23] and gaps [13,14,21]. On the other hand, some generalized matching problems such as parameterized matching [10,7], less than matching [6], swapped matching [3,27], overlap matching [5], and function matching [4,9] are studied extensively where *matching* relations are defined differently so that some properties of two strings are matched instead of exact matching of characters. However, none of this prior work addresses the *order relations*, which we focus on in this paper.

2. Problem formulation

2.1. Notations

Let Σ denote the set of numbers such that a comparison of two numbers can be done in constant time, and let Σ^* denote the set of strings over the alphabet Σ. Let $|x|$ denote the length of a string x. A string x is described by either a concatenation of characters $x[1] \cdot x[2] \cdot \cdots \cdot x[|x|]$ or as a sequence of characters $(x[1], x[2], \ldots, x[|x|])$ interchangeably. For a string x, let a substring $x[i..j]$ be $(x[i], x[i+1], \ldots, x[j])$ and the prefix x_i be $x[1..i]$. The rank of a character c in string x is defined as $rank_x(c) = 1 + |\{i\colon\ x[i] < c \text{ for } 1 \leqslant i \leqslant |x|\}|$. For simplicity, we assume that all the numbers in a string are *distinct*. When a number occurs more than once in a string, we can extend our character definition to a pair (character, index) so that the characters in the string become distinct.

2.2. Natural representation of order relations

For a string x, the *natural representation* of the order relations can be defined as $\sigma(x) = rank_x(x[1]) \cdot rank_x(x[2]) \cdot \cdots \cdot rank_x(x[|x|])$.

Function	Description
OS-Insert($\mathcal{T}, x, i$)	Insert $(x[i], i)$ to $\mathcal{T}$
OS-Delete($\mathcal{T}, x$)	Delete all the characters of string x from $\mathcal{T}$
OS-Rank($\mathcal{T}, c$)	Compute rank r of character c in $\mathcal{T}$
OS-Find-Prev-Index($\mathcal{T}, c$)	Find the index i of the largest character less than c
OS-Find-Next-Index($\mathcal{T}, c$)	Find the index i of the smallest character greater than c

Fig. 2. List of functions on $\mathcal{T}$ for dynamic order statistics.

Definition 2.1 (*Order-preserving matching*). Given a text $T[1..n] \in \Sigma^*$ and a pattern $P[1..m] \in \Sigma^*$, T is matched with P at position i if $\sigma(T[i-m+1..i]) = \sigma(P)$. Order-preserving matching is the problem of finding all positions of T matched with P.

For example, let's consider the two strings $P = (33, 42, 73, 57, 63, 87, 95, 79)$ and $T = (11, 15, 33, 21, 24, 50, 29, 36, 73, 85, 63, 69, 78, 88, 44, 62)$ shown in Fig. 1. The natural representation of P is $\sigma(P) = (1, 2, 5, 3, 4, 7, 8, 6)$, which is matched with $T[4..11] = (21, 24, 50, 29, 36, 73, 85, 63)$ at position 11 but is not matched at the other positions of T.

As the rank of a character depends on the substring in which the rank is computed, the string matching algorithms with $O(n+m)$ time complexity such as KMP, Boyer–Moore [18,20] cannot be applied directly. For example, the rank of $T[4]$ is 3 in $T[1..8]$ but is changed to 1 in $T[4..11]$.

The naive pattern matching algorithm is applicable to order-preserving matching if both the pattern and the text are converted to natural representations. If we use *the order-statistic tree* based on the red–black tree [18], computing the rank of a character in the string x takes $O(\log|x|)$, which makes the computation time of the natural representation $\sigma(x)$ be $O(|x|\log|x|)$. The naive order-preserving matching algorithm computes $\sigma(P)$ in $O(m\log m)$ time and $\sigma(T[i..i+m-1])$ for each position $i \in [1..n-m+1]$ of text T in $O(m\log m)$ time, and compares them in $O(m)$ time. As $n-m+1$ positions are considered, the total time complexity becomes $O((n-m+1)\cdot(m\log m)) = O(nm\log m)$. As this time complexity is much worse than $O(n+m)$ which we can obtain from the exact pattern matching, sophisticated matching techniques need to be considered for order-preserving matching as discussed in later sections.

3. $O(n\log m)$ algorithm

3.1. Prefix representation

An alternative way of representing order relations is to use the rank of each character in the prefix. Formally, the *prefix representation* of order relations can be defined as $\mu(x) = rank_{x_1}(x[1]) \cdot rank_{x_2}(x[2]) \cdot \cdots \cdot rank_{x_{|x|}}(x[|x|])$. For example, the prefix representation of P in Fig. 1 is $\mu(x) = (1, 2, 3, 3, 4, 6, 7, 6)$.

An advantage of the prefix representation is that $\mu(x)[i]$ can be computed without looking at characters in $x[i+1..|x|]$ ahead of position i. By using the order-statistic tree $\mathcal{T}$ for dynamic order statistics [18] containing characters of $x[1..i-1]$, $\mu(x)[i]$ can be computed in $O(\log|x|)$ time. Moreover, the prefix representation can be updated incrementally by inserting the next character to $\mathcal{T}$ or deleting the previous character from $\mathcal{T}$. Specifically, when $\mathcal{T}$ contains the characters in $x[1..i]$, $\mu(x[1..i+1])[i+1]$ can be computed if $x[i+1]$ is inserted to $\mathcal{T}$, and $\mu(x[2..i])[i-1]$ can be computed if $x[1]$ is deleted from $\mathcal{T}$.

Note that there is a *one-to-one* mapping between the natural representation and the prefix representation. The number of all the distinct natural representations for a string of length n is $n!$ which corresponds to the number of permutations, and the number of all the distinct prefix representations is $n!$ too, since there are i possible values for the i-th character of a prefix representation, which results in $1 \cdot 2 \cdot \cdots \cdot n = n!$ cases. For any natural representation of a string, there is a conversion function which returns the corresponding prefix representation and vice versa.

The prefix representation of P is easily computed by inserting each character $P[k]$ to $\mathcal{T}$ consecutively as in Compute-Prefix-Rep. The functions of the order-statistic tree are listed in Fig. 2. We assume that the index i of x is stored with $x[i]$ in OS-Insert($\mathcal{T}, x, i$) to support OS-Find-Prev-Index($\mathcal{T}, c$) and OS-Find-Next-Index($\mathcal{T}, c$) where the index i of the largest (smallest) character less than (greater than) c is retrieved.

```
Compute-Prefix-Rep($P$)
1  $m \leftarrow |P|$
2  $\mathcal{T} \leftarrow \phi$
3  OS-Insert($\mathcal{T}, P, 1$)
4  $\mu(P)[1] \leftarrow 1$
5  for $k \leftarrow 2$ to $m$
6      OS-Insert($\mathcal{T}, P, k$)
7      $\mu(P)[k] \leftarrow$ OS-Rank($\mathcal{T}, P[k]$)
8  return $\mu(P)$
```

The time complexity of Compute-Prefix-Rep is $O(m \log m)$ as each of OS-Insert and OS-Rank takes $O(\log m)$ time and there are $O(m)$ such operations.

3.2. KMP failure function

The KMP-style failure function π of order-preserving matching is well-defined under our prefix representation:

$$\pi[q] = \begin{cases} \max\{k\colon \mu(P[1..k]) = \mu(P[q-k+1..q]) \text{ for } 1 \leqslant k < q\} & \text{if } q > 1 \\ 0 & \text{if } q = 1 \end{cases}$$

Intuitively, π means that the longest proper prefix $\mu(P[1..k])$ of P is matched with $\mu(P[q-k+1..q])$ which is the prefix representation of the suffix of $P[1..q]$ with length k. For example, the failure function of P in Fig. 1 is $\pi[1..m] = (0,1,2,1,2,3,3,1)$. As shown in Fig. 3, $\pi[6] = 3$ implies that the longest prefix of $\mu(P[1..8])$ that is matched with the prefix representation of any suffix of $P[1..6] = (33,42,63,57,63,87)$ is $\mu(P[1..\pi[6]]) = (1,2,3)$.

The construction algorithm of π will be given in Section 3.4.

3.3. Text search

The failure function π can accelerate order-preserving matching by filtering mismatched positions as in the KMP algorithm. Let's assume that $\mu(P)[1..q]$ is matched with $\mu(T[i-q..i-1])[1..q]$ but a mismatch is found between $\mu(P)[q+1]$ and $\mu(T[i-q..i])[q+1]$. $\pi[q]$ means that $\mu(P)[1..\pi[q]]$ is already matched with $\mu(T[i-\pi[q]..i-1])[1..\pi[q]]$ and matching can be continued at $P[\pi[q]+1]$ comparing $\mu(P)[\pi[q]+1]$ with $\mu(T[i-\pi[q]..i])[\pi[q]+1]$. Since $P[1..\pi[q]]$ is the longest prefix whose order is matched with the suffix of $T[i-q..i-1]$, the positions from $i-q$ to $i-\pi[q]-1$ can be skipped without any comparisons as in the KMP algorithm. Fig. 3 shows how π can filter mismatched positions. When $\mu(P)[1..6]$ is matched with $\mu(T[1..6])$ but $\mu(P)[7]$ is different from $\mu(T[1..7])[7]$, we can skip the positions from 1 to 3 of P and continue by comparing $\mu(P)[4]$ with $\mu(T[4..7])[4]$.

KMP-Order-Matcher describes the order-preserving matching algorithm assuming that $\mu(P)$ and π are efficiently computed. In KMP-Order-Matcher, for each index i of T, q is maintained as the length of the longest prefix of P where $\mu(P)[1..q]$ is matched with $\mu(T)[i-q..i-1]$. At that time, the order-statistic tree $\mathcal{T}$ contains all the characters of $T[i-q..i-1]$. If the rank of $T[i]$ in $\mathcal{T}$ is not matched with that of $P[q+1]$, q is reduced to $\pi[q]$ by deleting all the characters $T[i-q..i-\pi[q]-1]$ from $\mathcal{T}$. If $P[q+1]$ and $T[i]$ have the same rank, i.e., $\mu(P)[1..q+1] = \mu(T)[i-q..i]$, the length of the matched pattern q is increased by 1. When q reaches m, the relative order of $T[i-m-1..i]$ matches the one of P.

```
KMP-Order-Matcher($T$, $P$)
 1  $n \leftarrow |T|$, $m \leftarrow |P|$
 2  $\mu(P) \leftarrow$ Compute-Prefix-Rep($P$)
 3  $\pi \leftarrow$ KMP-Compute-Failure-Function($P$, $\mu(P)$)
 4  $\mathcal{T} \leftarrow \phi$
 5  $q \leftarrow 0$
 6  for $i \leftarrow 1$ to $n$
 7      OS-Insert($\mathcal{T}$, $T$, $i$)
 8      $r \leftarrow$ OS-Rank($\mathcal{T}$, $T[i]$)
 9      while $q > 0$ and $r \neq \mu(P)[q+1]$
10          OS-Delete($\mathcal{T}$, $T[i-q..i-\pi[q]-1]$)
11          $q \leftarrow \pi[q]$
12          $r \leftarrow$ OS-Rank($\mathcal{T}$, $T[i]$)
13      $q \leftarrow q+1$
14      if $q = m$
15          print "pattern occurs at position" $i$
16          OS-Delete($\mathcal{T}$, $T[i-q..i-\pi[q]-1]$)
17          $q \leftarrow \pi[q]$
```

KMP-Order-Matcher is different from the original KMP algorithm used for exact pattern matching in that it matches order relations instead of characters. For each position i of T, the prefix representation $\mu(T[i-q..i])[q+1]$ of $T[i]$ is computed using the order-statistic tree $\mathcal{T}$. If $\mu(T[i-q..i])[q+1]$ does not match $\mu(P)[q+1]$, q is reduced to $\pi[q]$ so that P implicitly shifts right by $q-\pi[q]$.

Another subtle difference is that we do not check whether $r = \mu(P)[q+1]$ before increasing q by 1 in line 13 (cf. [18,20]) because it should be satisfied automatically. From the condition of the while loop in line 9, $q = 0$ or $r = \mu(P)[q+1]$ in line 13, and if $q = 0$, $\mu(P)[1] = 1$ for any pattern and it matches any text of length 1.

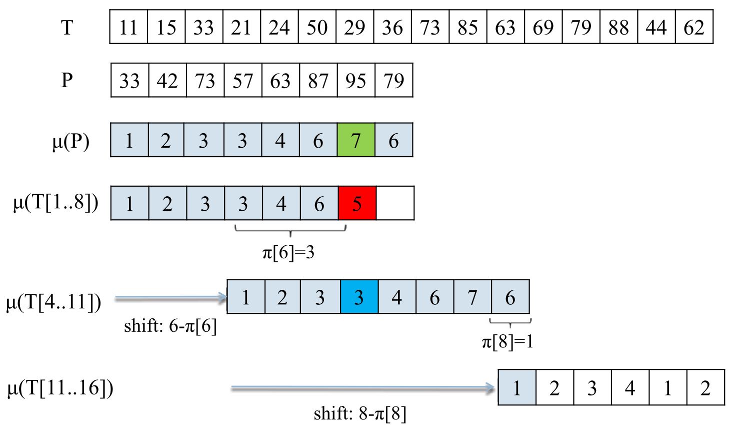

Fig. 3. Example of text search.

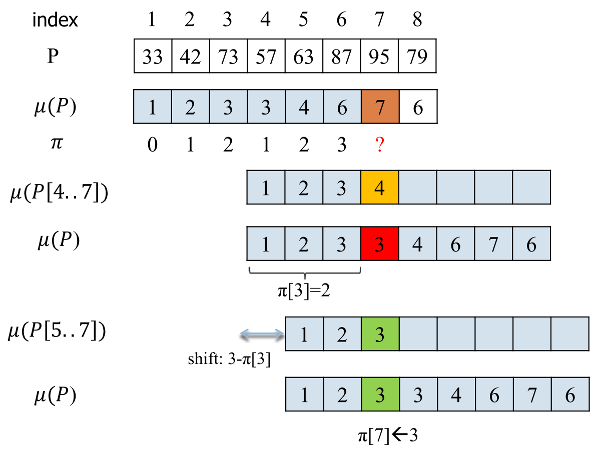

Fig. 4. Example of computing failure function.

The time required in KMP-ORDER-MATCHER, except for the computation of the prefix representation of P and the construction of the failure function π, can be analyzed as follows. Each OS-INSERT, OS-RANK is done in $O(\log m)$ time while OS-DELETE takes $O(\log m)$ time per character deletion. The number of calls to OS-INSERT is n, and the number of deletions is at most n, which makes the total time of deletions $O(n \log m)$. In the same way, the number of calls to OS-RANK is bounded by $2n$, n for new characters, and the other n for the computation of rank after reducing q, and thus the total cost of OS-RANK calls is also $O(n \log m)$. To sum up, the time for KMP-ORDER-MATCHER can be bounded by $O(n \log m)$ except for the external functions.

3.4. Construction of KMP failure function

The construction of failure function π can be done similarly to the text matching phase of the KMP algorithm, where each element $\pi[q]$ is computed by using the previous values $\pi[1..q-1]$.

KMP-COMPUTE-FAILURE-FUNCTION describes the construction algorithm of π. It first tries to compute $\pi[q]$ starting from the match of $\mu(P[1..\pi[q-1]])$ and $\mu(P[q-\pi[q-1]..q-1])$. If $\mu(P[1..\pi[q-1]+1])[\pi[q-1]+1] = \mu(P[q-\pi[q-1]..q])[\pi[q-1]+1]$, set $\pi[q] = \pi[q-1]+1$. Otherwise, it tries another match for $\pi[\pi[1..q-1]]$, and repeats until $\pi[q]$ is computed.

Fig. 4 shows an example of computing the failure function of P in Fig. 1 in which $\pi[7]$ is being computed. Starting from $q = \pi[6] = 3$, KMP-ORDER-MATCHER tries to match $\mu(P[4..8])[4]$ with $\mu(P)[4]$ but it fails. Then, q is decreased to $q = \pi[3] = 2$ and it tries to match $\mu(P[5..8])[3]$ with $\mu(P)[3]$ and it succeeds. $\pi[7]$ is assigned to $\pi[3]+1$, and the next iteration is started with $q = \pi[7]$.

```
KMP-COMPUTE-FAILURE-FUNCTION($P$, $\mu(P)$)
 1  $m \leftarrow |P|$
 2  $\mathcal{T} \leftarrow \phi$
 3  OS-INSERT($\mathcal{T}$, $P$, 1)
 4  $k \leftarrow 0$
 5  $\pi[1] \leftarrow 0$
 6  for $q \leftarrow 2$ to $m$
 7      OS-INSERT($\mathcal{T}$, $P$, $q$)
 8      $r \leftarrow$ OS-RANK($\mathcal{T}$, $P[q]$)
 9      while $k > 0$ and $r \neq \mu(P)[k+1]$
10          OS-DELETE($\mathcal{T}$, $P[q-k..q-\pi[k]-1]$)
11          $k \leftarrow \pi[k]$
12          $r \leftarrow$ OS-RANK($\mathcal{T}$, $P[q]$)
13      $k \leftarrow k+1$
14      $\pi[q] \leftarrow k$
15  return $\pi$
```

The time complexity of KMP-COMPUTE-FAILURE-FUNCTION can be analyzed in a similar way to KMP-ORDER-MATCHER, by replacing the length of T with the length of P, which results in $O(m\log m)$ time.

3.5. Correctness and time complexity

The correctness of our matching algorithm is due to the failure function being defined the same way as the original KMP algorithm. From the analysis of Sections 3.3 and 3.4, it is clear that our algorithm does not miss any matching position.

The total time complexity is $O(n\log m)$, with $O(m\log m)$ to compute the prefix representation and failure function and $O(n\log m)$ for text search. Compared with $O(n)$ time of the exact pattern matching, our algorithm has the overhead of $O(\log m)$ factor, which is optimized in Section 4.

3.6. Remark on the good/bad character heuristics

Variants of the Boyer–Moore algorithm [11,22,29] may be designed for order-preserving matching in which case the prefix representation should be replaced by the *suffix* representation to facilitate accessing the pattern from right to left during matching. The good suffix heuristic [11] is well-defined with the suffix representation, but the bad character heuristic [11] is not applicable since the character itself has nothing to do with order relations. As the performance of the Boyer–Moore algorithm is significantly dependent on the bad character heuristic, we cannot expect that the gain of Boyer–Moore variants for order-preserving matching is comparable to that of the original Boyer-Moore algorithm for the exact matching. Moreover, some practical algorithms such as the Horspool [22] and the Sunday algorithms [29] cannot be applied to order-preserving matching because they employ only the bad character heuristic for filtering mismatched positions.

4. $O(n+m\log m)$ algorithm

4.1. Nearest neighbor representation

The text search of the previous algorithm can be optimized further to remove the $O(\log m)$ overhead of computing rank functions. In the text search phase of the $O(n\log m)$ algorithm, the rank of each character $T[i]$ in $T[i-q-1..i]$ is computed to check whether it is matched with $\mu(P)[q+1]$ when we know that $\mu(P)[1..q]$ is matched with $\mu(T[i-q+1..i])$. If we can do it directly without computing $\mu(P)[q+1]$, the overhead of the operations on $\mathcal{T}$ can be removed.

The main idea is to check whether the order of each character in the text matches that of the corresponding character in the pattern by comparing the characters themselves without computing rank values explicitly. When we need to check if a character $x[i]$ of string x has a specific rank value r in prefix x_i, we can do it by checking $x[j] < x[i] < x[k]$ where $x[j]$ and $x[k]$ are characters having rank values nearest to r.

The *nearest neighbor representation* of the order relations can be defined as follows. For a string x, let $\nu_p(x)[1..|x|]$ and $\nu_n(x)[1..|x|]$ be the nearest neighbor representations of x where $\nu_p(x)[i]$ is the index of the largest character of x_{i-1} less than $x[i]$ and $\nu_n(x)[i]$ is the index of the smallest character of x_{i-1} greater than $x[i]$. Let $\nu_p(x)[i] = -\infty$ if there is no character less than $x[i]$ in x_{i-1} and let $\nu_n(x)[i] = \infty$ if there is no character greater than $x[i]$ in x_{i-1}. Let $x[-\infty] = -\infty$ and $x[\infty] = \infty$.

The advantage of the nearest neighbor representation is that we can check whether each text character is matched with the corresponding pattern character in constant time without computing its rank. Fig. 5 shows the nearest neighbor representation of the order relations of P in Fig. 1. Suppose that $\mu(P)[1..i-1] = \mu(T[1..i-1])$ for $1 \leqslant i \leqslant m$. If $T[\nu_p(P)[i]] < T[i] < T[\nu_n(P)[i]]$, then $\mu(P[1..i]) = \mu(T[1..i])$. For example, $\mu(T[1])[1]$ must be matched with $\mu(P)[1]$ since $T[\nu_p(P)[1]] <$

i	1	2	3	4	5	6	7	8
$\mu(P)[i]$	1	2	3	3	4	6	7	6
$\pi[i]$	0	1	2	1	2	3	3	1
$\nu_p(P)[i]$	$-\infty$	1	2	2	4	3	6	3
$\nu_n(P)[i]$	∞	∞	∞	3	3	∞	∞	6

Fig. 5. Example of the nearest neighbor representation of the pattern P.

$c < T[\nu_n(P)[1]]$ for any character c, which coincides with the fact that the rank in the text of size 1 is always 1. For the second character, $\mu(P)[2] = 2$ and $T[2]$ should be larger than $T[1]$ to have $\mu(T[1..2])[2] = 2$, which is represented by $\nu_p(P)[1] = 1$ and $\nu_n(P)[1] = \infty$. In this way, for each character, we can decide whether the order of $T[i]$ in $\mu(T[1..i])$ is matched with that of $P[i]$ in $\mu(P[1..i])$ by checking $T[\nu_p(P)[i]] < T[i] < T[\nu_n(P)[i]]$.

Compute-Nearest-Neighbor-Rep describes the construction of the nearest neighbor representation of the string P, where $\mathcal{T}$ contains the characters of P_{k-1} in each step of the loop. We assume that OS-Find-Prev-Index$(\mathcal{T}, c)$ (and OS-Find-Next-Index$(\mathcal{T}, c)$) returns the index i of the largest (smallest) character less than (greater than) c, and returns $-\infty$ (∞) if there is no such character.

```
Compute-Nearest-Neighbor-Rep($P$)
1  $m \leftarrow |P|$
2  $\mathcal{T} \leftarrow \phi$
3  OS-Insert($\mathcal{T}$, $P$, 1)
4  $(\nu_p(P)[1], \nu_n(P)[1]) \leftarrow (-\infty, \infty)$
5  for $k \leftarrow 2$ to $m$
6      OS-Insert($\mathcal{T}$, $P$, $k$)
7      $\nu_p(P)[k] \leftarrow$ OS-Find-Prev-Index($\mathcal{T}$, $P[k]$)
8      $\nu_n(P)[k] \leftarrow$ OS-Find-Next-Index($\mathcal{T}$, $P[k]$)
9  return $(\nu_p(P), \nu_n(P))$
```

The time complexity of Compute-Nearest-Neighbor-Rep is $O(m \log m)$ since it has m iterations of the loop and there are 3 function calls on the order-statistic tree $\mathcal{T}$ taking $O(\log m)$ time in each iteration.

4.2. Text search

With the nearest neighbor representation of pattern P and the failure function π, we can simplify the text search so that it does not involve $\mathcal{T}$ at all. For each character $T[i]$, we can check $\mu(P)[q+1] = \mu(T[i-q..i])[q+1]$ by comparing $T[i]$ with the characters in $T[i-q..i]$ whose indexes correspond to $\nu_p(P)[q+1]$ and $\nu_n(P)[q+1]$ in P. Specifically, if $T[i-q+\nu_p(P)[q+1]-1] < T[i] < T[i-q+\nu_n(P)[q+1]-1]$, then $\mu(P)[q+1] = \mu(T[i-q..i])[q+1]$ must be satisfied since the relative order of $T[i]$ in $T[i-q..i]$ is the same as that of $P[q+1]$ in $P[1..q+1]$.

To illustrate this, let us return to the text matching example in Fig. 3. When $\mu(P)[1..6]$ is matched with $\mu(T[1..6])$, we can check if $\mu(T[1..7])[7]$ is matched with $\mu(P)[7]$ by checking if $T[7-6+\nu_p(P)[7]-1] < T[7] < T[7-6+\nu_n(P)[7]-1]$, which can be done in constant time. As $T[6] = 50$, $T[\infty] = \infty$ but $T[7] = 29$, $T[7]$ should have a rank lower than $\mu(P)[7]$, thus $\mu(T[1..7])$ cannot be matched with $\mu(P)[1..7]$.

KMP-Order-Matcher2 describes the text search algorithm using the nearest neighbor representation. The algorithm is essentially equivalent to the previous one but simpler since no rank function has to be calculated explicitly.

```
KMP-Order-Matcher2($T$, $P$)
1   $n \leftarrow |T|$, $m \leftarrow |P|$
2   $(\nu_p(P), \nu_n(P)) \leftarrow$ Compute-Nearest-Neighbor-Rep($P$)
3   $\pi \leftarrow$ KMP-Compute-Failure-Function2($P$, $\nu_p(P)$, $\nu_n(P)$)
4   $q \leftarrow 0$
5   for $i \leftarrow 1$ to $n$
6       $(j_1, j_2) \leftarrow (\nu_p(P)[q+1], \nu_n(P)[q+1])$
7       while $q > 0$ and $(T[i] < T[i-q+j_1-1]$ or $T[i] > T[i-q+j_2-1])$
8           $q \leftarrow \pi[q]$
9           $(j_1, j_2) \leftarrow (\nu_p(P)[q+1], \nu_n(P)[q+1])$
10      $q \leftarrow q+1$
11      if $q = m$
12          print "pattern occurs at position" $i$
13          $q \leftarrow \pi[q]$
```

The time complexity of KMP-Order-Matcher2 except for the precomputation of the prefix representation and the failure function is $O(n)$ because only one scan of the text is required in the for loop as in the KMP algorithm.

4.3. Construction of KMP failure function

The construction of the failure function π is an extension of KMP-Compute-Failure-Function in Section 3.4 where the rank function on $\mathcal{T}$ is replaced by a comparison of characters using $\nu_p(P)$ and $\nu_n(P)$ as in KMP-Order-Matcher2. KMP-Compute-Failure-Function2 describes the construction of the KMP failure function from the nearest neighbor representation of pattern P.

```
KMP-Compute-Failure-Function2($P$, $\nu_p(P)$, $\nu_n(P)$)
1  $m \leftarrow |P|$
2  $k \leftarrow 0$
3  $\pi[1] \leftarrow 0$
4  for $q \leftarrow 2$ to $m$
5      $(j_1, j_2) \leftarrow (\nu_p(P)[k+1], \nu_n(P)[k+1])$
6      while $k > 0$ and $(P[q] < P[i-k+j_1-1]$ or $P[q] > P[i-k+j_2-1])$
7          $k \leftarrow \pi[k]$
8          $(j_1, j_2) \leftarrow (\nu_p(P)[k+1], \nu_n(P)[k+1])$
9      $k \leftarrow k+1$
10     $\pi[q] \leftarrow k$
11 return $\pi$
```

The time complexity of KMP-Compute-Failure-Function2 is $O(m)$ from the linear scan of the pattern, similarly to KMP-Order-Matcher2.

4.4. Correctness and time complexity

The correctness of our optimized algorithm is derived from that of the previous $O(n \log m)$ algorithm since the difference of the text search is only on rank comparison logic and each comparison result is the same as the previous one. The same failure function π is applied and the order-statistic tree $\mathcal{T}$ is only used to compute the nearest neighbor representation of P.

The time complexity of the overall algorithm is $O(n + m \log m)$: $O(m \log m)$ time for the computation of the nearest neighbor representation of the pattern, $O(m)$ time for the construction of π function, and $O(n)$ time for text search. $O(n + m \log m)$ is almost linear to the text length n when n is much larger than m, which is a typical case in pattern matching problems. The only non-linear factor $\log m$ comes from computing the representation of order relations.

4.5. Generalized order-preserving matching

A generalization of order-preserving matching is possible with some practical applications if we consider only the orders of the last k characters for a given $k \leqslant m$. For example, in the stock market scenario of finding a period when a share price of a company dropped consecutively for 10 days and then went up for the next 5 days, it is sufficient to compare each share price with the share price of the day before, which corresponds to $k = 1$. Our solution is easily applicable to this generalized problem if the order-statistic tree $\mathcal{T}$ is maintained to keep only the last k inserted characters. The time complexity of the $O(n \log m)$ algorithm that uses prefix representation becomes $O(n \log k)$, and that of the $O(n + m \log m)$ algorithm that uses nearest neighbor representation becomes $O(n + m \log k)$, since the number of characters in $\mathcal{T}$ is bounded to k. Both time complexities are reduced to $O(n)$ if k is a constant number.

4.6. Remark on the alphabet size

We have no restrictions on the numbers in Σ, insofar as a comparison of two numbers can be done in constant time. In the case of $\Sigma = \{1, 2, \dots, U\}$, however, the order-statistic tree in Compute-Nearest-Neighbor-Rep can be replaced by a van Emde Boas tree [30] or y-fast trie [31] which takes $O(U)$ space and requires $O(\log \log U)$ time per operation.

5. $O(n \log m)$ algorithm for multiple patterns

In this section, we consider a generalization of order-preserving matching for multiple patterns.

Definition 5.1 *(Order-preserving matching for multiple patterns).* Given a text $T[1..n] \in \Sigma^*$ and a set of patterns $\mathcal{P} = \{P_1, P_2, \dots, P_w\}$ where $P_i \in \Sigma^*$ for all $1 \leqslant i \leqslant w$, order-preserving matching for multiple patterns is the problem of finding all positions of T matched with any pattern in $\mathcal{P}$.

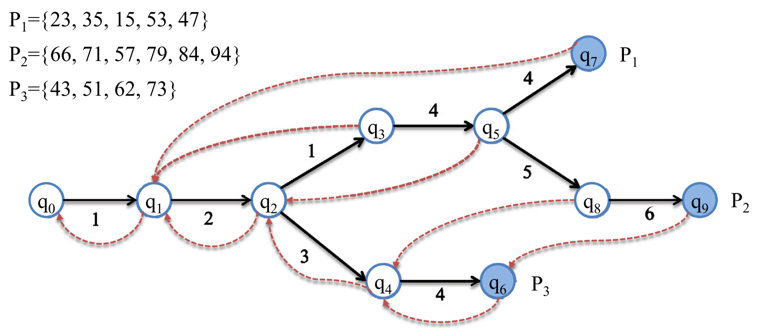

Fig. 6. Example of AC automaton and failure function.

We propose a variant of the Aho–Corasick algorithm [1] for the multiple pattern case whose time complexity is $O(n\log m)$ where m is the sum of the lengths of the patterns.

5.1. Prefix representation of the Aho–Corasick automaton

From the prefix representation of the given patterns, an Aho–Corasick automaton can be defined to match order relations. The Aho–Corasick automaton consists of the following components.

1. Q : a finite set of states where $q_0 \in Q$ is the initial state.
2. $g : Q \times \mathbb{N}_m \to Q \cup \{\text{fail}\}$: a forward transition function. $\mathbb{N}_m$ is the set of integers in $[1..m]$.
3. $\pi : Q \to Q$: a failure function.
4. $d : Q \to \mathbb{Z}$: the length of the prefix represented by each state q.
5. $P : Q \to \mathcal{P}$: a representative pattern of each state q which has the prefix represented by q. If there are more than one such patterns, we use the pattern with the smallest index.
6. $out : Q \to \mathcal{P} \cup \{\phi\}$: the output pattern of each state q. If q does not match any pattern, $out[q] = \phi$, otherwise $out[q] = P_i$ for the longest pattern P_i such that the prefix representation of P_i is matched with that of any suffix of $P[q][1..d[q]]$.

Given the set of patterns, an Aho–Corasick automaton of the prefix representations is constructed from a trie in which each node represents a prefix of the prefix representation of some pattern. The nodes of the trie are the states of the automaton and the root is the initial state q_0, representing the empty prefix. Each node q is an accepting state if $out[q] \neq \phi$, which means that q corresponds to the prefix representation of the pattern $out[q]$. The forward transition function g is defined so that $g[q_i, \alpha] = q_j$ when q_i corresponds to $\mu(P_k)[1..d[q_i]]$ and q_j corresponds to $\mu(P_k)[1..d[q_i] + 1]$ for some pattern P_k where $\alpha = \mu(P_k)[d[q_i]]$. The trie can be constructed in $O(m)$ time once the prefix representations of the patterns are given.

Fig. 6 shows an example of an Aho–Corasick automaton with three patterns $P_1 = \{23, 35, 15, 53, 47\}$, $P_2 = \{66, 71, 57, 79, 84, 93\}$, $P_3 = \{43, 51, 62, 73\}$. The automaton is constructed from the prefix representations $\mu(P_1) = (1, 2, 1, 4, 4)$, $\mu(P_2) = (1, 2, 1, 4, 5, 6)$ and $\mu(P_3) = (1, 2, 3, 4)$ regardless of the pattern characters. For example, q_5 represents the prefix $(1, 2, 1, 4)$, which matches with $\mu(P_1)$ and $\mu(P_2)$ even though $P_1[1..4]$ and $P_2[1..4]$ have different characters.

Compared to the original Aho–Corasick algorithm, we have two additional values $d[q]$ and $P[q]$ for each state q. Both of them are recorded to maintain the order-statistic tree per pattern during the construction of the failure function π. The details are described in the following sections.

5.2. Aho–Corasick failure function

The failure function π can be defined so that $\pi[q_i] = q_j$ if and only if the prefix represented by q_j (i.e. $\mu(P[q_j])[1..d[q_j]]$) is the prefix representation of the longest proper suffix of $P[q_i]$ (i.e. $\mu(P[q_i][k..d[q_i]])$ for some k). For example, for q_8 in Fig. 6 with the prefix $(1, 2, 1, 4, 5)$ of $\mu(P_2)$, $\pi[q_8] = q_4$ because $P_2[3..5]$ is the longest proper suffix of P_2 whose prefix representation $(1, 2, 3)$ is the prefix of some pattern. Here, $P[q_4] = P_3$ and $\mu(P[q_4])[1..3] = (1, 2, 3)$ which is matched with $\mu(P_2[3..5])$.

5.3. Text search

A variant of the Aho–Corasick algorithm can be designed for the multiple pattern matching of order relations as in AC-ORDER-MATCHER-MULTIPLE. Assuming that the prefix representations of all the patterns and the failure function are available, it scans the text and follows the Aho–Corasick automaton until there is no matched forward transition. Then, it follows the failure function until a successful forward transition is found. In the initial state q_0, it *never* fails to follow the forward transition because any character can be matched at the first character. Whenever it reaches one of the accepting states, it outputs the position of the text and the matched pattern.

The order-statistic tree $\mathcal{T}$ is maintained to compute each rank value adaptively. For every forward transition, $T[i]$ is inserted to $\mathcal{T}$, and for every backward transition $\pi[q_i] = q_j$, the oldest $d[q_i] - d[q_j]$ characters are deleted from $\mathcal{T}$. The rank of $T[i]$ should be calculated again for each backward transition after $\mathcal{T}$ is properly updated. For example, when AC-Order-Matcher-Multiple reaches state q_3 of Fig. 6 after reading the first three characters from the text (20, 30, 10, 15), $\mathcal{T}$ contains $\{20, 30, 10\}$, which is the prefix of the text represented by q_3. As there is no forward transition from q_3 that matches the rank 2 of the next character 15, the state is changed to q_1 by following the failure transition. The oldest $d[q_3] - d[q_1] = 2$ characters are deleted from $\mathcal{T}$ so that it contains $\{10\}$ at the next step. The state is then changed to q_2 by following the forward transition 2 and inserting 15 to $\mathcal{T}$ (which is rank 2 in $\{10, 15\}$).

```
AC-Order-Matcher-Multiple($T$, $\mathcal{P}$)
1  $n \leftarrow |T|$, $w \leftarrow |\mathcal{P}|$
2  for $i \leftarrow 1$ to $w$
3      $\mu(P_i) \leftarrow$ Compute-Prefix-Rep($P_i$)
4  $(\pi, out) \leftarrow$ Compute-AC-Failure-Function($\mathcal{P}$)
5  $\mathcal{T} \leftarrow \phi$
6  $q \leftarrow q_0$
7  for $i \leftarrow 1$ to $n$
8      OS-Insert($\mathcal{T}, T, i$)
9      $r \leftarrow$ OS-Rank($\mathcal{T}, T[i]$)
10     while $g[q, r] = fail$
11         OS-Delete($\mathcal{T}, T[i - d[q]..i - d[\pi[q]] - 1]$)
12         $q \leftarrow \pi[q]$
13         $r \leftarrow$ OS-Rank($\mathcal{T}, T[i]$)
14     $q \leftarrow g[q, r]$
15     if $out[q] \neq \phi$
16         print "pattern" $out[q]$ "occurs at position" $i$
```

The time complexity of AC-Order-Matcher-Multiple is $O(n \log m)$ (except for the preprocessing of the patterns) because it does n insertions in $\mathcal{T}$ and thus at most n deletions can take place. Checking $g[q, r]$ in line 10 takes $O(\log m)$ time as well. As each operation takes $O(\log m)$ time and there are $O(n)$ operations, the total time is $O(n \log m)$.

5.4. *Construction of Aho–Corasick failure function*

Compute-AC-Failure-Function shows the construction algorithm of the Aho–Corasick failure function. As in the original Aho–Corasick algorithm, it computes the failure function in the breadth first order of the automaton.

The main difference from the original Aho–Corasick algorithm is that we maintain multiple order-statistic trees simultaneously (one per pattern) because the rank value of a character depends on the pattern in which the rank is calculated. Let $\mathcal{T}(P_i)$ denote the order-statistic tree for the pattern P_i, and assume that a representative pattern $P[q]$ is recorded for each node q such that q is reachable by some prefix of the prefix representation of $P[q]$.

We maintain each order-statistic tree $\mathcal{T}(P[q])$ of $P[q]$ so that it contains the characters of the longest proper suffix of $P[q][1..d[q]]$ whose prefix representation is a prefix of the prefix representation of some pattern. Consider a forward transition $g[q_i, \alpha] = q_j$ such that $\pi[q_i]$ is available but $\pi[q_j]$ is to be computed. If $P[q_i] = P[q_j]$, $\mathcal{T}(P[q_i]) = \mathcal{T}(P[q_j])$ and $\mathcal{T}(P[q_j])$ already contains the characters of $P[q_j]$. It can be updated by inserting $P[q_j][d[q_j]]$ and deleting some characters from $\mathcal{T}(P[q_j])$. However, if $P[q_i] \neq P[q_j]$, we should initialize $\mathcal{T}(P[q_j])$ by inserting characters of the suffix of $P[q_j][1..d[q_j] - 1]$ so that it has the same number of characters as $\mathcal{T}(P[q_i])$. $\mathcal{T}(P[q_j])$ can then be updated as in the other case. In both cases, the rank of $P[q_j][d[q_j]]$ in $\mathcal{T}(P[q_j])$ is computed again to find the correct forward transition starting from $\pi[q_i]$.

For instance, let's consider node q_5 in Fig. 6. $P[q_5] = P_1$ and $\mathcal{T}(P_1)$ has $\{15, 53\}$ since $d[\pi[q_5]] = 2$. When $\pi[q_7]$ is computed, it inserts 47 to $\mathcal{T}(P_1)$, which has rank 2 in $\{15, 53, 47\}$, and tries to follow the rank 2 from $\pi[q_5] = q_2$. As there is no forward transition of q_2 with label 2, it follows the failure function $\pi[q_2] = q_1$ and deletes 15 from $\mathcal{T}(P_1)$. Similarly, there is no forward transition of the rank 1 of 47 in $\{53, 47\}$ from q_1, it reaches q_0. Finally, it follows the forward transition of q_1 by the rank 1 of 47 in $\{47\}$ and $\pi[q_7] = q_1$. On the other hand, when $\pi[q_8]$ is computed, $P[q_8] = P_2$ and $P[q_8] \neq P[q_7]$. The last $d[\pi[q_5]]$ characters of $P_2[1..d[q_5]]$ are inserted to $\mathcal{T}(P_2)$, and $\mathcal{T}(P_2)$ becomes $\{57, 79\}$. Then, the next character 84 of $P[q_8]$ is inserted to $\mathcal{T}(P_2)$, which is rank 3 of $\{57, 79, 84\}$, and it follows the rank 3 from q_2, which results in $\pi[q_8] = q_4$.

```
Compute-AC-Failure-Function($T$, $\mathcal{P}$)
 1  $\pi[q_0] \leftarrow q_0$
 2  **for each** $P_i \in \mathcal{P}$
 3      $\mathcal{T}(P_i) \leftarrow \phi$
 4      $out[q_i] \leftarrow P_i$ for the last state $q_i$ of $P_i$
 5  **for each** $q_i \in Q$ (BFS order)
 6      **for each** $\alpha$ such that $g[q_i, \alpha] \neq$ fail
 7          $q_j \leftarrow g[q_i, \alpha]$, $c \leftarrow P[q_j][d[q_j]]$
 8          **if** $P[q_i] \neq P[q_j]$
 9              **for** $k \leftarrow 1$ **to** $d[\pi[q_i]]$
10                  OS-Insert($\mathcal{T}(P[q_j]), P[q_j], d[q_i] - d[\pi[q_i]] + k$)
11          OS-Insert($\mathcal{T}(P[q_j]), P[q_j], d[q_j]$)
12          $r \leftarrow$ OS-Rank($\mathcal{T}(P[q_j]), c$)
13          $q_p \leftarrow q_i$, $q_h \leftarrow \pi[q_i]$
14          **while** $g[q_h, r] = fail$
15              OS-Delete($\mathcal{T}(P[q_j]), P[q_j][i - d[q_p] + 1..i - d[q_h]]$)
16              $r \leftarrow$ OS-Rank($\mathcal{T}(P[q_j]), c$)
17              $q_p \leftarrow q_h$, $q_h \leftarrow \pi[q_h]$
18          $\pi[q_j] \leftarrow g[q_h, r]$
19          **if** $out[q_j] = \phi$
20              $out[q_j] \leftarrow out[\pi[q_j]]$
21  **return** $(\pi, out)$
```

The time complexity of Compute-AC-Failure-Function can be analyzed as follows. The number of all forward transitions is at most m and there are at most m insert operations on $\mathcal{T}$ because each character of a pattern can be inserted either in line 10 or in line 11, but not in both. The number of deleted characters cannot exceed the number of inserted characters and the number of rank computations is also bounded by m. As the number of operations is bounded by $O(m)$ and each takes $O(\log m)$, the total time complexity is $O(m \log m)$.

5.5. Correctness and time complexity

The correctness of our algorithm can be easily derived from the correctness of the original Aho–Corasick algorithm and our version for the single pattern case.

The total time complexity is $O(n \log m)$: $O(m \log m)$ to compute the prefix representation and failure function, and $O(n \log m)$ for text search. Compared with $O(n \log |\Sigma|)$ time of the exact pattern matching where Σ is the alphabet, our algorithm has a comparable time complexity since $|\Sigma|$ for numeric strings can be as large as m.

Note that we cannot remove $\log m$ factor from the above time complexity as in the single pattern case since $O(\log m)$ time has to be spent at each state to find the forward transition to follow even with the nearest neighbor representation.

6. Conclusion

We have introduced *order-preserving matching* and defined the *prefix representation* and the *nearest neighbor representation* of order relations of a numeric string. By using these representations, we developed an $O(n + m \log m)$ algorithm for single pattern matching and an $O(n \log m)$ algorithm for multiple pattern matching. We believe that our work opens a new direction in string matching over numeric strings, with many practical applications.

Acknowledgements

This work is partially the result of collaboration at the IPDF 2012 Workshop on Algorithmics on Massive Data, held in Sunshine Coast, August 23–27, 2012, supported by the University of Sydney International Program Development Funding. The work of Jinil Kim and Kunsoo Park was supported by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (2011-0029924). Peter Eades was supported by a Discovery grant from the Australian Research Council, and IPDF grant from the University of Sydney. Rudolf Fleischer acknowledges support by the National Natural Science Foundation of China (No. 60973026), the Shanghai Leading Academic Discipline Project (project number B114), the Shanghai Committee of Science and Technology of China (Nos. 08DZ2271800 and 09DZ2272800), and the Research Council (TRC) of the Sultanate of Oman. Seok-Hee Hong was supported by a Future Fellowship from the Australian Research Council. Simon J. Puglisi was supported by the Newton Fellowship Alumni Scheme and by Academy of Finland grant 250345 (CoECGR). Takeshi Tokuyama was supported by JSPS KAKENHI Grant Numbers 22300001 and 24650001.

References

[1] A.V. Aho, Algorithms for finding patterns in strings, in: Handbook of Theoretical Computer Science, vol. A: Algorithms and Complexity (A), 1990, pp. 255–300.

[2] A. Amir, Y. Aumann, P. Indyk, A. Levy, E. Porat, Efficient computations of l_1 and l_{INFINITY} rearrangement distances, Theor. Comput. Sci. 410 (43) (2009) 4382–4390.

[3] A. Amir, Y. Aumann, G.M. Landau, M. Lewenstein, N. Lewenstein, Pattern matching with swaps, J. Algorithms 37 (2) (2000) 247–266.

[4] A. Amir, Y. Aumann, M. Lewenstein, E. Porat, Function matching, SIAM J. Comput. 35 (5) (2006) 1007–1022.

[5] A. Amir, R. Cole, R. Hariharan, M. Lewenstein, E. Porat, Overlap matching, Inf. Comput. 181 (1) (2003) 57–74.

[6] A. Amir, M. Farach, Efficient 2-dimensional approximate matching of half-rectangular figures, Inf. Comput. 118 (1) (1995) 1–11.

[7] A. Amir, M. Farach, S. Muthukrishnan, Alphabet dependence in parameterized matching, Inf. Process. Lett. 49 (3) (1994) 111–115.

[8] A. Amir, O. Lipsky, E. Porat, J. Umanski, Approximate matching in the L_1 metric, in: CPM, 2005, pp. 91–103.

[9] A. Amir, I. Nor, Generalized function matching, J. Discrete Algorithms 5 (3) (2007) 514–523.

[10] B.S. Baker, A theory of parameterized pattern matching: algorithms and applications, in: STOC, 1993, pp. 71–80.

[11] R.S. Boyer, J.S. Moore, A fast string searching algorithm, Commun. ACM 20 (10) (Oct. 1977) 762–772.

[12] E. Cambouropoulos, M. Crochemore, C.S. Iliopoulos, L. Mouchard, Y.J. Pinzon, Algorithms for computing approximate repetitions in musical sequences, Int. J. Comput. Math. 79 (11) (2002) 1135–1148.

[13] D. Cantone, S. Cristofaro, S. Faro, An efficient algorithm for alpha-approximate matching with *delta*-bounded gaps in musical sequences, in: WEA, 2005, pp. 428–439.

[14] D. Cantone, S. Cristofaro, S. Faro, On tuning the (δ, α)-sequential-sampling algorithm for δ-approximate matching with alpha-bounded gaps in musical sequences, in: ISMIR, 2005, pp. 454–459.

[15] P. Clifford, R. Clifford, C.S. Iliopoulos, Faster algorithms for (δ, γ)-matching and related problems, in: CPM, 2005, pp. 68–78.

[16] R. Clifford, C.S. Iliopoulos, Approximate string matching for music analysis, Soft Comput. 8 (9) (2004) 597–603.

[17] R. Cole, C.S. Iliopoulos, T. Lecroq, W. Plandowski, W. Rytter, On special families of morphisms related to δ-matching and don't care symbols, Inf. Process. Lett. 85 (5) (2003) 227–233.

[18] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 3rd edition, The MIT Press, 2009.

[19] M. Crochemore, C.S. Iliopoulos, T. Lecroq, W. Plandowski, W. Rytter, Three heuristics for δ-matching: δ-BM algorithms, in: CPM, 2002, pp. 178–189.

[20] M. Crochemore, W. Rytter, Jewels of Stringology: Text Algorithms, World Scientific, 2003.

[21] K. Fredriksson, S. Grabowski, Efficient algorithms for (δ, γ, α) and $(\delta, k_{\mathrm{delta}}, \alpha)$-matching, Int. J. Found. Comput. Sci. 19 (1) (2008) 163–183.

[22] R.N. Horspool, Practical fast searching in strings, Softw. Pract. Exp. 10 (1980) 501–506.

[23] I. Lee, R. Clifford, S.-R. Kim, Algorithms on extended (δ, γ)-matching, in: ICCSA, vol. 3, 2006, pp. 1137–1142.

[24] I. Lee, J. Mendivelso, Y.J. Pinzon, Delta–gamma-parameterized matching, in: SPIRE, 2008, pp. 236–248.

[25] O. Lipsky, E. Porat, Approximate matching in the L_{infinity} metric, Inf. Process. Lett. 105 (4) (2008) 138–140.

[26] J. Mendivelso, I. Lee, Y.J. Pinzon, Approximate function matching under δ- and γ-distances, in: SPIRE, 2012, pp. 348–359.

[27] S. Muthukrishnan, New results and open problems related to non-standard stringology, in: CPM, 1995, pp. 298–317.

[28] E. Porat, K. Efremenko, Approximating general metric distances between a pattern and a text, in: SODA, 2008, pp. 419–427.

[29] D.M. Sunday, A very fast substring search algorithm, Commun. ACM 33 (8) (Aug. 1990) 132–142.

[30] P. van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, Inf. Process. Lett. 6 (3) (1977) 80–82.

[31] D.E. Willard, Log-logarithmic worst-case range queries are possible in space $\Theta(N)$, Inf. Process. Lett. 17 (2) (1983) 81–84.