Information Processing Letters 115 (2015) 397–402

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

A fast algorithm for order-preserving pattern matching ☆

Sukhyeun Cho [a], Joong Chae Na [b], Kunsoo Park [c], Jeong Seop Sim [a,*]

[a] Department of Computer and Information Engineering, Inha University, Incheon 402-751, South Korea
[b] Department of Computer Science and Engineering, Sejong University, Seoul 143-747, South Korea
[c] School of Computer Science and Engineering, Seoul National University, Seoul 151-742, South Korea

ARTICLE INFO

Article history:
Received 8 April 2014
Received in revised form 27 September 2014
Accepted 29 October 2014
Available online 1 November 2014
Communicated by Tsan-sheng Hsu

Keywords:
Analysis of algorithms
Order-preserving pattern matching
Order-isomorphism
Horspool algorithm
KMP algorithm

ABSTRACT

Given a text T and a pattern P, the order-preserving pattern matching (OPPM) problem is to find all substrings in T which have the same relative orders as P. The OPPM has been studied in the fields of finding some patterns affected by relative orders, not by their absolute values. In this paper, we present a method of deciding the order-isomorphism between two strings even when there are same characters. Then, we show that the bad character rule of the Horspool algorithm for generic pattern matching problems can be applied to the OPPM problem and we present a space-efficient algorithm for computing shift tables for text search. Finally, we combine our bad character rule with the KMP-based algorithm to improve the worst-case running time. We give experimental results to show that our algorithm is about 2 to 6 times faster than the KMP-based algorithm in reasonable cases.

1. Introduction

Given a text T and a pattern P, the order-preserving pattern matching (OPPM for short) problem is to find all substrings in T which have the same relative orders as P. For example, when $P = (35, 40, 23, 40, 40, 28, 30)$ and $T = (10, 20, 15, 28, 32, 12, 32, 32, 20, 25, 15, 25)$ are given, P has the same relative orders as the substring $T' = (28, 32, 12, 32, 32, 20, 25)$ of T. In T' (resp. P), the first character 28 (resp. 35) is the 4-th smallest, the second character 32 (resp. 40) is the 5-th smallest, the third character 12 (resp. 23) is the smallest, and so on. See Fig. 1. The OPPM has been studied in the fields of finding some patterns affected by relative orders, not by their absolute values. For example, it can be applied to time series analysis like share prices on stock markets and to musical melody matching of two musical scores [2].

Recently, several results were presented on the OPPM problem. For the OPPM problem, the order-isomorphism must be defined. Kim et al. [2] defined the order-isomorphism as the equivalence of permutations converted from strings with an assumption that all the characters in a string are distinct. Given T ($|T| = n$) and P ($|P| = m$), they proposed an algorithm for the OPPM problem running in $O(n + m \log m)$ time based on the Knuth–Morris–Pratt (KMP) algorithm [3]. Meanwhile, Kubica et al. [4] defined the order-isomorphism as the equivalence of all relative orders between two strings, and presented a method of deciding the order-isomorphism of two strings even when there are same characters. They independently proposed an algorithm for the OPPM problem based on the KMP algorithm running in $O(n + m \log m)$ time for a general alphabet and $O(n + m)$ time for an integer alphabet whose characters can be sorted in linear time. More recently, Crochemore et al. [5] introduced order-preserving suffix trees, and they suggested an algorithm finding all occurrences of P in T running in $O(m + z)$ time where z is the number of occurrences.

☆ A preliminary version of this paper appeared in COCOA 2013 [1].

* Corresponding author. Tel.: +82 32 860 7455.
E-mail address: jssim@inha.ac.kr (J.S. Sim).

http://dx.doi.org/10.1016/j.ipl.2014.10.018
0020-0190/

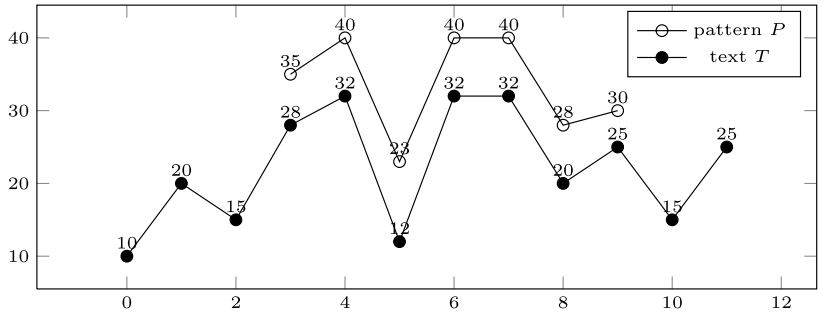

Fig. 1. An OPPM example for $P = (35, 40, 23, 40, 40, 28, 30)$ and $T = (10, 20, 15, 28, 32, 12, 32, 32, 20, 25, 15, 25)$.

In this paper, we propose fast algorithms for the OPPM problem based on the Horspool algorithm [6–8]. Experimental results show that our algorithms are about 2 to 6 times faster than the KMP-based algorithm in reasonable cases. Our contributions are as follows.

- We present a method of deciding the order-isomorphism between two strings even when there are same characters. We show that Kubica et al.'s method [4] may decide it incorrectly when there are same characters.
- We show that the bad character rule can be applied to the OPPM problem by defining a group of characters as one character. Kim et al. [2] mentioned the hardness of applying the Boyer–Moore algorithm [9] to the OPPM problem. The good suffix rule could be well-defined but the bad character rule could not be directly applied to the OPPM problem.
- We present a space-efficient algorithm computing the shift table for text search based on a factorial number system. Let q be a size of the group of characters and $|\Sigma|$ be the size of an alphabet. Then, our algorithm uses $O(q!)$ space for the shift table while the algorithms of [6,7] for the generic pattern matching problem use $O(|\Sigma|^q)$ space for the shift table.
- We also show that our bad character rule can be combined with the KMP-based algorithm to improve the worst-case running time of [1]. The combined algorithm guarantees $O(n + m \log m)$ time for a general alphabet and $O(n + m)$ time for an integer alphabet in the worst case when q is a constant.

2. Preliminaries

Let Σ denote an alphabet and $\sigma = |\Sigma|$. Let $|x|$ denote the length of a string x. A string x is described by a sequence of characters $(x[0], x[1], \ldots, x[|x| - 1])$.

Now, we formally define the order-isomorphism and the order-preserving pattern matching problem. Two strings x and y of the same length over Σ are called *order-isomorphic*, written $x \approx y$, if

$$x[i] \le x[j] \quad \Leftrightarrow \quad y[i] \le y[j] \quad \text{for } 0 \le i, j < |x| \text{ [4].}$$

If two strings x and y are not order-isomorphic, we write $x \not\approx y$. Given a text $T[0..n-1]$ and a pattern $P[0..m-1]$, we say that T *matches* P at position i if $T[i-m+1..i] \approx P$. In the previous example shown in Fig. 1, T matches P at position 9 because $T[3..9] \approx P$. The *order-preserving pattern matching problem* is to find all positions of T matched with P.

Let us define a *prefix table* μ_x of string x:

$$\mu_x[i] = \left|\left\{j : x[j] \le x[i] \text{ for } 0 \le j < i\right\}\right|.$$

See Table 1 for an example.

Table 1
μ_P, $LMax_P$, $LMin_P$, π_P for $P = (35, 40, 23, 40, 40, 28, 30)$.

i	0	1	2	3	4	5	6
$P[i]$	35	40	23	40	40	28	30
$\mu_P[i]$	0	1	0	3	4	1	2
$LMax_P[i]$	−1	0	−1	1	3	2	5
$LMin_P[i]$	−1	−1	0	1	3	0	0
$\pi_P[i]$	0	1	1	2	1	1	2

Lemma 1. *For two strings x and y, if $x \approx y$, then $\mu_x = \mu_y$.*

Proof. By the assumption that $x \approx y$, $x[i] \le x[j] \Leftrightarrow y[i] \le y[j]$ for $0 \le i < j < |x|$. Hence, $\mu_x = \mu_y$. □

Lemma 2. *Assume that $x[0..t] \approx y[0..t]$. For all $0 \le i, j \le t$, if $x[i] < x[j]$, then $y[i] < y[j]$, and if $x[i] = x[j]$, then $y[i] = y[j]$.*

Proof. We first prove by contradiction the first proposition (when $x[i] < x[j]$). Suppose that $y[i] \ge y[j]$. Then, by the definition of order-isomorphism, $x[i] \ge x[j]$, which contradicts the assumption that $x[i] < x[j]$.

Next, consider the case when $x[i] = x[j]$. Then, since $x[i] \le x[j]$, $y[i] \le y[j]$ by the definition of order-isomorphism. Moreover, since $x[j] \le x[i]$, $y[j] \le y[i]$. Since $y[i] \le y[j]$ and $y[j] \le y[i]$, $y[i] = y[j]$. □

Kubica et al. [4] used location tables called *LMax* and *LMin* for the order information of prefixes of P: Given a string x, for $i = 0, \ldots, |x| - 1$,

$$LMax_x[i] = j$$
$$\text{if } x[j] = \max\left\{x[k] : k \in [0, i-1], x[k] \le x[i]\right\} \quad \text{and}$$
$$LMin_x[i] = j$$
$$\text{if } x[j] = \min\left\{x[k] : k \in [0, i-1], x[k] \ge x[i]\right\}.$$

If there is no such j then $LMin_x[i] = -1$ and $LMax_x[i] = -1$. If more than one such j exists, we select the rightmost one among them. Intuitively, $LMax_x[i]$ indicates the position of the largest character which is not larger than $x[i]$ in $x[0..i-1]$, and $LMin_x[i]$ indicates the position of the smallest character which is not smaller than $x[i]$ in $x[0..i-1]$. See Table 1 for an example. Notice the location tables of x can be computed in $O(|x|)$ time for an integer alphabet and in $O(|x| \log |x|)$ time for a general alphabet [4].

In the KMP algorithm, the failure function π_x for x is well-defined in the order-preserving pattern matching [2,4]. See Table 1 for an example.

3. New decision of order-isomorphism

In this section, we show that Kubica et al.'s method [4] for deciding the order-isomorphism of two strings may be incorrect when there are same characters and present a new method which corrects Kubica et al.'s one. Kubica et al. [4] claimed that the order-isomorphism of two strings x and y could be decided using the location tables as follows.

Lemma 3. *(See [4].) Assume that $x[0..t] \approx y[0..t]$, $t < |x| - 1, |y| - 1$ and $a = LMax_x[t+1]$, $b = LMin_x[t+1]$. Then, $x[0..t+1] \approx y[0..t+1] \Leftrightarrow y[a] \le y[t+1] \le y[b]$. In case a or b is equal to -1, we omit the respective inequality in the condition.*

This lemma may not hold when there are same characters. For example, consider two strings $x = (1, 3, 2)$ and $y = (1, 2, 2)$. Then, $y[LMax_x[i]] \le y[i] \le y[LMin_x[i]]$ for all $0 \le i < 3$. But, by the definition of order-isomorphism, $y \not\approx x$ because $x[1] \not\le x[2]$ and $y[1] \le y[2]$. The reasons why Lemma 3 may not hold when there are same characters in the given strings are as follows. In the proof of Lemma 3, to show $x[0..t+1] \approx y[0..t+1]$ (when $y[a] \le y[t+1] \le y[b]$), they tried to prove that $x[i] \le x[t+1] \Leftrightarrow y[i] \le y[t+1]$ for $i \le t$. For this, they proved that $x[i] \le x[t+1] \Rightarrow y[i] \le y[t+1]$ and $x[i] \ge x[t+1] \Rightarrow y[i] \ge y[t+1]$. But, it is not equivalent to $x[i] \le x[t+1] \Leftrightarrow y[i] \le y[t+1]$. Instead of the latter $x[i] \ge x[t+1] \Rightarrow y[i] \ge y[t+1]$, it should be proven that $x[i] > x[t+1] \Rightarrow y[i] > y[t+1]$. As seen in our example, however, $x[1] > x[2] \not\Rightarrow y[1] > y[2]$.

We show a new lemma for deciding the order-isomorphism of two strings even when there are same characters.

Lemma 4. *Assume that $x[0..t] \approx y[0..t]$, $t < |x| - 1, |y| - 1$ and $a = LMax_x[t+1]$, $b = LMin_x[t+1]$. Let p be the condition $y[a] < y[t+1]$ and q be the condition $y[t+1] < y[b]$. Then, $x[0..t+1] \approx y[0..t+1] \Leftrightarrow (p \wedge q) \vee (\neg p \wedge \neg q)$. In case a or b is equal to -1, we assume the respective condition p or q is true.*

Proof. Without loss of generality, we assume that $a \neq -1$ and $b \neq -1$. By definitions of *LMax* and *LMin*, $x[a] \le x[b]$ and also $y[a] \le y[b]$ since $a, b \le t$ and $x[0..t] \approx y[0..t]$. Hence, $(\neg p \wedge \neg q)$, i.e., $y[a] \ge y[t+1] \ge y[b]$ is equivalent to $y[a] = y[t+1] = y[b]$ under the assumption $x[0..t] \approx y[0..t]$.

($\Rightarrow$) We show $x[0..t+1] \approx y[0..t+1] \Rightarrow (p \wedge q) \vee (\neg p \wedge \neg q)$. By definitions of *LMax* and *LMin*, $x[a] \le x[t+1] \le x[b]$. We have two cases according to whether $x[a] = x[b]$ or not.

- Case when $x[a] = x[b]$: In this case, $x[a] = x[t+1] = x[b]$. Since $x[0..t+1] \approx y[0..t+1]$, $y[a] = y[t+1] = y[b]$ by Lemma 2, i.e., $(\neg p \wedge \neg q)$ is true.
- Case when $x[a] < x[b]$: We first prove that $x[a] \neq x[t+1] \neq x[b]$. Without loss of generality, suppose $x[t+1] = x[a]$. Then, $LMax_x[t+1] = LMin_x[t+1]$ by definitions. Hence, $x[a] = x[b]$, which contradicts the condition that $x[a] < x[b]$. Since $x[a] \neq x[t+1] \neq x[b]$, $x[a] < x[t+1] < x[b]$ and thus $y[a] < y[t+1] < y[b]$ by Lemma 2, i.e., $(p \wedge q)$ is true.

Therefore, $x[0..t+1] \approx y[0..t+1] \Rightarrow (p \wedge q) \vee (\neg p \wedge \neg q)$.

($\Leftarrow$) We show $x[0..t+1] \approx y[0..t+1] \Leftarrow (p \wedge q) \vee (\neg p \wedge \neg q)$. Since we have already $x[0..t] \approx y[0..t]$ (assumption), to show $x[0..t+1] \approx y[0..t+1]$, we only need to prove that for all $i \le t$,

$$x[i] \le x[t+1] \quad \Leftrightarrow \quad y[i] \le y[t+1] \quad \text{and}$$

$$x[t+1] \le x[i] \quad \Leftrightarrow \quad y[t+1] \le y[i].$$

We only consider the former, i.e., $x[i] \le x[t+1] \Leftrightarrow y[i] \le y[t+1]$. (The latter can be proven in a similar way.)

We first show that $x[i] \le x[t+1] \Rightarrow y[i] \le y[t+1]$ when $(p \wedge q) \vee (\neg p \wedge \neg q)$. If $x[i] \le x[t+1]$, $x[i] \le x[a]$ by the definition of *LMax* and also $y[i] \le y[a]$ since $x[0..t] \approx y[0..t]$. Finally, by the hypothesis $(p \wedge q) \vee (\neg p \wedge \neg q)$, $y[a] \le y[t+1]$. Hence, we get $y[i] \le y[t+1]$.

Next, we show that $y[i] \le y[t+1] \Rightarrow x[i] \le x[t+1]$ when $(p \wedge q) \vee (\neg p \wedge \neg q)$. We have two cases according to the hypothesis $(p \wedge q) \vee (\neg p \wedge \neg q)$.

- Case when $y[a] = y[t+1] = y[b]$ $(\neg p \wedge \neg q)$: In this case, $x[a] = x[b]$ by Lemma 2, and thus $x[a] = x[t+1] = x[b]$. When $y[i] \le y[a] (= y[t+1])$, $x[i] \le x[a] (= x[t+1])$ since $x[0..t] \approx y[0..t]$. Hence, $y[i] \le y[t+1] \Rightarrow x[i] \le x[t+1]$.
- Case when $y[a] < y[t+1] < y[b]$ $(p \wedge q)$: We prove it by contradiction. Suppose there exists $x[i]$ $(0 \le i \le t)$ such that $x[i] > x[t+1]$ when $y[i] \le y[t+1]$. Then, $x[i] \ge x[b]$ by the definition of *LMin*, and thus $y[i] \ge y[b]$ since $x[0..t] \approx y[0..t]$. Moreover, since $y[b] > y[t+1]$, we have $y[i] > y[t+1]$. It contradicts the condition that $y[i] \le y[t+1]$.

Therefore, $(p \wedge q) \vee (\neg p \wedge \neg q) \Rightarrow x[0..t+1] \approx y[0..t+1]$. □

For example, let us consider again the two strings $x = (1, 3, 2)$, $y = (1, 2, 2)$ and the location tables $LMax_x = (-1, 0, 0)$, $LMin_x = (-1, -1, 1)$ shown as the counter-example. Obviously, $x[0..1] \approx y[0..1]$ by the definition of the order-isomorphism. Then, $y \not\approx x$ because $y[LMax_x[2]] < y[2] = y[LMin_x[2]]$.

4. Fast order-preserving pattern matching algorithm

4.1. Basic idea

Basically, our algorithm for the OPPM problem is based on the Horspool algorithm widely used for generic pattern matching problems. The Horspool algorithm for generic pattern matching problems uses the shift table for filtering mismatched positions to expect sublinear behavior. (This method is well known as the bad character rule.) That is, when a mismatch occurs, the generic Horspool algorithm shifts the pattern using the shift table by setting the character of T compared with $P[m-1]$ as the bad character. However, as mentioned in [2], it is not easy to apply the bad character rule to the OPPM problem since the order-isomorphism is defined using the orders of characters, not just the character itself.

To solve the hardness of defining bad characters in the OPPM, we use the notion of q-grams, as in some variants [6,7] of the Horspool algorithm, which consider q consecutive characters as one character. Given a q-gram x and a pattern P of length m, let us define

$$l_x = \max\{i \mid \mu_{P[i-q+1..i]} = \mu_x \text{ for } q-1 \le i < m-1\}.$$

Roughly, l_x means the last position of P matching a q-gram x. Since, if $\mu_{P[i-q+1..i]} \neq \mu_x$, $P[i-q+1..i] \not\approx x$ by Lemma 1, we do not miss any position of P that matches the q-gram x. Then, the shift table D is defined as

$$D\big[f(x)\big] = \min(m-q+1, m-l_x-1),$$

where $f(x)$ is a fingerprint mapping a q-gram x to an integer. For space-efficiency of the shift table, differently from the variants [6,7] of the Horspool algorithm, we define $f(x)$ using the prefix table μ_x and a factorial number system [10] as follows:

$$f(x) = \sum_{k=0}^{q-1} \mu_x[k] \cdot k!.$$

Note that since there are $i+1$ possible values for the i-th element of the prefix table, we can use the factorial number system [10,11]. Since the prefix tables are uniquely mapped to integers from 0 to $q!-1$ [10,11], our shift table D needs $O(q!)$ space.

4.2. Search algorithm

Our algorithm consists of two steps. In the first step, we compute the location tables $LMax_P$, $LMin_P$ and the shift table D of pattern P. As mentioned above, the location tables can be computed in $O(m \log m)$ time for a general alphabet and can be computed in $O(m)$ time for an integer alphabet [4]. To compute D, all the fingerprints of q-grams of P must be computed. For all the q-grams of P, prefix tables can be computed in $O(mq \log q)$ time using dynamic order-statistics trees [2] for a general alphabet and can be computed in $O(mq)$ time using word-encoded sets [11] for an integer alphabet where $\sigma = 2^{\lfloor w/q \rfloor - 1}$ and w is the word size. Then, after computing all the prefix tables, all the fingerprints can be computed in $O(mq)$ time by Horner's rule [3]. Finally, D can be computed in $O(q! + mq \log q)$ time [6,7]. Note that we need $O(q!)$ time for initialization of D. The first step takes $O(q! + mq \log q + m \log m)$ for a general alphabet.

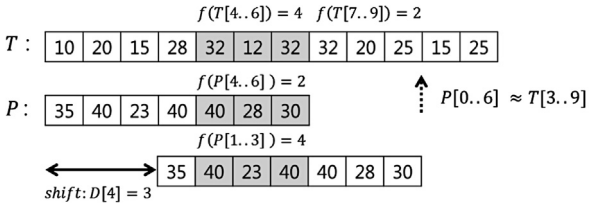

Fig. 2. Searching $P = (35, 40, 23, 40, 40, 28, 30)$ in $T = (10, 20, 15, 28, 32, 12, 32, 32, 20, 25, 15, 25)$.

Algorithm 1 shows the pseudo-code of our algorithm searching for P in T using the shift table D. Suppose we check if P matches $T[i-m+1..i]$. We first compare the last q-grams of P and $T[i-m+1..i]$ using their fingerprints, i.e., $f(P[m-q..m-1])$ and $f(T[i-q+1..i])$. If they are the same, we check the order-isomorphism of P and $T[i-m+1..i]$ character by character using $LMax_P$ and $LMin_P$ (Lemma 4). Otherwise, we do not compare P and $T[i-m+1..i]$ because $T[i-m+1..i]$ cannot be order-isomorphic to P by Lemma 1. Then, we shift P forward by $D[f(T[i-q+1..i])]$. We repeat this until P reaches the rightmost of T. Fig. 2 shows how Algorithm 1 works on the previous example shown in Fig. 1. We first compare the fingerprints $f(T[4..6]) = 4$ and $f(P[4..6]) = 2$. Since they are distinct, we shift P by $D[f(T[4..6])] = D[4] = 3$. Next, since $f(T[7..9])$ and $f(P[4..6])$ are the same, we compare P and $T[3..9]$ using Lemma 4. Since $P \approx T[3..9]$, Algorithm 1 reports the position 9 as an occurrence. Since the second step takes $O(nm + nq \log q)$ time for a general alphabet, Algorithm 1 takes $O(nm + nq \log q + q!)$ time overall. For an integer alphabet of size $\sigma = 2^{\lfloor w/q \rfloor - 1}$ where w is the word size, Algorithm 1 takes $O(nm + nq + q!)$ time.

4.3. Improving the worst-case complexity

We can improve the worst-case running time of Algorithm 1 by combining with the KMP-based algorithm. A similar approach was used in [12] for generic pattern matching. Algorithm 2 shows the pseudo-code of our improved version consisting of two parts: the Horspool-based algorithm (lines 6 to 8) and the KMP-based algorithm (lines 9 to 13). Initially, we run the Horspool-based algorithm until a match of fingerprints occurs. If fingerprints of q-grams of P and T do not match (line 6), we continue to run the Horpool-based algorithm. But, when they match, we run the KMP-based algorithm to check whether P occurs at position i (lines 10–12) and to compute the amount of a pattern shift $j - \pi_P[j-1]$ (line 13). If $j \neq 1$, we continue to run the KMP-based algorithm; otherwise, we run the Horspool-based algorithm again.

Now we analyze the time complexity of Algorithm 2. For a general alphabet, the preprocessing step (line 1) runs in $O(q! + mq \log q + m \log m)$ time since the failure function π_P can be computed in $O(m)$ time [4,2] using Lemma 4. Note that at most $n-m+1$ fingerprints

Algorithm 1

```
1: Preprocess D, LMax_P, LMin_P
2: m ← |P|, n ← |T|, t ← f(P[m − q..m − 1])
3: i ← m − 1
4: while i < n do
5:     c ← f(T[i − q + 1..i])
6:     if c = t then                                   ▷ Compare the last q-grams
7:         if T[i − m + 1..i] ≈ P then
8:             print "pattern occurs at position" i
9:     i ← i + D[c]                                    ▷ Shift P by D[c]
```

Algorithm 2

```
1: Preprocess D, LMax_P, LMin_P, π_P
2: m ← |P|, n ← |T|, t ← f(P[m − q..m − 1])
3: i ← m − 1, j ← 1
4: while i < n do
5:     if j = 1 then
6:         while (c ← f(T[i − q + 1..i])) ≠ t do        ▷ Horspool-based algorithm
7:             i ← i + D[c]
8:             if i ≥ n then return
9:     s ← i − m + 1
10:    while j < m and P[0..j] ≈ T[s..s + j] do         ▷ KMP-based algorithm
11:        j ← j + 1
12:    if j = m then print "pattern occurs at position" i
13:    i ← i + (j − π_P[j − 1]), j ← max(1, π_P[j − 1])
```

Table 2
Search times (in seconds) for 1,000 random patterns in a random text of length 5,000,000.

σ	m	5	10			15			20		
	q	3	3	4	5	3	4	5	3	4	5
2^{30}	OKMP	39.9	39.9			39.7			39.9		
	OHq	**29.3**	17.5	**13.3**	15.2	15.6	8.9	8.9	15.3	7.2	**6.5**
	OHy	32.1	19.6	13.7	15.0	17.9	9.2	**8.8**	17.6	7.4	**6.5**
10	OKMP	39.0	39.1			38.9			38.9		
	OHq	**29.0**	17.4	**13.4**	15.2	15.6	9.1	9.0	15.3	7.3	6.6
	OHy	30.9	19.8	13.8	15.1	18.0	9.3	**8.9**	17.7	7.5	**6.5**
4	OKMP	39.5	39.0			39.0			39.2		
	OHq	**31.2**	18.8	**14.3**	15.6	17.0	9.9	**9.3**	16.6	8.2	**7.0**
	OHy	33.0	21.7	15.1	15.6	19.7	10.4	**9.3**	19.1	8.6	**7.0**
2	OKMP	**38.1**	38.3			38.1			38.2		
	OHq	**38.1**	24.6	19.2	**18.3**	21.7	14.4	**12.0**	21.0	12.3	**9.2**
	OHy	38.2	27.3	20.7	19.0	25.1	15.9	12.5	24.2	13.3	9.6

of T are computed, the Horspool-based algorithm takes $O(nq\log q)$ time in total. Since the KMP-based algorithm takes $O(n+m\log m)$ time [4,2] in total, Algorithm 2 takes $O((m+n)q\log q+q!+m\log m)$ time overall. For an integer alphabet where $\sigma=2^{\lfloor w/q\rfloor-1}$ and w is the word size, the preprocessing step takes $O(q!+mq)$ time as explained above. The Horspool-based algorithm and the KMP-based algorithm take $O(nq)$ time and $O(n)$ time, respectively, Algorithm 2 takes $O((m+n)q+q!)$ time in total.

5. Experimental results

We conducted experiments to compare the practical performance of our algorithms and the KMP-based algorithm. Our Horspool-based algorithm and the worst-case improved algorithm are denoted by OHq and OHy, respectively. The KMP-based algorithm, denoted by OKMP, was implemented based on the algorithms of [2,4]. We used a naive approach to compute the fingerprints instead of using dynamic order-statistics trees or word-encoded sets because they are less practical when implemented. All algorithms were implemented in C++ and performed on a Windows 8 PC (64 bit) with Intel Core i7 3820 processor and 32 GB RAM.

We tested for a random text T of length $n=5{,}000{,}000$ from an integer alphabet and searched for 1,000 random patterns of length $m=5,10,15,20$, respectively. We performed experiments with varying q from 3 to 5 and $\sigma=2^{30},10,4,2$. Note that when $m=5$, we performed tests for the case only when $q=3$ because if q is almost the same as m, q-gram technique has no effect on speedup.

Table 2 shows search times. As the pattern length m becomes longer, OHq and OHy run faster compared to OKMP. Especially, for example, when $\sigma=2^{30}$, $m=20$, and $q=5$, OHq and OHy are about 6 times faster than OKMP. However, as σ decreases, OHq and OHy do not work well compared to OKMP, for example, when $\sigma=2$, $m=10$, and $q=5$, they are about 2 times faster than

OKMP. The reason why our algorithms are relatively slower in this case is because they are based on the Horspool algorithm which works better as patterns are longer and σ is larger.

Acknowledgements

Joong Chae Na was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (2014R1A1A1004901). Kunsoo Park was supported by 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). Jeong Seop Sim was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. 2012R1A2A2A01014892 & 2014R1A2A1A11050337), by the IT R&D program of MSIP/KEIT [10041971, Development of Power Efficient High-Performance Multimedia Contents Service Technology using Context-Adapting Distributed Transcoding], and by Inha University Research Grant.

References

[1] S. Cho, J.C. Na, K. Park, J.S. Sim, Fast order-preserving pattern matching, in: Combinatorial Optimization and Applications, Springer, 2013, pp. 295–305.
[2] J. Kim, P. Eades, R. Fleischer, S.-H. Hong, C.S. Iliopoulos, K. Park, S.J. Puglisi, T. Tokuyama, Order-preserving matching, Theor. Comput. Sci. 525 (2014) 68–79.
[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 3rd edition, The MIT Press, 2009.
[4] M. Kubica, T. Kulczynski, J. Radoszewski, W. Rytter, T. Walen, A linear time algorithm for consecutive permutation pattern matching, Inf. Process. Lett. 113 (12) (2013) 430–433.
[5] M. Crochemore, C.S. Iliopoulos, T. Kociumaka, M. Kubica, A. Langiu, S.P. Pissis, J. Radoszewski, W. Rytter, T. Waleń, Order-preserving incomplete suffix trees and order-preserving indexes, in: String Processing and Information Retrieval, Springer, 2013, pp. 84–95.
[6] R. Baeza-Yates, Improved string searching, Softw. Pract. Exp. 19 (3) (1989) 257–271.
[7] J. Tarhio, H. Peltola, String matching in the DNA alphabet, Softw. Pract. Exp. 27 (7) (1997) 851–861.
[8] R.N. Horspool, Practical fast searching in strings, Softw. Pract. Exp. 10 (6) (1980) 501–506.
[9] R.S. Boyer, J.S. Moore, A fast string searching algorithm, Commun. ACM 20 (10) (1977) 762–772.
[10] D.E. Knuth, The Art of Computer Programming, Volume 2: Seminumerical Algorithms, 3rd edition, Addison-Wesley, 1997.
[11] M. Mares, M. Straka, Linear-time ranking of permutations, in: Algorithms – ESA 2007, 2007, pp. 187–193.
[12] F. Franek, C.G. Jennings, W.F. Smyth, A simple fast hybrid pattern-matching algorithm, J. Discrete Algorithms 5 (4) (2007) 682–695.