공학박사학위논문

수치 문자열의 순서를 보존하는 매칭 기법

Order-Preserving Matching in Numeric Strings

2014년 2월

서울대학교 대학원

전기 · 컴퓨터공학부

김 진 일

수치 문자열의 순서를 보존하는 매칭 기법

Order-Preserving Matching in Numeric Strings

지도교수 박 근 수

이 논문을 공학박사 학위논문으로 제출함
2013년 11월

서울대학교 대학원
전기 · 컴퓨터 공학부
김 진 일

김진일의 공학박사 학위논문을 인준함
2013년 12월

Abstract

String matching is a fundamental problem in computer science and has been extensively studied. Sometimes a string consists of numeric values instead of alphabet characters, and we are interested in some *trends* in the text rather than specific patterns. We introduce a new string matching problem called *order-preserving matching* on numeric strings, where a pattern matches a text substring of the same length if the relative orders in the substring coincide with those of the pattern. Order-preserving matching is applicable to many scenarios such as stock price analysis and musical melody matching.

In this thesis, we define order-preserving matching in numeric strings, and present various representations of order relations and efficient algorithms of order-preserving matching with those representations. For single pattern matching, we give an $O(n \log m)$ time algorithm with the *prefix representation* based on the KMP algorithm, and optimize it further to obtain $O(n + m \log m)$ time with the *nearest neighbor representation*, where n and m are the lengths of the text and the pattern, respectively. For multiple pattern matching, we present an $O((n+m) \log m)$ time algorithm with the *prefix representation* based on the Aho-Corasick algorithm, where n is the text length and m is the sum of the lengths of the patterns. Our algorithms are presented in binary order relations first, and then extended to ternary order relations. With our extensions, the time complexities in binary order relations can be achieved in ternary order relations as well.

Keywords: order-preserving matching, order relation, pattern matching, numeric string, KMP algorithm, Aho-Corasick algorithm

Student Number: 2007-30219

Contents

List of Figures

List of Tables

Chapter 1

Introduction

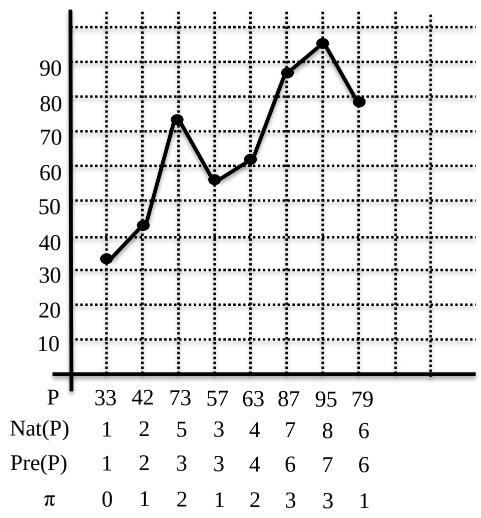
1.1 Background

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 matches a text substring if the relative orders of the substring coincide with those of the pattern. For example, in Fig. 1.1, pattern $P = (33, 42, 73, 57, 63, 87, 95, 79)$ matches the text substring (21, 24, 50, 29, 36, 73, 85, 63) since it has the same relative orders as those of 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.

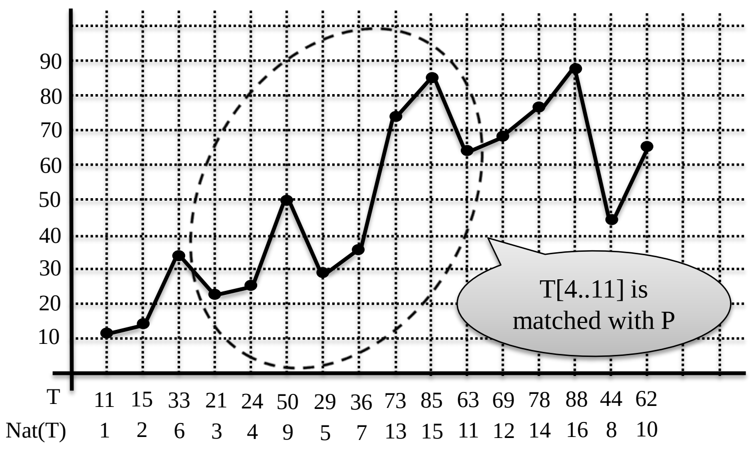
Solving order-preserving matching is closely related to *representations of order relations* of a numeric string. If we replace each character in a numeric string by its rank in the string, then we can obtain a (natural) representation of order relations. But this natural representation is not amenable 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.

An order relation between two characters is a ternary relation ($>$, $<$, $=$) rather than a binary relation ($>$, $<$). The number of possible order relations (weak orderings) on a sequence of n elements is known as *Ordered Bell Number*, which is approximated as $\frac{n!}{2(\log 2)^{n+1}} \approx \frac{(1.443)^{n+1} \cdot n!}{2}$ [26, 11]. It is exponentially larger than $n!$, the number of possible order relations in a binary relation on n distinct numbers. We extend the representations of order relations to ternary order relations, and prove the equivalence of those representations. With our extensions, the time complexities of order-preserving matching in binary order relations can be achieved in ternary order relations as well.



P	33	42	73	57	63	87	95	79
Nat(P)	1	2	5	3	4	7	8	6
Pre(P)	1	2	3	3	4	6	7	6
π	0	1	2	1	2	3	3	1

(a) Example of a pattern



T	11	15	33	21	24	50	29	36	73	85	63	69	78	88	44	62
Nat(T)	1	2	6	3	4	9	5	7	13	15	11	12	14	16	8	10

(b) Example of a text

Figure 1.1: Example of a pattern and a text

1.2 Contribution

In this thesis, we define a new class of string matching problem, called *order-preserving matching*, and present efficient algorithms for single and multiple pattern matching.

1. We present order-preserving matching algorithms in binary order relations assuming that all the characters are *distinct*. For single pattern case, we propose an $O(n \log m)$ algorithm based on the Knuth-Morris-Pratt (KMP) algorithm [20, 23], and optimize it further to obtain $O(n+m\log m)$ time. For multiple pattern case, we present an $O((n+m)\log m)$ algorithm based on the Aho-Corasick algorithm [1].

2. We extend the representations of order relations to ternary order relations. We prove the equivalence of those representations, and generalize the KMP-based algorithm to adopt any representation of order relations. With the *extended prefix representation*, order-preserving matching can be done in $O(n \log m)$ time, and the representation of order relations takes $(\log m + 1)$ bits per character. With the *nearest neighbor representation*, the matching can be done in $O(n+m\log m)$, but the representation takes $(2\log m)$ bits per character.

1.3 Related Work

The study of order-preserving matching was introduced by Kubica et al. [32] and Kim et al. [28] where Kubica et al. [32] defined order relations by order isomorphism of two strings, while Kim et al. [28] defined them explicitly by the sequence of rank values (which they called the *natural representation*). In both papers, an $O(n + m\log m)$ time algorithm is proposed for single pattern matching where n is the text length and m is the pattern length.

Recently, some new results of order-preserving matching are of interest. Crochemore et al. [21] defined an order-preserving suffix tree and presented the construction algorithm in $O(\frac{n\log n}{\log\log n})$ time. Cho et al. [16] proposed an order-preserving matching using the bad character heuristics on q-grams, which is practically faster than the KMP-based one. Gawrychowski et al. [25] considered an approximate order-preserving matching problem with k-mismatches and presented an $O(n(\log\log n + k\log\log k))$ time algorithm.

Norm matching and (δ, γ)-matching have been studied to search for similar patterns of numeric strings. In norm matching [9, 35, 3, 39], 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 [13, 22, 18, 17, 33, 34, 37], 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 [19], transposition-invariant [33] and gaps [14, 15, 24].

On the other hand, some generalized matching problems such as parameterized matching [12, 8], less than matching [7], swapped matching [4, 38], overlap matching [6], and function matching [5, 10] 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 those work addresses the *order relations*, which we focus on in this paper.

1.4 Organization

The rest of the thesis is organized as follows. In Chapter 2, we define *order-preserving matching* and present efficient algorithms for single pattern matching in binary order relations. In Chapter 3, we consider multiple pattern matching and present an efficient algorithm. In Chapter 4, we extend the representations of order relations to ternary relations and prove the equivalence of those representations. Finally, we conclude in Chapter 5

Chapter 2

Order-Preserving Pattern Matching

In this chapter, we define the prefix representation and the nearest neighbor representation of order relations, and present an $O(n \log m)$ algorithm and an $O(n + m \log m)$ algorithm from those representations.

2.1 Preliminaries

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. For a string $x \in \Sigma^*$, let $|x|$ denote the length of x. A string x is described by a sequence of characters ($x[1]$, $x[2]$, ..., $x[|x|]$). Let a substring $x[i..j]$ be ($x[i]$, $x[i+1]$, ..., $x[j]$) and a prefix x_i be $x[1..i]$. For a character $c \in \Sigma$, let $rank_x(c) = 1 + |\{i : x[i] < c \text{ for } 1 \leq i \leq |x|\}|$.

2.1.1 Definitions of Order Relations

Given two numeric strings, the notion of order-preserving matching can be defined by either the order isomorphism [32], or the natural representation [28].

Definition 2.1.1 (Order Isomorphism [32]) *For two strings x and y of length n, x and y are order-isomorphic if $\forall i, j \in [1..n]$, $x[i] \leq x[j] \Leftrightarrow y[i] \leq y[j]$.*

Order isomorphism *implicitly* deals with ternary order relations because each of ternary order relations ($>$, $<$, $=$) can be checked by variants of the proposition in Definition 2.1.1 (changing i and j, taking the contrapositive, or both). For example, if $x[i] > x[j]$, then $y[i] > y[j]$ by the contrapositive of $x[i] \leq x[j] \Leftarrow y[i] \leq y[j]$. If $x[i] = x[j]$, then $y[i] = y[j]$ by $x[i] \leq x[j] \Rightarrow y[i] \leq y[j]$ and $x[j] \leq x[i] \Rightarrow y[j] \leq y[i]$.

The definition of order isomorphism looks simple, but it is not easy to understand and somewhat complicated to handle in practice. The number of the order relations involved in checking order isomorphism of two strings of length n is $O(n^2)$, hence it has an inherent quadratic term if the definition is used directly for order-preserving matching. Moreover, the ternary order relations are not explicitly stated but implied by the proposition in Definition 2.1.1. In fact, there was an incorrect proof of order isomorphism in [32] due to a missing case, which was fixed later in [16].

Alternatively, the *natural representation* can be used for comparing order relations of two strings [28].

Definition 2.1.2 (Natural Representation [28]) *For a string x of length n, the natural representation of the order relations is defined as $Nat(x) = (rank_x(x[1]), rank_x(x[2]), ..., rank_x(x[n]))$.*

For example, for x = (30, 10, 50, 20, 30, 20, 20), and y = (35, 15, 55, 25, 35, 25, 35), the natural representations are $Nat(x)$ = (5, 1, 7, 2, 5, 2, 2) and $Nat(y)$ = (4, 1, 7, 2, 4, 2, 4), respectively.

In the natural representation, ternary order relations are *explicitly* stated in terms of ranks. For example, $x[i] > x[j]$ if and only if $Nat(x)[i] > Nat(x)[j]$, and $x[i] = x[j]$ if and only if $Nat(x)[i] = Nat(x)[j]$. That is, the order relations of two strings coincide if and only if $Nat(x) = Nat(y)$. The comparison of two natural representations takes $O(n)$ time if the natural representations are given.

These two definitions are equivalent because the natural representations of two strings are identical if and only if they are order-isomorphic. We adopt the natural representation throughout this paper because the definition itself and subsequent analysis are more intuitive.

2.1.2 Number of Representations

The number of the natural representations on n characters coincides with that of weak orderings on a sequence of n elements which is known as the *ordered Bell number* [29, 26, 27]. The ordered Bell number is the solution of the recurrence $f(n) = 1 + \sum_{j=1}^{n-1} \binom{n}{j} f(n-j)$ [27], and it is approximated by $f(n) \approx \frac{n!}{2(\log 2)^{n+1}} \approx \frac{(1.443)^{n+1} \cdot n!}{2}$ [11] for sufficiently large n.

The recurrence of the ordered Bell number has a simple interpretation: For a sequence of length n allowing ties, let $j \in [1..n]$ be the number of occurrences of the largest character in the sequence. All of such sequences can be generated by generating a sequence of $n - j$ characters less than the largest character first, and then inserting j occurrences of the largest character between the $n - j$ characters. The number of such insertions is the number of combinations choosing j positions from $n-j+1$ positions with repetitions [31], which is equal to $\binom{n}{j}$. The recurrence is derived by summing $\binom{n}{j} f(n-j)$ cases for $j \in [1..n-1]$ and adding the extra case for $j = n$ when all characters are the same.

2.1.3 Problem Formulation

Order-preserving matching can be defined in terms of natural representations.

Definition 2.1.3 (Order-Preserving Matching [28]) *Given a text $T[1..n] \in \Sigma^*$ and a pattern $P[1..m] \in \Sigma^*$, P matches T at position i if $Nat(P) = Nat(T[i-m+1..i])$. Order-preserving matching is the problem of finding all positions of T matched with P.*

For example, let us 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 1.1. The natural representation of P is $\sigma(P) = (1, 2, 5, 3, 4, 7, 8, 6)$, which matches $T[4..11] = (21, 24, 50, 29, 36, 73, 85, 63)$ 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 [20, 23] 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 [20], 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.

2.2 $O(n \log m)$ Algorithm

In this section, we define the *prefix representation*, and present an $O(n \log m)$ algorithm for single pattern matching based on the KMP algorithm [30, 20]. We consider *binary order relations* between two characters assuming that all the characters in a string are *distinct*. The extensions to ternary order relations are covered in Section 4.3.

2.2.1 Prefix Representation

The *prefix representation* [28] can be defined as a sequence of rank values of characters in prefixes by Definition 2.2.1.

Definition 2.2.1 (Prefix Representation [28]) *For a string x, the prefix representation of the order relations is defined as $Pre(x) = (rank_{x_1}(x[1]), rank_{x_2}(x[2]), ..., rank_{x_{|x|}}(x[|x|]))$.*

For example, the prefix representation of P in Fig 1.1 is $Pre(x) = (1, 2, 3, 3, 4, 6, 7, 6)$.

An advantage of the prefix representation is that $Pre(x)[i]$ 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 [20] containing characters of $x[1..i-1]$, $Pre(x)[i]$ 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 $x[1..i]$, $Pre(x[1..i+1])[i+1]$ can be computed if $x[i+1]$ is inserted to $\mathcal{T}$, and $Pre(x[2..i])[i-1]$ 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 in binary order relations. 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 $1 \cdot 2 \cdot ...n = n!$ 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 2.1. 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.

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

Table 2.1: List of functions in the order-statistic tree $\mathcal{T}$

```
COMPUTE-PREFIX-REP($P$)
1  $m \leftarrow |P|$
2  $\mathcal{T} \leftarrow \phi$
3  OS-INSERT($\mathcal{T}, P, 1$)
4  $Pre(P)[1] \leftarrow 1$
5  for $k \leftarrow 2$ to $m$
6      OS-INSERT($\mathcal{T}, P, k$)
7      $Pre(P)[k] \leftarrow$ OS-RANK($\mathcal{T}, P[k]$)
8  return $Pre(P)$
```

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.

2.2.2 KMP Failure Function

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

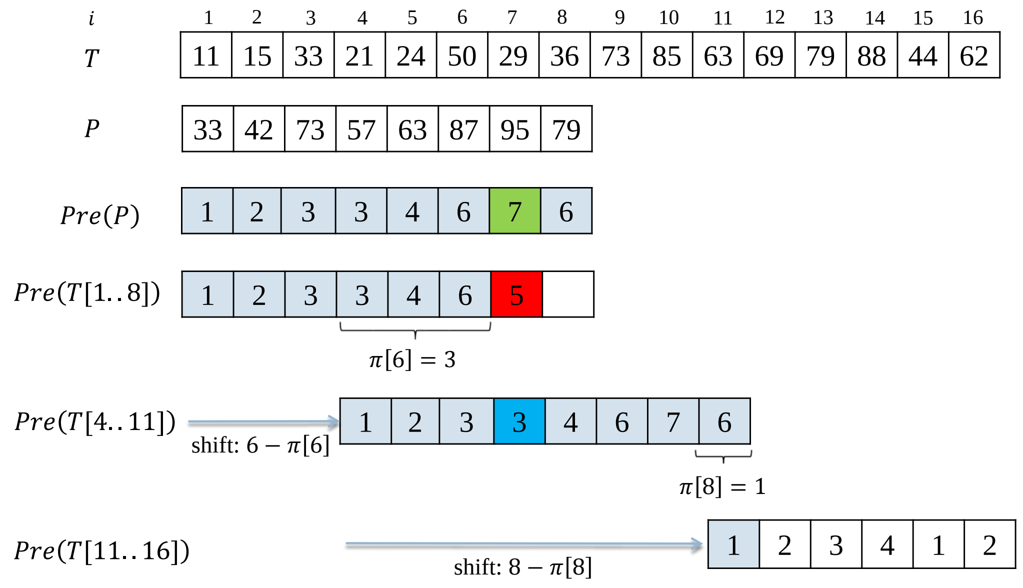


Figure 2.1: Example of text search with the prefix representation

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

Intuitively, π means that the longest proper prefix $Pre(P[1..k])$ of P matches $Pre(P[q-k+1..q])$ which is the prefix representation of the suffix of $P[1..q]$ with length k. For example, the failure function of P in Fig 1.1 is $\pi[1..m] = (0, 1, 2, 1, 2, 3, 3, 1)$. As shown in Fig 2.1, $\pi[6] = 3$ implies that the longest prefix of $Pre(P[1..8])$ that matches the prefix representation of any suffix of $P[1..6] = (33, 42, 63, 57, 63, 87)$ is $Pre(P[1..\pi[6]]) = (1, 2, 3)$.

The construction algorithm of π will be given in Section 2.2.4.

2.2.3 Text Search

The failure function π can accelerate order-preserving matching by filtering mismatched positions as in the KMP algorithm. Let us assume that $Pre(P)[1..q]$ matches $Pre(T[i-q..i-1])[1..q]$ but a mismatch is found between $Pre(P)[q+1]$

and $Pre(T[i-q..i])[q+1]$. $\pi[q]$ means that $Pre(P)[1..\pi[q]]$ is already matched with $Pre(T[i-\pi[q]..i-1])[1..\pi[q]]$ and matching can be continued at $P[\pi[q]+1]$ comparing $Pre(P)[\pi[q]+1]$ with $Pre(T[i-\pi[q]..i])[\pi[q]+1]$. Since $P[1..\pi[q]]$ is the longest prefix whose order matches 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.

An example of text search in Fig 2.1 shows that how π can filter mismatched positions. When $Pre(P)[1..6]$ matches $Pre(T[1..6])$ but $Pre(P)[7]$ is different from $Pre(T[1..7])[7]$, we can skip the positions from 1 to 3 of P and continue by comparing $Pre(P)[4]$ with $Pre(T[4..7])[4]$. At this time, $Pre(P)$ matches $Pre(T[4..11])$ at position 11, and the matched position is shifted again by $8-\pi[8]$ looking for the next matched position.

KMP-Order-Matcher describes the order-preserving matching algorithm assuming that $Pre(P)$ 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 $Pre(P)[1..q]$ matches $Pre(T)[i-q..i-1]$. 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., $Pre(P)[1..q+1] = Pre(T)[i-q..i]$, 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  $Pre(P) \leftarrow$ Compute-Prefix-Rep($P$)
 3  $\pi \leftarrow$ KMP-Compute-Failure-Function($P, Pre(P)$)
 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 $r \neq Pre(P)[q+1]$
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 $Pre(T[i-q..i])[q+1]$ of $T[i]$ is computed using the order-statistic tree $\mathcal{T}$. If $Pre(T[i-q..i])[q+1]$ does not match $Pre(P)[q+1]$, 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 $r = Pre(P)[q+1]$ before increasing q by 1 in line 7 (cf. [20, 23]) because it should be satisfied automatically. From the condition of the while loop in line 5, $q = 0$ or $r = Pre(P)[q+1]$ in line 7, and if $q = 0$, $Pre(P)[1] = 1$ for any pattern and it matches any text of length 1.

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.

2.2.4 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 $Pre(P[1..\pi[q-1]])$ and $Pre(P[q-\pi[q-1]..q-1])$. If $Pre(P[1..\pi[q-1]+1])[\pi[q-1]+1] = Pre(P[q-\pi[q-1]..q])[\pi[q-1]+1]$, 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 2.2 shows an example of computing the failure function of P in Fig. 1.1 in which $\pi[7]$ is being computed. Starting from $q = \pi[6] = 3$, KMP-Order-Matcher tries to match $Pre(P[4..8])[4]$ with $Pre(P)[4]$ but it fails. Then, q is decreased to $q = \pi[3] = 2$ and it tries to match $Pre(P[5..8])[3]$ with $Pre(P)[3]$ and it succeeds. $\pi[7]$ is assigned to $\pi[3] + 1$, and the next iteration is started with $q = \pi[7]$.

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.

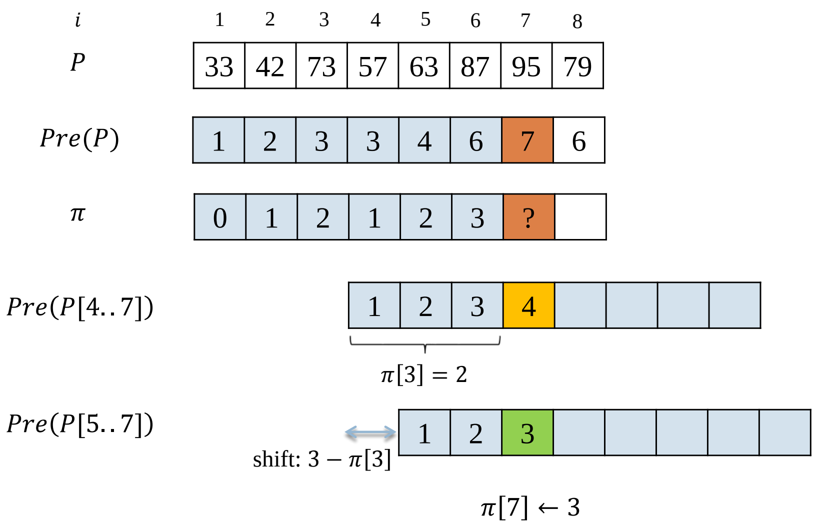


Figure 2.2: Example of computing the failure function π

```
KMP-COMPUTE-FAILURE-FUNCTION($P, Pre(P)$)
 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 $r \neq Pre(P)[k+1]$
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 $\pi$
```

2.2.5 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 2.2.3 and 2.2.4, 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 2.3.

2.3 $O(n + m \log m)$ Algorithm

In this section, we define the *nearest neighbor representation*, and present an $O(n + m \log m)$ algorithm for single pattern matching based on the KMP algorithm [30]. We consider *binary order relations* as in Section 2.2. The extensions to ternary order relations are covered in Section 4.3.

2.3.1 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 matches $Pre(P)[q + 1]$ when we know that $Pre(P)[1..q]$ matches $Pre(T[i - q + 1..i])$. If we can do it directly without computing $Pre(P)[q + 1]$, 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.

i	1	2	3	4	5	6	7	8
P	33	42	73	57	63	87	95	79
$Nat(P)$	1	2	5	3	4	7	8	6
$Pre(P)$	1	2	3	3	4	6	7	6
$NN(P)$	$\begin{pmatrix} -\infty \\ \infty \end{pmatrix}$	$\begin{pmatrix} 1 \\ \infty \end{pmatrix}$	$\begin{pmatrix} 2 \\ \infty \end{pmatrix}$	$\begin{pmatrix} 2 \\ 3 \end{pmatrix}$	$\begin{pmatrix} 4 \\ 3 \end{pmatrix}$	$\begin{pmatrix} 3 \\ \infty \end{pmatrix}$	$\begin{pmatrix} 6 \\ \infty \end{pmatrix}$	$\begin{pmatrix} 3 \\ 6 \end{pmatrix}$
π	0	1	2	1	2	3	3	1

Table 2.2: Example of the nearest neighbor representation

The *nearest neighbor representation* of the order relations can be defined as follows. For a string x, let $LMax_x[i]$ be the index of the largest character of x_{i-1} less than $x[i]$, and $LMin_x[i]$ be the index of the smallest character of x_{i-1} greater than $x[i]$. $NN(x)[1..|x|]$ be the nearest neighbor representations of x where $NN(x)[i] = \begin{pmatrix} LMax_x[i] \\ LMin_x[i] \end{pmatrix}$. Let $LMax_x[i] = -\infty$ if there is no character less than $x[i]$ in x_{i-1} and let $LMin_x[i] = \infty$ 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 matches the corresponding pattern character in constant time without computing its rank. Fig 2.2 shows the nearest neighbor representation of the order relations of P in Fig 1.1. Suppose that $Pre(P)[1..i-1] = Pre(T[1..i-1])$ for $1 \leq i \leq m$. If $T[LMax_P[i]] < T[i] < T[LMin_P[i]]$, then $Pre(P[1..i]) = Pre(T[1..i])$. For example, $Pre(T[1])[1]$ must be matched with $Pre(P)[1]$ since $T[LMax_P[1]] < c < T[LMin_P[1]]$ 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, $Pre(P)[2] = 2$ and $T[2]$ should be larger than $T[1]$ to have $Pre(T[1..2])[2] = 2$, which is represented by $LMax_P[1] = 1$ and $LMin_P[1] = \infty$. In this way, for each character, we can decide whether the order of $T[i]$ in $Pre(T[1..i])$ matches that of $P[i]$ in $Pre(P[1..i])$ by checking $T[LMax_P[i]] < T[i] < T[LMin_P[i]]$.

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  $(LMax_P[1], LMin_P[1]) \leftarrow (-\infty, \infty)$
5  for $k \leftarrow 2$ to $m$
6       OS-INSERT($\mathcal{T}, P, k$)
7       $LMax_P[k] \leftarrow$ OS-FIND-PREV-INDEX($\mathcal{T}, P[k]$)
8       $LMin_P[k] \leftarrow$ OS-FIND-NEXT-INDEX($\mathcal{T}, P[k]$)
9  return $(LMax_P, LMin_P)$
```

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.

2.3.2 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 $Pre(P)[q+1] = Pre(T[i-q..i])[q+1]$ by comparing $T[i]$ with the characters in $T[i-q..i]$ whose indexes correspond to $LMax_P[q+1]$ and $LMin_P[q+1]$ in P. Specifically, if $T[i-q+LMax_P[q+1]-1] < T[i] <$

$T[i-q+LMin_P[q+1]-1]$, then $Pre(P)[q+1] = Pre(T[i-q..i])[q+1]$ 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 2.1. When $Pre(P)[1..6]$ matches $Pre(T[1..6])$, we can check if $Pre(T[1..7])[7]$ matches $Pre(P)[7]$ by checking if $T[7-6+LMax_P[7]-1] < T[7] < T[7-6+LMin_P[7]-1]$, 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 $Pre(P)[7]$, thus $Pre(T[1..7])$ cannot be matched with $Pre(P)[1..7]$.

An example in Fig 2.3 shows that how the nearest neighbor representation is used in the text search. Suppose that pattern P and text T are given as before, and the order relations of $P[1..6]$ and $T[1..6]$ are matched. For position 7, $P[7]$ is 95, and the largest character less than 95 is 87 whose index is 6. So $LMax_P[7]$ is 6. As there is no character greater than 95 in $P[1..6]$, $LMin_P[7]$ is ∞. To match the order of $T[7]$, it is sufficient that it is more than $T[6]$, but 29 is less than 50. The pattern is shifted by $6 - \pi[6]$, and the next match is tried between $P[4]$ and $T[7]$. At this time $P[4]$ is 57, and the nearest neighbors of 57 are 42 and 73, thus the nearest neighbor representation of $P[4]$ is 2 and 3. For the text, the matched substring is $T[4..6]$ and $T[7]$ is the 4th character in the substring. 29 is between 24 and 50, which are the second and the third positions of the matched substring, and thus the order of $T[7]$ is matched.

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  $(LMax_P, LMin_P) \leftarrow$ COMPUTE-NEAREST-NEIGHBOR-REP($P$)
 3  $\pi \leftarrow$ KMP-COMPUTE-FAILURE-FUNCTION2($P, LMax_P, LMin_P$)
 4  $q \leftarrow 0$
 5  for $i \leftarrow 1$ to $n$
 6      $(j_1, j_2) \leftarrow (LMax_P[q+1], LMin_P[q+1])$
 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          $(j_1, j_2) \leftarrow (LMax_P[q+1], LMin_P[q+1])$
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.

2.3.3 Construction of KMP Failure Function

The construction of π is an extension of KMP-COMPUTE-FAILURE-FUNCTION in Section 2.2.4 where the rank function on $\mathcal{T}$ is replaced by a comparison of characters using $LMax_P$ and $LMin_P$ 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($P, LMax_P, LMin_P$)
 1  $m \leftarrow |P|$
 2  $k \leftarrow 0$
 3  $\pi[1] \leftarrow 0$
 4  for $q \leftarrow 2$ to $m$
 5      $(j_1, j_2) \leftarrow (LMax_P[k+1], LMin_P[k+1])$
 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          $(j_1, j_2) \leftarrow (LMax_P[k+1], LMin_P[k+1])$
 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.

2.3.4 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.

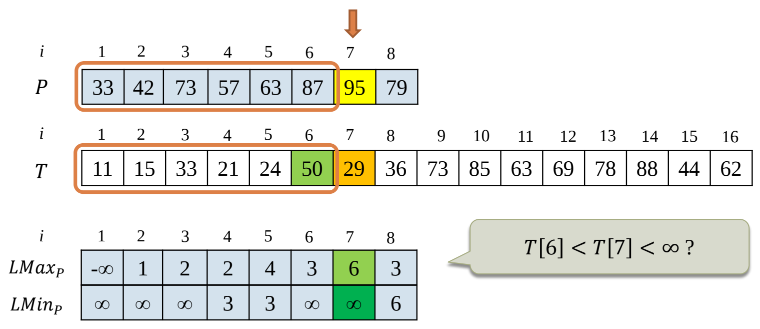
The time complexity of the overall algorithm is $O(n + m \log m)$: $O(m \log m)$ time for the computation of the nearest neighbor representation of the pattern, $O(m)$ time for the construction of π function, and $O(n)$ time for text search. $O(n + m \log m)$ is almost linear to the text length n when 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.

2.3.5 Generalized Order-Preserving Matching

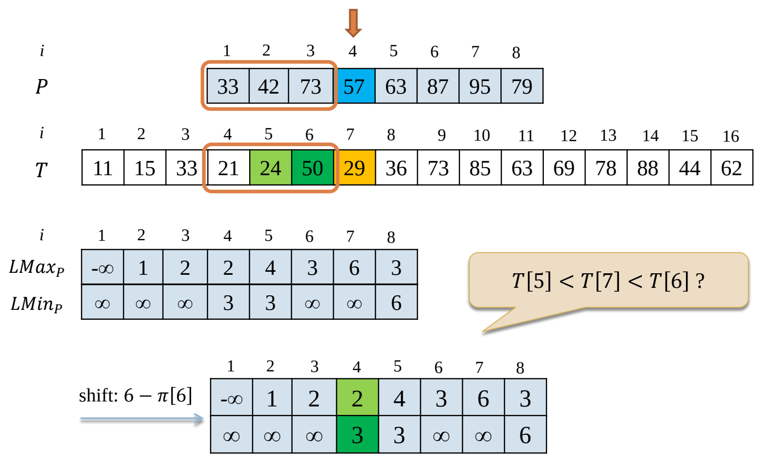
A generalization of order-preserving matching is possible with some practical applications if we consider only the orders of the last k characters for a given $k \leq m$. For example, in the stock market scenario of finding a period when a share price of a company dropped consecutively for 10 days and 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.

2.3.6 Remark on Alphabet Size

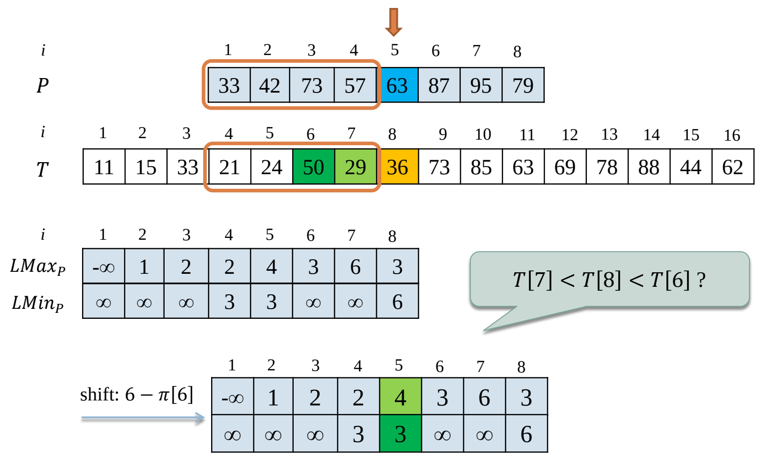
We have no restrictions on the numbers in Σ, insofar as a comparison of two numbers can be done in constant time. In the case of $\Sigma = \{1, 2, \ldots, U\}$, however, the order-statistic tree in Compute-Nearest-Neighbor-Rep can be replaced by a van Emde Boas tree [41] or y-fast trie [42] which takes $O(U)$ space and requires $O(\log \log U)$ time per operation.



(a) Step 1



(b) Step 2



(c) Step 3

Figure 2.3: Example of text search with the nearest neighbor representation

Chapter 3

Order-Preserving Multiple Pattern Matching

In this chapter, we present an $O((n+m)\log m)$ algorithm for multiple pattern matching based on the Aho-Corasick algorithm [2] in binary order relations. The extensions to ternary order relations are covered in Section 4.2.

Order-preserving matching is well-defined for multiple patterns as follows.

Definition 3.0.1 (Order-Preserving Matching for Multiple Patterns) *Given a text $T[1..n] \in \Sigma^*$ and a set of patterns $\mathcal{P} = \{P_1, P_2, ..., P_w\}$ where $P_i \in \Sigma^*$ for all $1 \leq i \leq w$, order-preserving matching for multiple patterns is the problem of finding all positions of T matched with any pattern in $\mathcal{P}$.*

3.1 $O((n+m)\log m)$ Algorithm

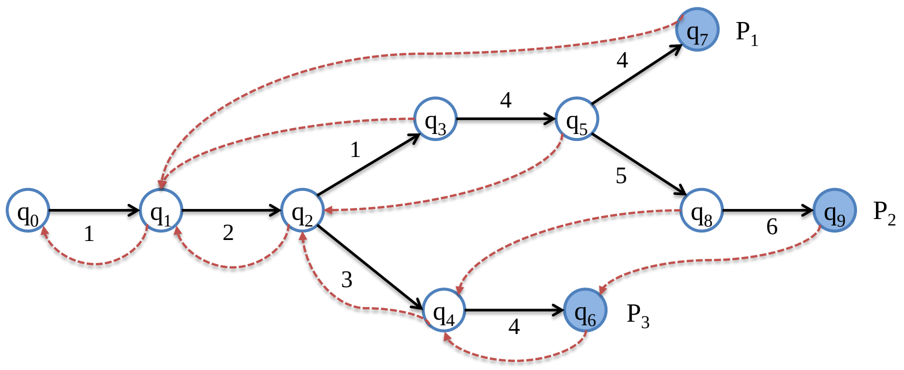
We propose a variant of the Aho-Corasick algorithm [1] for multiple pattern case whose time complexity is $O((n+m)\log m)$ where m is the sum of the lengths of the patterns.

3.1.1 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 matches 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 $Pre(P_k)[1..d[q_i]]$ and q_j corresponds to $Pre(P_k)[1..d[q_i] + 1]$ for some pattern P_k where $\alpha = Pre(P_k)[d[q_i]]$. The trie can be constructed in $O(m)$ time once the prefix representations of the patterns are given.



$P_1 = (23, 35, 15, 53, 47)$ $Pre(P_1) = (1,2,1,4,4)$

$P_2 = (66, 71, 57, 79, 84, 94)$ $Pre(P_2) = (1,2,1,4,5,6)$

$P_3 = (43, 51, 62, 73)$ $Pre(P_3) = (1,2,3,4)$

Figure 3.1: Example of an Aho-Corasick automaton

Fig. 3.1 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 $Pre(P_1) = (1, 2, 1, 4, 4)$, $Pre(P_2) = (1, 2, 1, 4, 5, 6)$ and $Pre(P_3) = (1, 2, 3, 4)$ regardless of the pattern characters. For example, q_5 represents the prefix (1, 2, 1, 4), which matches with $Pre(P_1)$ and $Pre(P_2)$ 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.

3.1.2 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. $Pre(P[q_j])[1..d[q_j]]$) is the prefix representation of the longest proper suffix of $P[q_i]$ (i.e. $Pre(P[q_i][k..d[q_i]])$ for some k). For example, for q_8 in Fig. 3.1 with the prefix (1, 2, 1, 4, 5) of $Pre(P_2)$, $\pi[q_8] = q_4$ 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 $Pre(P[q_4])[1..3] = (1, 2, 3)$ which matches $Pre(P_2[3..5])$.

3.1.3 Text Search

A variant of the Aho-Corasick algorithm can be designed for 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. 3.1 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      $Pre(P_i) \leftarrow$ 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 $O((n+m)\log m)$ (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 $O((n + m)\log m)$.

3.1.4 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 us consider node q_5 in Fig. 3.1. $P[q_5] = P_1$ 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 $O(m \log m)$.

3.1.5 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 single pattern matching.

The total time complexity is $O((n+m)\log m)$: $O(m \log m)$ to compute the prefix representation and failure function, and $O((n+m)\log m)$ 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 single pattern matching 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.

Chapter 4

Extensions to Ternary Order Relations

In this chapter, we extend the representations of order relations by Kim et al. [28] to ternary order relations, and prove the equivalence of those representations. With the *extended prefix representation*, order-preserving matching can be done in $O(n \log m)$ time, and the representation of order relations takes $(\log m + 1)$ bits per character. With the *nearest neighbor representation*, the matching can be done in $O(n + m \log m)$, but the representation takes $(2 \log m)$ bits per character. The *nearest neighbor representation* is suitable for single pattern matching while the *extended prefix representation* is space-efficient and can be useful for some order-preserving applications such as multiple pattern matching [28, 2] and the construction of suffix trees [21, 36, 40].

4.1 Preliminaries

We can consider any representation $R(\cdot)$ of order relations for order-preserving matching if $R(\cdot)$ is equivalent to $Nat(\cdot)$ by Definition 4.1.1.

Definition 4.1.1 (Equivalent Representation) *For any two representations* *of order relations $R_1(\cdot)$ and $R_2(\cdot)$, R_1 is equivalent to R_2 if $R_1(x) = R_1(y) \Leftrightarrow R_2(x) = R_2(y)$ for any two strings x, y.*

In KMP-based algorithms in Section 2.2 and 2.3, the length of matches is incrementally increased when the next character of the text matches that of the pattern. Such a match operation is formalized by the *match condition* as follows.

Definition 4.1.2 (Match Condition) *A match condition of a representation $R(\cdot)$ is a boolean function $Match(x, y, R(x), t+1)$ such that $Nat(x_{t+1}) = Nat(y_{t+1})$ holds if and only if $Nat(x_t) = Nat(y_t)$ and $Match(x, y, R(x), t+1)$ where x, y are any strings of the same length, and $t \in [1..|x|-1]$.*

4.2 Extension of Prefix Representation

The *prefix representation* in Definition 2.2.1 has an ambiguity between different strings in ternary order relations. For example, when x = (10, 30, 20), and y = (10, 20, 20), the prefix representations of both x and y are (1, 2, 2).

In this section, we define the *extended prefix representation* in ternary order relations, and prove that it is equivalent to the natural representation. The ambiguity is resolved in the extended prefix representation by adding a boolean value to mark whether each character appeared in the proper prefix or not. A match condition of the extended prefix representation is also presented to produce an $O(n \log m)$ algorithm as in binary order relations [28].

For a character $c \in \Sigma$, let $exist_x(c)$ be 1 if c exists in x, and 0 otherwise. Let $ex\text{-}rank_x(c) = (rank_x(c), exist_x(c))$. For any boolean condition *cond*, let $\delta(cond)$ be 1 if *cond* is true, 0 otherwise.

Definition 4.2.1 (Extended Prefix Representation) *For a string x, the extended prefix representation of order relations is defined as follows.*

$$Ex\text{-}pre(x) = (ex\text{-}rank_{x_1}(x[1]), ex\text{-}rank(x_2[2]), ..., ex\text{-}rank(x_{|x|}[|x|]))$$

For example, the extended prefix representations of x, y in the previous example are as follows.

$$\textit{Ex-pre}(x) = \left(\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}3\\0\end{pmatrix},\begin{pmatrix}2\\0\end{pmatrix},\begin{pmatrix}3\\1\end{pmatrix},\begin{pmatrix}2\\1\end{pmatrix},\begin{pmatrix}2\\1\end{pmatrix}\right)$$
$$\textit{Ex-pre}(y) = \left(\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}1\\0\end{pmatrix},\begin{pmatrix}3\\0\end{pmatrix},\begin{pmatrix}2\\0\end{pmatrix},\begin{pmatrix}3\\1\end{pmatrix},\begin{pmatrix}2\\1\end{pmatrix},\begin{pmatrix}4\\1\end{pmatrix}\right)$$

The relationship of $Nat(\cdot)$ and *Ex-Pre*$(\cdot)$ is given in Lemma 4.2.1 where $Nat(x_{t+1})$ can be computed from $Nat(x_t)$ and *Ex-pre*(x_{t+1}) for any $t \in [1..|x|-1]$.

Lemma 4.2.1 (Representation Conversion) *Given* $Nat(x_t)$ *and Ex-pre*(x_{t+1}),

$$Nat(x_{t+1})[i] = \begin{cases} a + \delta((a > c) \vee (a = c \wedge d = 0)) & \textit{for } 1 \leq i \leq t \\ c & \textit{for } i = t+1 \end{cases}$$

where $a = Nat(x_t)[i]$ *and* $\begin{pmatrix}c\\d\end{pmatrix} = \textit{Ex-pre}(x_{t+1})[t+1]$.

Proof: When $i = t+1$, it is obvious that $Nat(x_{t+1})[i] = c$ by definition.

Consider when $i \in [1..t]$. Let $b = Nat(x_{t+1})[i]$. Then, equation (4.1) holds because $a = rank_{x_t}(x[i])$ and $b = rank_{x_{t+1}}(x[i]) = rank_{x_t}(x[i]) + \delta(x[i] > x[t+1])$ (see Fig. 4.1).

$$b = a + \delta(x[i] > x[t+1]) \tag{4.1}$$

We have the following cases.

Case 1: $a > c$. Since $b \geq a$ from equation (4.1), we have $b > c$, which implies $Nat(x_{t+1})[i] > Nat(x_{t+1})[t+1]$. Therefore, $x[i] > x[t+1]$.

Case 2: $a < c$. Since $b \leq a+1$ from equation (4.1), and $a+1 \leq c$ from the case assumption, we have $b \leq c$, which implies $x[i] \leq x[t+1]$. If $x[i] = x[t+1]$, then $a = b = c$, which contradicts $a < c$. Therefore, $x[i] < x[t+1]$.

Case 3: $a = c$. Since $b \geq a$ from equation (4.1), we have $b \geq c$. Thus, $x[i] \geq x[t+1]$.

Case 3.1: $d = 0$. We have $x[i] \neq x[t+1]$, and thus $x[i] > x[t+1]$.

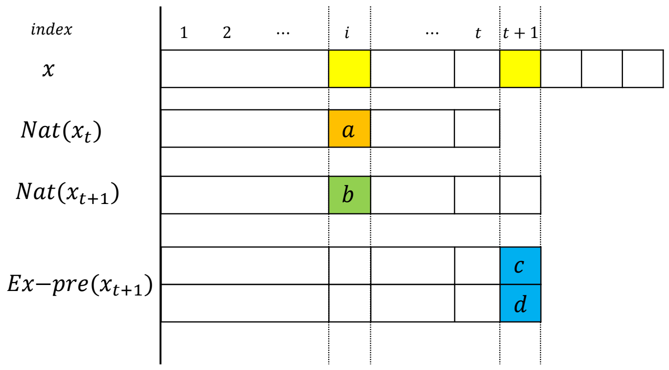


Figure 4.1: Symbols in Lemma 4.2.1

Case 3.2: $d = 1$. We have $x[j] = x[t+1]$ for some $j \in [1..t]$. If $x[i] > x[t+1]$, then $a \geq c+1$ is derived as follows.

$$\begin{aligned} a &= rank_{x_t}(x[i]) \\ &= rank_{x_{t+1}}(x[i]) - 1 \\ &\geq rank_{x_{t+1}}(x[t+1]) + 1 \text{ (because at least } x[t+1] \text{ and } x[j] \text{ are excluded)} \\ &= c+1 \end{aligned}$$

It contradicts $a = c$, which implies $x[i] = x[t+1]$.

From the above cases, $x[i] > x[t+1]$ if and only if $(a > c)$ or $(a = c \wedge d = 0)$. □

An example of Lemma 4.2.1 is shown in Figure 4.1 for $x = (30, 10, 50, 20, 30, 20, 25, 20)$. Let us consider when $t+1 = 7$. For $i = 1$, we have $Nat(x_6)[1] = a = 4$ and $Ex\text{-}pre(x_7)[7] = \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} 4 \\ 0 \end{pmatrix}$. From equation (4.1) of Lemma 4.2.1, $Nat(x_7)[1] = b = a + \delta(x[1] > x[7])$. Since $a = c$ and $d = 0$, it belongs to Case 3.1, and thus $x[1] > x[7]$, which implies $Nat(x_7)[1] = a + 1 = 5$. For $i = 2$, we have $Nat(x_6)[2] = a = 1$ and $\begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} 4 \\ 0 \end{pmatrix}$. Since $a < c$, it belongs to Case 2, and $x[2] < x[7]$, which implies $Nat(x_7)[2] = a = 1$. For $i = 3$, we have $Nat(x_6)[3] = a = 6$ and $\begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} 4 \\ 0 \end{pmatrix}$. Since $a > c$, it belongs to Case 1, and $x[3] > x[7]$, which implies $Nat(x_7)[3] = a + 1 = 7$. For $i = 7$, we get $Nat(x_7)[7] = c = 4$ since $i = t+1$.

i	1	2	3	4	5	6	7	8
x	30	10	50	20	30	20	25	20
$Nat(x_6)$	4	1	6	2	4	2		
Ex-pre(x_7)	$\begin{pmatrix}1\\0\end{pmatrix}$	$\begin{pmatrix}1\\0\end{pmatrix}$	$\begin{pmatrix}3\\0\end{pmatrix}$	$\begin{pmatrix}2\\0\end{pmatrix}$	$\begin{pmatrix}3\\1\end{pmatrix}$	$\begin{pmatrix}2\\1\end{pmatrix}$	$\begin{pmatrix}4\\0\end{pmatrix}$	
$NN(x_7)$	$\begin{pmatrix}-\infty\\\infty\end{pmatrix}$	$\begin{pmatrix}-\infty\\1\end{pmatrix}$	$\begin{pmatrix}2\\1\end{pmatrix}$	$\begin{pmatrix}2\\1\end{pmatrix}$	$\begin{pmatrix}1\\1\end{pmatrix}$	$\begin{pmatrix}4\\4\end{pmatrix}$	$\begin{pmatrix}6\\5\end{pmatrix}$	
$Nat(x_7)$	5	1	7	2	5	2	4	
Ex-pre(x_8)	$\begin{pmatrix}1\\0\end{pmatrix}$	$\begin{pmatrix}1\\0\end{pmatrix}$	$\begin{pmatrix}3\\0\end{pmatrix}$	$\begin{pmatrix}2\\0\end{pmatrix}$	$\begin{pmatrix}3\\1\end{pmatrix}$	$\begin{pmatrix}2\\1\end{pmatrix}$	$\begin{pmatrix}4\\0\end{pmatrix}$	$\begin{pmatrix}2\\1\end{pmatrix}$
$NN(x_8)$	$\begin{pmatrix}-\infty\\\infty\end{pmatrix}$	$\begin{pmatrix}-\infty\\1\end{pmatrix}$	$\begin{pmatrix}2\\1\end{pmatrix}$	$\begin{pmatrix}2\\1\end{pmatrix}$	$\begin{pmatrix}1\\1\end{pmatrix}$	$\begin{pmatrix}4\\4\end{pmatrix}$	$\begin{pmatrix}6\\5\end{pmatrix}$	$\begin{pmatrix}6\\6\end{pmatrix}$
$Nat(x_8)$	6	1	8	2	6	2	5	2

Table 4.1: Example of Lemma 4.2.1 and Lemma 4.3.1

Consider the next step when $t+1 = 8$. For $i = 4$, we have $Nat(x_7)[4] = a = 2$ and $\begin{pmatrix}c\\d\end{pmatrix} = \begin{pmatrix}2\\1\end{pmatrix}$. As $a = c$ and $d = 1$, it belongs to Case 3.2, and we get $x[4] = x[8]$, which implies $Nat(x_8)[4] = a = 2$.

Theorem 1 *Ex-pre*$(\cdot)$ *is equivalent to* $Nat(\cdot)$.

Proof: ($\Rightarrow$) Given *Ex-pre*(x), we can compute $Nat(x)$ by applying Lemma 4.2.1 repetitively to the prefixes of x. Therefore, $Nat(x) = Nat(y)$ if *Ex-pre*(x) = *Ex-pre*(y).
($\Leftarrow$) Given $Nat(x)$, we can compute *Ex-pre*(x) directly by comparing the rank values of $Nat(x)$, which implies that *Ex-pre*(x) = *Ex-pre*(y) if $Nat(x) = Nat(y)$.
□

Theorem 2 (Match Condition of Extended Prefix Representation) *Given x, y and t, the condition Ex-pre$(x_{t+1})[t+1]$ = Ex-pre$(y_{t+1})[t+1]$ is a match condition of Ex-pre$(\cdot)$.*

Proof: We need to prove the following two directions by Definition 4.1.2.

($\Rightarrow$) Suppose that $Nat(x_{t+1}) = Nat(y_{t+1})$. Both $Nat(x_t) = Nat(y_t)$ and *Ex-pre*(x_{t+1}) = *Ex-pre*(y_{t+1}) are derived from $Nat(x_{t+1}) = Nat(y_{t+1})$, and the condition holds.

($\Leftarrow$) Suppose that $Nat(x_t) = Nat(y_t)$ and the condition holds. *Ex-pre*(x_t) = *Ex-pre*(y_t) by Theorem 1, and we get *Ex-pre*(x_{t+1}) = *Ex-pre*(y_{t+1}) by the condition. Applying Theorem 1 to both *Ex-pre*(x_{t+1}) and *Ex-pre*(y_{t+1}), we get $Nat(x_{t+1}) = Nat(y_{t+1})$. $\square$

4.3 Extension of Nearest Neighbor Representation

In this section, we define the *nearest neighbor representation* [28, 32, 16], and prove that it is equivalent to the *natural representation.*

We define $LMax_x[i]$ and $LMin_x[i]$ as follows.

$$LMax_x[i] = \begin{cases} j & \text{if } x[j] = \max\{x[k] : x[k] \le x[i] \text{ for } 1 \le k \le i-1\} \\ -\infty & \text{if no such } j \end{cases}$$

$$LMin_x[i] = \begin{cases} j & \text{if } x[j] = \min\{x[k] : x[k] \ge x[i] \text{ for } 1 \le k \le i-1\} \\ \infty & \text{if no such } j \end{cases}$$

If there are multiple j's for $LMax_x[i]$ or $LMin_x[i]$, we choose the rightmost one. In Figure 4.1, $LMax_x[8] = 6$ since $x[6]$ is the rightmost one among the maximum values which are less than or equal to $x[8]$ in $x[1..7]$. Similarly, $LMin_x[8] = 6$.

Definition 4.3.1 (Nearest Neighbor Representation [28, 32, 16]) *For a string x, the nearest neighbor representation of order relations can be defined as* $NN(x) = \begin{pmatrix} LMax_x[1] \\ LMin_x[1] \end{pmatrix} \begin{pmatrix} LMax_x[2] \\ LMin_x[2] \end{pmatrix} \cdots \begin{pmatrix} LMax_x[|x|] \\ LMin_x[|x|] \end{pmatrix}$.

For example, for x = (30, 10, 50, 20, 30, 20, 20) and y = (35, 15, 55, 25, 35, 25, 35), the nearest neighbor representations are as follows.

$$NN(x) = \left(\begin{pmatrix}-\infty\\ \infty\end{pmatrix}, \begin{pmatrix}-\infty\\ 1\end{pmatrix}, \begin{pmatrix}1\\ \infty\end{pmatrix}, \begin{pmatrix}2\\ 1\end{pmatrix}, \begin{pmatrix}1\\ 1\end{pmatrix}, \begin{pmatrix}4\\ 4\end{pmatrix}, \begin{pmatrix}6\\ 6\end{pmatrix}\right)$$
$$NN(y) = \left(\begin{pmatrix}-\infty\\ \infty\end{pmatrix}, \begin{pmatrix}-\infty\\ 1\end{pmatrix}, \begin{pmatrix}1\\ \infty\end{pmatrix}, \begin{pmatrix}2\\ 1\end{pmatrix}, \begin{pmatrix}1\\ 1\end{pmatrix}, \begin{pmatrix}4\\ 4\end{pmatrix}, \begin{pmatrix}5\\ 5\end{pmatrix}\right)$$

For convenience, let $x[-\infty] = -\infty$, $x[\infty] = \infty$, $Nat(x)[-\infty] = 0$ and $Nat(x)[\infty] = |x| + 1$ for any string x. Then, $Nat(x)[LMax_x[i]] \leq Nat(x)[i] \leq Nat(x)[LMin_x[i]]$ holds for any $i \in [1..|x|]$ even when $LMax_x[i] = -\infty$ or $LMin_x[i] = \infty$.

The relationship between $Nat(\cdot)$ and $NN(\cdot)$ is given in Lemma 4.3.1.

Lemma 4.3.1 (Representation Conversion) *Given $Nat(x_t)$ and $NN(x_{t+1})$,*

$$Nat(x_{t+1})[i] = \begin{cases} a + \delta((a > f) \vee (a = f \wedge e \neq f)) & \text{for } 1 \leq i \leq t \\ f & \text{for } i = t+1 \end{cases}$$

where $a = Nat(x_t)[i]$, $\begin{pmatrix}c\\ d\end{pmatrix} = NN(x_{t+1})[t+1]$, $e = Nat(x_t)[c]$ and $f = Nat(x_t)[d]$.

Proof: When $i = t + 1$, we have $x[c] \leq x[t+1] \leq x[d]$ and $rank_{x_{t+1}}(x[t+1]) = rank_{x_t}(x[d])$ by definition of $NN(\cdot)$, which implies $Nat(x_{t+1})[t+1] = Nat(x_t)[d] = f$.

Consider when $i \in [1..t]$. Let $b = Nat(x_{t+1})[i]$. We have equation (4.2) as in Lemma 4.2.1.

$$b = a + \delta(x[i] > x[t+1]) \tag{4.2}$$

Case 1: $a > f$. $Nat(x_{t+1})[t+1] \leq Nat(x_{t+1})[d]$ by definition of $NN(\cdot)$, and $Nat(x_{t+1})[d] \leq f + 1$ because $rank_{x_{t+1}}(x[d]) \leq rank_{x_t}(x[d]) + 1$. We have $f + 1 \leq a$ from the case assumption, and $a \leq b$ by equation (4.2). By transitivity, $Nat(x_{t+1})[t+1] \leq b$ is derived, which implies $x[i] \geq x[t+1]$. If $x[t+1] = x[i]$, then $a = f$, which contradicts $a > f$. Thus, $x[i] > x[t+1]$.

Case 2: $a < e$. $Nat(x_{t+1})[t+1] \geq Nat(x_{t+1})[c]$ by definition of $NN(\cdot)$, and $Nat(x_{t+1})[c] \geq e$ because $rank_{x_{t+1}}(x[c]) \geq rank_{x_t}(x[c])$. We have $e \geq a+1$ from the case assumption, and $a+1 \geq b$ by equation (4.2). By transitivity, we get $Nat(x_{t+1})[t+1] \geq b$, and thus $x[i] \leq x[t+1]$. If $x[t+1] = x[i]$, then $a = e$, which contradicts $a < e$. Thus, $x[i] < x[t+1]$.

Case 3: $a = e = f$. We have $x[i] = x[c] = x[d]$ and $Nat(x_{t+1})[c] = Nat(x_{t+1})[d]$. Since $Nat(x_{t+1})[c] \leq Nat(x_{t+1})[t+1] \leq Nat(x_{t+1})[d]$, we have $x[t+1] = x[c]$. Hence, $x[i] = x[t+1]$.

Case 4: $e \leq a \leq f$ and $e \neq f$. Since $e \neq f$, $x[t+1]$ doesn't occur in x_t.

Case 4.1: $a = f$. We have $Nat(x_{t+1})[t+1] \leq Nat(x_{t+1})[d]-1$ since $x[t+1] \neq x[d]$, and $Nat(x_{t+1})[d]-1 \leq f$ since $rank_{x_{t+1}}(x[d]) \leq rank_{x_t}(x[d])+1$. We have $f = a$ from the case assumption, and $a \leq b$ by equation (4.2). By transitivity, $Nat(x_{t+1})[t+1] \leq b$, which implies $x[i] \geq x[t+1]$. Since $x[i] \neq x[t+1]$, $x[i] > x[t+1]$ holds.

Case 4.2: $a = e$. We have $Nat(x_{t+1})[t+1] \geq Nat(x_{t+1})[c]+1$ since $x[t+1] \neq x[c]$, and $Nat(x_{t+1})[c] \geq e$ since $rank_{x_{t+1}}(x[c]) \geq rank_{x_t}(x[c])$. We have $e = a$ from the case assumption, and $a+1 \geq b$ by equation (4.2). By transitivity, $Nat(x_{t+1})[t+1] \geq b$, which implies $x[i] \leq x[t+1]$. Since $x[i] \neq x[t+1]$, $x[i] < x[t+1]$ holds.

Case 4.3: $e < a < f$. We have $x[c] < x[i] < x[d]$, which contradicts the definition of $NN(\cdot)$ since $x[i]$ is closer than $x[c]$ or $x[d]$ to $x[t+1]$. Therefore, there is no such case.

From the above cases, $x[i] > x[t+1]$ if and only if $(a > f) \vee (a = f \wedge e \neq f)$. □

An example of Lemma 4.3.1 is shown in Figure 4.1 for $x = (30, 10, 50, 20, 30, 20, 25, 20)$. Let us consider when $t+1 = 7$. For $i = 1$, we have $Nat(x_6)[1] = a = 4$, $NN(x_7)[7] = \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} 6 \\ 5 \end{pmatrix}$, $e = 2$ and $f = 4$. Since $a = f$ and $e \neq f$, it belongs to Case 4.1, and we get $Nat(x_7)[1] = a+1 = 5$. For $i = 2$, $\begin{pmatrix} c \\ d \end{pmatrix}$, e and f are the same as for $i = 1$, and we have $a = 1$. Since $a < e$, it belongs to Case 2, which implies $Nat(x_7)[2] = a = 1$. For $i = 3$, we have $a = 6$ and $a > f$, and thus it belongs to Case 1. Therefore, $Nat(x_7)[3] = a + 1 = 7$. For $i = 4$, we have $a = 2$. Since $a = e$ and $e \neq f$, it belongs to Case 4.2, and we get $Nat(x_7)[4] = a = 2$. For $i = 7$, we get $Nat(x_7)[7] = f = 4$ since $i = t + 1$.

Consider the next step when $t + 1 = 8$. For $i = 4$, we have $a = 2$, $NN(x_8)[8] = \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$, $e = 2$ and $f = 2$. Since $a = e = f$, it belongs to Case 3, which implies $Nat(x_8)[4] = a = 2$.

Theorem 3 *$NN(\cdot)$ is equivalent to $Nat(\cdot)$.*

Proof: The proof is identical to that of Theorem 1 if we use Lemma 4.3.1 instead of Lemma 4.2.1. $\square$

A naive match condition of the nearest neighbor representation is $NN(x_{t+1})[t+1] = NN(y_{t+1})[t+1]$ as that of the extended prefix representation in Theorem 2, which requires computing the nearest neighbor representations of both x and y. Kubica et al. [32] proposed an efficient match condition for ternary order relations which can be checked in constant time when the nearest neighbor representation of x is given, but it was faulty. Cho et al. [16] presented a modified match condition in ternary order relations, which can produce an $O(n + m \log m)$ algorithm as in binary order relations. Since the match condition by Cho et al. [16] was proved in terms of order-isomorphism, we provide an alternative proof based on the natural representation in Theorem 4.

Theorem 4 (Match Condition of Nearest Neighbor Representation [16]) *Given x, y and t, the condition $(y[c] < y[t+1] < y[d]) \vee (y[t+1] = y[c] = y[d])$ is a match condition of $NN(\cdot)$ where $\begin{pmatrix} c \\ d \end{pmatrix} = NN(x_{t+1})[t+1]$.*

Proof:

We need to prove the following two directions by Definition 4.1.2.

($\Rightarrow$) Suppose that $Nat(x_{t+1}) = Nat(y_{t+1})$. Then, $Nat(x_t) = Nat(y_t)$ is obvious.

Case 1: $x[t+1]$ does not exist in x_t. We have $Nat(x_{t+1})[c] < Nat(x_{t+1})[t+1] < Nat(x_{t+1})[d]$, which implies $Nat(y_{t+1})[c] < Nat(y_{t+1})[t+1] < Nat(y_{t+1})[d]$. Therefore, $y[c] < y[t+1] < y[d]$.

Case 2: $x[t+1]$ exists in x_t. Let i be the rightmost position in x_t such that $x[i] = x[t+1]$. Then, $NN(x_{t+1})[t+1] = NN(y_{t+1})[t+1] = \begin{pmatrix} i \\ i \end{pmatrix} = \begin{pmatrix} c \\ d \end{pmatrix}$. Therefore, $y[t+1] = y[c] = y[d]$.

Combining the two cases, the condition holds.

($\Leftarrow$) Suppose that $Nat(x_t) = Nat(y_t)$ and the condition holds. First of all, we will show that c is the rightmost position of the characters $y[c]$ in y_t. For any $i \in [1..t]$, if $y[i] = y[c]$, then $x[i] = x[c]$ from $Nat(x_t) = Nat(y_t)$. As c is the rightmost position of the characters $x[c]$ in x_t, we have $i \leq c$ for such i. Similarly, d is the rightmost position of the characters $y[d]$ in y_t.

Case 1: $y[c] < y[t+1] < y[d]$. For any $i \in [1..t]$, $x[i] \leq x[c]$ or $x[i] \geq x[d]$. If $x[i] \leq x[c]$, then $y[i] \leq y[c]$ because $Nat(x_t) = Nat(y_t)$. Similarly, if $x[i] \geq x[d]$, then $y[i] \geq y[d]$. In both cases, $y[i] \leq y[c] < y[t+1]$ or $y[i] \geq y[d] > y[t+1]$, and thus $NN(y_{t+1})[t+1] = \begin{pmatrix} c \\ d \end{pmatrix}$.

Case 2: $y[t+1] = y[c] = y[d]$. As c and d are the rightmost position of the characters $y[c] = y[d]$ in y_t, we have $c = d$ and $NN(y_{t+1})[t+1] = \begin{pmatrix} c \\ d \end{pmatrix}$.

In both cases, $NN(x_{t+1})[t+1] = NN(y_{t+1})[t+1]$, and thus $NN(x_{t+1}) = NN(y_{t+1})$ since $Nat(x_t) = Nat(y_t)$. By Theorem 3, we get $Nat(x_{t+1}) = Nat(y_{t+1})$. □

4.4 Generalized Order-Preserving KMP Algorithm

The KMP failure function π is defined in terms of the natural representation [28]:

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

From the definition above, if $Nat(P_q) = Nat(T[i-q..i-1])$, the longest prefix of P whose natural representation coincides with that of a proper suffix of $T[i-q..i-1]$ is $P_{\pi[q]}$. For any representation $R(\cdot)$ of order relations, the generalized KMP algorithm can be written as follows.

```
GENERALIZED-KMP-ORDER-MATCHER($T, P$)
 1  $R(P) \leftarrow$ COMPUTE-REP($P$)
 2  $\pi \leftarrow$ KMP-COMPUTE-FAILURE-FUNCTION($P, R(P)$)
 3  $q \leftarrow 0$
 4  for $i \leftarrow 1$ to $|T|$
 5      while $q > 0$ and not MATCH($T[i-q..i], P, R(P), q+1$)
 6          $q \leftarrow \pi[q]$
 7      $q \leftarrow q+1$
 8      if $q = |P|$
 9          print “pattern occurs at position" $i$
10          $q \leftarrow \pi[q]$
```

In GENERALIZED-KMP-ORDER-MATCHER, we assume that COMPUTE-REP(P) computes $R(P)$ for any string P, KMP-COMPUTE-FAILURE-FUNCTION($P, R(P)$) computes the failure function π, and MATCH($x, y, R(x), t+1$) is a match condition of $R(\cdot)$.

The correctness comes from the loop invariant $Nat(P_q) = Nat(T[i-q..i-1])$ at line 5. If the match condition is true, then the matched length is increased by one. Otherwise, it is reduced to $\pi[q]$ without missing any matched positions.

The time complexity can be represented by $O(C+F+M\cdot n)$ where C is time for computing $R(P)$, F for computing π, and M for checking the match condition. If we use the extended prefix representation (i.e., $R = $ *Ex-pre*), then $C = O(m \log m)$, $F = O(m \log m)$ and $M = O(\log m)$, which produces $O(n \log m)$ time. If we use the nearest neighbor representation (i.e., $R = NN$), then $C = O(m \log m)$, $F = O(m)$ and $M = O(1)$, which produces $O(m \log m + n)$

time, which is consistent with the results in [28, 32].

The generalized KMP algorithm above is based on the abstraction of the match conditions of the representations of order relations. Similarly, if we generalize the Aho-Corasick algorithm in Section 3.1 for multiple pattern matching, we can obtain an $O((n + m) \log m)$ algorithm in ternary order relations using the extended prefix representation.

Chapter 5

Conclusion

In this thesis, we have introduced *order-preserving matching* and defined various representations of order relations of a numeric string. We have presented efficient algorithms for single and multiple pattern matching.

First, we have presented efficient algorithms for order-preserving single pattern matching in binary order relations. The order relations of a string are defined as a sequence of rank values which we call the *natural representation*, and *order-preserving matching* problem is defined in terms of the *natural representation.* The naive algorithm using the *natural representation* takes $O(nm \log m)$ time, which can be reduced to $O(n \log m)$ using the *prefix representation*, which is defined as the sequence of rank values *in the prefixes.* The KMP algorithm can be applied with the *prefix representation* with an additional $\log m$ term from the order-statistic tree. The time complexity is optimized to $O(n + m \log m)$ with the *nearest neighbor representation*, which is defined as the sequence of the two indices of the nearest values in the matched text substring. The advantage of the *nearest neighbor representation* is that the order relation of each text character can be checked in constant time.

Second, we have presented an efficient algorithm for order-preserving multiple pattern matching in binary order relations. We developed an $O((n + m) \log m)$ algorithm using the *prefix representation* based on the Aho-Corasick algorithm where m is the sum of lengths of all the patterns. In contrast to single pattern matching, the time complexity is not reduced further even with the *nearest neighbor representation*.

Third, we have extended our results to ternary order relations allowing multiple occurrences of equal characters. We have extended the *prefix representation* and the *nearest neighbor representation*, and presented the match conditions of both representations. The time complexities in binary order relations can be achieved in ternary order relations as well.

Order-preserving matching is a new class of string matching problems, and there are many variations of practical importance and theoretical interest [21, 25, 16]. We believe that there are still great opportunities for further research on order-preserving matching and its variations.

Bibliography

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

[2] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Comm. ACM*, 18(6):333–340, 1975.

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

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

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

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

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

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

[9] A. Amir, O. Lipsky, E. Porat, and J. Umanski. Approximate matching in the l_1 metric. In *CPM*, pages 91–103, 2005.

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

[11] R. W. Bailey. The number of weak orderings of a finite set. *Social Choice and Welfare*, 15(4):559–562, 1998.

[12] B. S. Baker. A theory of parameterized pattern matching: algorithms and applications. In *STOC*, pages 71–80, 1993.

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

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

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

[16] S. Cho, J. C. Na, K. Park, and J. S. Sim. Fast order-preserving pattern matching. In *COCOA*, 2013.

[17] P. Clifford, R. Clifford, and C. S. Iliopoulos. Faster algorithms for (δ, γ)-matching and related problems. In *CPM*, pages 68–78, 2005.

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

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

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

[21] 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 *SPIRE*, pages 84–95, 2013.

[22] M. Crochemore, C. S. Iliopoulos, T. Lecroq, W. Plandowski, and W. Rytter. Three heuristics for δ-matching: δ-BM algorithms. In *CPM*, pages 178–189, 2002.

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

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

[25] P. Gawrychowski and P. Uznanski. Order-preserving pattern matching with k mismatches. *CoRR*, abs/1309.6453, 2013.

[26] I. J. Good. The number of orderings of n candidates when ties are permitted. *Fibonacci Quarterly*, 13:11–18, 1975.

[27] O. A. Gross. Preferential arrangements. *The American Mathematical Monthly*, 69:4–8, 1962.

[28] J. Kim, P. Eades, R. Fleischer, S.-H. Hong, C. S. Iliopoulos, K. Park, S. J. Puglisi, and T. Tokuyama. Order-preserving matching. *To appear in Theor. Comput. Sci.*

[29] A. Knopfmacher and M. E. Mays. A survey of factorization counting functions. *Inter. J. Number Th*, 1:563–581, 2005.

[30] D. E. Knuth, J. H. M. Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[31] E. Kreyszig. *Advanced Engineering Mathematics.* John Wiley & Sons, 2010.

[32] 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):430–433, 2013.

[33] I. Lee, R. Clifford, and S.-R. Kim. Algorithms on extended (δ, γ)-matching. In *ICCSA (3)*, pages 1137–1142, 2006.

[34] I. Lee, J. Mendivelso, and Y. J. Pinzon. delta-gamma-parameterized matching. In *SPIRE*, pages 236–248, 2008.

[35] O. Lipsky and E. Porat. Approximate matching in the $\mathrm{l}_{\mathrm{infinity}}$ metric. *Inf. Process. Lett.*, 105(4):138–140, 2008.

[36] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[37] J. Mendivelso, I. Lee, and Y. J. Pinzon. Approximate function matching under δ- and γ- distances. In *SPIRE*, pages 348–359, 2012.

[38] S. Muthukrishnan. New results and open problems related to non-standard stringology. In *CPM*, pages 298–317, 1995.

[39] E. Porat and K. Efremenko. Approximating general metric distances between a pattern and a text. In *SODA*, pages 419–427, 2008.

[40] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

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

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

초 록

문자열 매칭은 컴퓨터 과학에서 수십 년간 연구되어온 중요한 문제이다. 때때로 문자열은 알파벳이 아니라 숫자로 구성되고 여기서 특정 패턴이 아니라 경향성이 중요한 의미를 가지는 경우가 있다. 이를 찾기 위해 본 논문에서는 수치 문자열에 대하여 순서를 보존하는 매칭 문제(order-preserving matching)를 소개한다. 순서를 보존하는 매칭에서 패턴은 같은 길이를 가지는 텍스트의 부분문자열과 상대적인 크기 순서가 일치할 때 매칭된다. 순서를 보존하는 매칭은 주식 가격 분석이나 유사 멜로디 매칭과 같은 경우에 적용 가능하다.

본 논문에서는 수치문자열에 대해 순서를 보존하는 매칭 문제를 정의하고 다양한 순서 관계 표현들과 이를 이용한 효율적인 알고리즘들을 제안한다. 패턴이 하나인 경우, 패턴의 길이가 m, 텍스트의 길이가 n일 때, 우리는 접두사 표현 방법(prefix representation)을 이용하여 KMP 알고리즘에 기반한 수행시간이 $O(n \log m)$인 알고리즘을 제안하고, 이를 최적화하여 근접 이웃 표현 방법(nearest neighbor representation)을 이용한 수행시간이 $O(n + m \log m)$인 알고리즘을 얻는다. 패턴이 여러 개인 경우, 모든 패턴의 길이의 합이 m, 텍스트의 길이가 n일 때, 접두사 표현 방법을 이용하여 Aho-Corasick 알고리즘에 기반한 수행시간이 $O((n + m) \log m)$인 알고리즘을 제안한다. 본 논문에서는 먼저 2항 순서 관계(binary order relation)를 가정한 알고리즘들을 제안하고 이 결과를 3항 순서 관계(ternary order relation)로 확장한다. 본 논문의 확장을 이용하면 2항 순서 관계에서 얻은 시간복잡도들을 3항 순서 관계에서도 얻을 수 있다.

주요어: 순서를 보존하는 매칭, 순서 관계, 패턴 매칭, 수치 문자열, KMP 알고리즘, Aho-Corasick 알고리즘

학번: 2007-30219