


Fast and Linear-Time String Matching Algorithms Based on the Distances of q -Gram Occurrences

Satoshi Kobayashi

Graduate School of Information Sciences, Tohoku University, Sendai, Japan
satoshi_kobayashi@shino.ecei.tohoku.ac.jp

Diptarama Hendrian 

Graduate School of Information Sciences, Tohoku University, Sendai, Japan
diptarama@tohoku.ac.jp

Ryo Yoshinaka 

Graduate School of Information Sciences, Tohoku University, Sendai, Japan
ryoshinaka@tohoku.ac.jp

Ayumi Shinohara 

Graduate School of Information Sciences, Tohoku University, Sendai, Japan
ayumis@tohoku.ac.jp

Abstract

Given a text T of length n and a pattern P of length m , the string matching problem is a task to find all occurrences of P in T . In this study, we propose an algorithm that solves this problem in $O((n+m)q)$ time considering the distance between two adjacent occurrences of the same q -gram contained in P . We also propose a theoretical improvement of it which runs in $O(n+m)$ time, though it is not necessarily faster in practice. We compare the execution times of our and existing algorithms on various kinds of real and artificial datasets such as an English text, a genome sequence and a Fibonacci string. The experimental results show that our algorithm is as fast as the state-of-the-art algorithms in many cases, particularly when a pattern frequently appears in a text.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases String matching algorithm, text processing

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.13

Supplementary Material The implementations of our algorithms are available at <https://github.com/ushitora/distq>.

Funding *Diptarama Hendrian*: Supported by JSPS KAKENHI Grant Number JP19K20208.

1 Introduction

The exact string matching problem is a task to find all occurrences of P in T when given a text T of length n and a pattern P of length m . A brute-force solution of this problem is to compare P with all the substrings of T of length m . It takes $O(nm)$ time. The Knuth-Morris-Pratt (KMP) algorithm [17] is well known as an algorithm that can solve the problem in $O(n+m)$ time. However, it is not efficient in practice, because it scans every position of the text at least once. The Boyer-Moore algorithm [3] is famous as an algorithm that can perform string matching fast in practice by skipping many positions of the text, though it has $O(nm)$ worst-case time complexity. Like this, many efficient algorithms whose worst-case time complexity is the same or even worse than the naive method have been proposed so far [16, 21, 19]. For example, the HASH_q algorithm [19] focuses on the substrings of length q in a pattern and obtains a larger shift amount. However, considering that such an algorithm is embedded in software and actually used, if the worst-case input strings are given, the operation of the software may be slowed down. Therefore, an algorithm



© Satoshi Kobayashi, Diptarama Hendrian, Ryo Yoshinaka, and Ayumi Shinohara; licensed under Creative Commons License CC-BY

18th International Symposium on Experimental Algorithms (SEA 2020).

Editors: Simone Faro and Domenico Cantone; Article No. 13; pp. 13:1–13:13

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

that operates theoretically and practically fast is important. Franek et al. [14] proposed the Franek-Jennings-Smyth (FJS) algorithm, which is a hybrid of the KMP algorithm and the Sunday algorithm [21]. The worst-case time complexity of the FJS algorithm is $O(n + m + \sigma)$ and it works fast in practice, where σ is the alphabet size. Kobayashi et al. [18] proposed an algorithm that improves the speed of the FJS algorithm by combining a method that extends the idea of the Quite-Naive algorithm [4]. This algorithm has the same worst-case time complexity as the FJS algorithm, and it runs faster than the FJS algorithm in many cases. The LWFR q algorithm [8] is a practically fast algorithm that works in linear time. This algorithm uses a method of quickly recognizing substrings of a pattern using a hash function. See [12, 15] for recent surveys on exact string matching algorithms.

This paper proposes two new exact string matching algorithms based on the HASH q and KMP algorithms incorporating a new idea based on the distances of occurrences of the same q -grams. The time complexity of the preprocessing phase of the first algorithm is $O(mq)$ and the search phase runs in $O(nq)$ time. The second algorithm improves the theoretical complexity of the first algorithm, where the preprocessing and searching times are $O(m)$ and $O(n)$, respectively. Our algorithms are as fast as the state-of-the-art algorithms in many cases. Particularly, our algorithms work faster when a pattern frequently appears in a text.

This paper is organized as follows. Section 2 briefly reviews the KMP and HASH q algorithms, which are the basis of the proposed algorithms. Section 3 proposes our algorithms. Section 4 shows experimental results comparing the proposed algorithms with several other algorithms using artificial and practical data. Section 5 draws our conclusions.

2 Preliminaries

2.1 Notation

Let Σ be a set of characters called an *alphabet* and $\sigma = |\Sigma|$ be its size. Σ^* denotes the set of all strings over Σ . The length of a string $w \in \Sigma^*$ is denoted by $|w|$. The *empty string*, denoted by ε , is the string of length zero. The i -th character of w is denoted by $w[i]$ for each $1 \leq i \leq |w|$. The substring of w starting at i and ending at j is denoted by $w[i : j]$ for $1 \leq i \leq j \leq |w|$. For convenience, let $w[i : j] = \varepsilon$ if $i > j$. A string $w[1 : i]$ is called a *prefix* of w and a string $w[i : |w|]$ is called a *suffix* of w . A string v is a *border* of w if v is both a prefix and a suffix of w . Note that the empty string is a border of any string. Moreover, a prefix, a suffix or a border v of w is called *proper* when $v \neq w$. The length of the longest proper border of $w[1 : i]$ for $1 \leq i \leq |w|$ is given by

$$\text{Bord}_w[i] = \max\{j \mid w[1 : j] = w[i - j + 1 : i] \text{ and } 0 \leq j < i\}.$$

Throughout this paper, we assume Σ is an integer alphabet.

2.2 The exact string matching problem

The exact string matching problem is defined as follows:

Input: A text $T \in \Sigma^*$ of length n and a pattern $P \in \Sigma^*$ of length m ,

Output: All positions i such that $T[i : i + m - 1] = P$ for $1 \leq i \leq n - m + 1$.

We will use a text $T \in \Sigma^*$ of length n and a pattern $P \in \Sigma^*$ of length m throughout the paper.

Let us consider comparing $T[i : i + m - 1]$ and $P[1 : m]$. The naive method compares characters of the two strings from left to right. When a character mismatch occurs, the

■ **Algorithm 1** Computing KMP_Shift .

```

1 Function PreKMPShift( $P$ )
2    $m \leftarrow |P|$ ;  $i \leftarrow 1$ ;  $j \leftarrow 0$ ;
3    $Strong\_Bord_P[1] \leftarrow -1$ ;
4   while  $i \leq m$  do
5     while  $j > 0$  and  $P[i] \neq P[j]$  do  $j \leftarrow Strong\_Bord_P[j]$ ;
6      $i \leftarrow i + 1$ ;  $j \leftarrow j + 1$ ;
7     if  $i \leq m$  and  $P[i] = P[j]$  then  $Strong\_Bord_P[i] \leftarrow Strong\_Bord_P[j]$ ;
8     else  $Strong\_Bord_P[i] \leftarrow j$ ;
9   for  $j \leftarrow 1$  to  $m$  do  $KMP\_Shift[j] \leftarrow j - Strong\_Bord_P[j] - 1$ ;
10  return  $KMP\_Shift$ 

```

pattern is shifted to the right by one character. That is, we compare $T[i + 1 : i + m]$ and $P[1 : m]$. This naive method takes $O(nm)$ time for matching. There are a number of ideas to shift the pattern more so that searching T for P can be performed more quickly, using shifting functions obtained by preprocessing the pattern.

2.3 Knuth-Morris-Pratt algorithm

The Knuth-Morris-Pratt (KMP) algorithm [17] is well known as a string matching algorithm that has linear worst-case time complexity. When the KMP algorithm has confirmed that $T[i : i + j - 2] = P[1 : j - 1]$ and $T[i + j - 1] \neq P[j]$ for some $j \leq m$, it shifts the pattern so that a suffix of $T[i : i + j - 2]$ matches a prefix of P and we do not have to re-scan any part of $T[i : i + j - 2]$ again. That is, the pattern can be shifted by $j - k - 1$ for $k = Bord_P[j - 1]$. In addition, if $P[k + 1] = P[j]$, the same mismatch will occur again after the shift. In order to avoid this kind of mismatch, we use $Strong_Bord[1 : m + 1]$ given by

$$Strong_Bord_P(j) = \begin{cases} Bord_P(m) & \text{if } j = m + 1, \\ \max(\{k \mid P[1 : k] = P[j - k : j - 1], P[k + 1] \neq P[j], \\ \quad 0 \leq k < j\} \cup \{-1\}) & \text{otherwise.} \end{cases}$$

The amount $KMP_Shift[j]$ of the shift is given by

$$KMP_Shift[j] = j - Strong_Bord_P(j) - 1.$$

This function has a domain of $\{1, \dots, m + 1\}$ and is implemented as an array in the algorithm. Hereafter, we identify some functions and the arrays that implement them.

► **Fact 1.** *If $P[1 : j - 1] = T[i : i + j - 2]$ and $P[j] \neq T[i + j - 1]$, then $P[1 : j - k_j - 1] = T[i + k_j : i + j - 2]$ holds for $k_j = KMP_Shift[j]$. Moreover, there is no positive integer $k < KMP_Shift[j]$ such that $P = T[i + k : i + k + m - 1]$.*

Note that if the algorithm has confirmed $T[i : i + m - 1] = P$, the shift is given by $KMP_Shift[m + 1]$ after reporting the occurrence of the pattern. Algorithm 1 shows a pseudocode to compute the array KMP_Shift . It runs in $O(m)$ time. By using KMP_Shift , the KMP algorithm finds all occurrences of P in T in $O(n)$ time.

2.4 HASH q algorithm

The HASH q algorithm [19] is an adaptation of the Wu-Manber multiple string matching algorithm [22] to the single string matching problem. Before comparing P and $T[i : i + m - 1]$, the HASH q algorithm shifts the pattern so that the suffix q -gram $T[i + m - q : i + m - 1]$ of the text substring shall match the rightmost occurrence of the same q -gram in the pattern. For practical efficiency, we use a hash function, though it may result in aligning mismatching q -grams occasionally. The shift amount is given by $shift(h(T[i + m - q : i + m - 1]))$ where

$$shift(c) = m - \max(\{j \mid h(P[j - q + 1 : j]) = c, q \leq j \leq m\} \cup \{q - 1\}),$$

$$h(x) = (2^{q-1} \cdot x[1] + 2^{q-2} \cdot x[2] + \cdots + 2 \cdot x[q - 1] + x[q]) \bmod 2^8.$$

We repeatedly shift the pattern till the suffix q -grams of the pattern and the considered text substring have a matching hash value, in which case the shift amount will be 0. We then compare the characters of the pattern and the text substring from left to right. If a character mismatch occurs during the comparison, the pattern is shifted by

$$\min(\{k \mid h(P[m' - k : m - k]) = h(P[m' : m]), 1 \leq k \leq m - q\} \cup \{m'\}) \quad (1)$$

where $m' = m - q + 1$, since the q -gram suffixes of the pattern and the text substring have the same hash values. The time complexity of the preprocessing phase for computing the shift function is $O(mq)$. The searching phase has $O(n(m + q))$ time complexity. The worst-case time complexity is worse than that of the naive method, but it works fast in practice.

► **Fact 2.** *If $shift(h(T[i + m - q : i + m - 1])) = j \neq m - q + 1$, then $h(P[m - j - q + 1 : m - j]) = h(T[i + m - q : i + m - 1])$. There is no positive integer $k < j$ such that $P = T[i + k : i + k + m - 1]$.*

3 Proposed algorithms

3.1 DIST q algorithm

Our proposed algorithm uses three kinds of shifting functions. The first one HQ_Shift is essentially the same as $shift$, the one used in the HASH q algorithm, except for the hashing function. The second one $dist$ is based on the distance of the closest occurrences of the q -grams of the same hash value in the pattern. We involve KMP_Shift as the third one to guarantee the linear-time behavior.

Formally, the first shifting function is given as

$$HQ_Shift[c] = m - \max(\{j \mid h(P[j - q + 1 : j]) = c, q \leq j \leq m\} \cup \{q - 1\}),$$

where $h(x) = (4^{q-1} \cdot x[1] + 4^{q-2} \cdot x[2] + \cdots + 4 \cdot x[q - 1] + x[q]) \bmod 2^{16}$.

Fact 2 holds for HQ_Shift .

The second shifting function is defined for $j = q, \dots, m$ by

$$dist[j] = \min(\{k \mid h(P[j' - k : j - k]) = h(P[j' : j]), 1 \leq k \leq j - q\} \cup \{j'\})$$

where $j' = j - q + 1$. This function $dist$ is a generalization of the shift (Eq. 1) used in the HASH q algorithm. We have $dist[j] = k < j'$ if the q -gram ending at j and the one ending at $j - k$ have the same hash value, while no q -grams occurring between those have the same value. If no q -gram ending before j has the same hash value, then $dist[j] = j'$. By using this, in the situation where $h(P[j - q + 1 : j]) = h(T[i + j - q : i + j - 1])$, when a mismatch occurs anywhere between $T[i : i + m - 1]$ and P , the pattern can be shifted by $dist[j]$.

Algorithm 2 Computing HQ_Shift .

```

1 Function PreHqShift( $P, q$ )
2    $m \leftarrow |P|$ ;
3   for  $i \leftarrow 0$  to  $2^{16} - 1$  do
4      $HQ\_Shift[i] \leftarrow m - q + 1$ ;
5   for  $i \leftarrow q$  to  $m$  do
6      $hash \leftarrow h(P[i - q + 1 : i])$ ;
7      $HQ\_Shift[hash] \leftarrow m - i$ ;
8   return  $HQ\_Shift$ ;

```

Algorithm 3 Computing $dist$.

```

1 Function PreDistArray( $P, q$ )
2   for  $j \leftarrow 1$  to  $q - 1$  do
3      $dist[j] \leftarrow 1$ ;
4   for  $j \leftarrow 0$  to  $2^{16} - 1$  do
5      $prevpos[j] \leftarrow 0$ ;
6   for  $j \leftarrow q$  to  $|P|$  do
7      $hash \leftarrow h(P[j - q + 1 : j])$ ;
8     if  $prevpos[hash] = 0$  then  $d \leftarrow j - q + 1$ ;
9     else  $d \leftarrow j - prevpos[hash]$ ;
10     $dist[j] \leftarrow d$ ;  $prevpos[hash] \leftarrow j$ ;
11  return  $dist$ ;

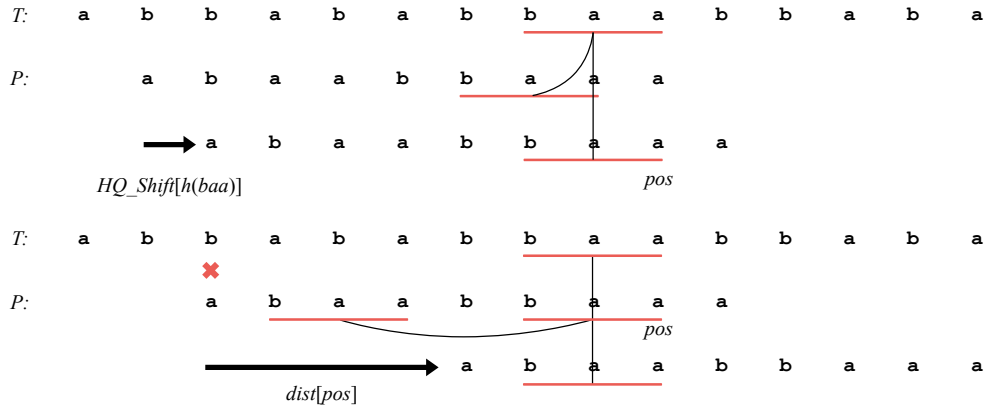
```

► **Fact 3.** Suppose that $h(P[j - q + 1 : j]) = h(T[i + j - q : i + j - 1])$. Then $h(P[j - q + 1 - dist[j] : j - dist[j]]) = h(T[i + j - q : i + j - 1])$, unless $dist[j] = j - q + 1$. Moreover, there is no positive integer $k < dist[j]$ such that $P = T[i + k : i + k + m - 1]$.

Those functions HQ_Shift , $dist$ and KMP_Shift are computed in the preprocessing phase. Algorithms 2 and 3 compute the arrays HQ_Shift and $dist$, respectively.

Figure 1 shows examples of shifting the pattern using HQ_Shift and $dist$. Both functions HQ_Shift and $dist$ shift the pattern using q -gram hash values based on Facts 2 and 3, respectively. The latter can be used only when we know that the pattern and the text substring have aligned q -grams ending at j with the same hash value and it may shift the pattern at most $j - q + 1$, while the former can be used anytime and the maximum possible shift is $m - q + 1$. The advantage of the function $dist$ is in the computational cost. If we know that the premise of Fact 3 is satisfied, we can immediately perform the shift based on $dist$, while computing $HQ_Shift(h(w))$ for the concerned q -gram w in the text is not as cheap as $dist[j]$. Our algorithm exploits this advantage of the new shifting function $dist$.

Next, we explain our searching algorithm shown in Algorithm 4. The searching phase is divided into three: *Alignment-phase*, *Comparison-phase*, and *KMP-phase*. The goal of the Alignment-phase is to shift the pattern as far as possible without comparing each single character of the pattern and the text. The Alignment-phase ends when we align the pattern and a text substring that have (a) aligned q -grams of the same hash value and (b) the same first character. Suppose P and $T[k - m + 1 : k]$ are aligned at the beginning of the Alignment-phase. If $s = HQ_Shift[h(T[k - q + 1 : k])] \leq m - q$, by shifting the pattern by s , we find the



■ **Figure 1** Shifting a pattern using HQ_Shift and $dist$.

aligned q -grams of the same hash value. Namely, $h(P[m-q-s+1 : m-s]) = h(T[k-q+1 : k])$. Otherwise, we shift the pattern by $m-q+1$ repeatedly until we find aligned q -grams of the same hash value. When finding a position pos satisfying (a) by aligning P and $T[k'-m+1 : k']$ for some k' , i.e., $h(P[pos-q+1 : pos]) = h(T[k'-m+pos-q+1 : k'-m+pos])$, we simply check the condition (b). If $P[1]$ and the corresponding text character match, we move to the Comparison-phase. Otherwise, we safely shift the pattern using $dist[pos]$. Note that although it is possible to use the function $HQ_Shift(h(T[k'-q+1 : k']))$ rather than $dist[pos]$, the computation would be more expensive. Shifting the pattern by $dist[pos]$, unless $dist[pos] = pos - q + 1$, still the pattern and the aligned text substring satisfy (a). However, we do not repeat $dist$ -shift any more, since the smaller pos becomes, the smaller the expected shift amount will be. We simply restart the Alignment-phase. Once the conditions (a) and (b) are satisfied, we move on to the Comparison-phase.

In the Comparison-phase, we check the characters from $P[2]$ to $P[m]$. If a character mismatch occurs during the comparison, either of the shift by KMP_Shift or by $dist$ is possible. Therefore, we select the one where the resumption position of the character comparison goes further to the right after shifting the pattern. If the resumption position of the comparison is the same, we select the one with the larger shift amount. Recall that when the KMP algorithm finds that $P[1 : j-1] = T[i : i+j-2]$ and $P[j] \neq T[i+j-1]$, it resumes comparison from checking the match between $T[i+j-1]$ and $P[j - KMP_Shift[j]]$ if $KMP_Shift[j] < j$, and $T[i+j]$ and $P[1]$ if $KMP_Shift[j] = j$. On the other hand, if we shift the pattern by $dist[pos]$, we simply resume matching $T[i + dist[pos]]$ and $P[1]$. Therefore, we should use $KMP_Shift[j]$ rather than $dist[pos]$ when either $KMP_Shift[j] < j$ and $dist[pos] < j - 1$ or $KMP_Shift[j] = j > dist[pos]$. Summarizing the discussion, we shift the pattern by $dist[pos]$ if $dist[pos] \geq j - 1$ and $dist[pos] \geq KMP_Shift[j]$ hold. Otherwise, we shift the pattern by $KMP_Shift[j]$. At this moment, we may have a “partial match” between the pattern and the aligned text substring. If we have performed the KMP-shift with $KMP_Shift[j] < j - 1$, then we have a match between the nonempty prefixes of the pattern and the aligned text substring of length $j - KMP_Shift[j] - 1$. In this case, we go to the KMP-phase, where we simply perform the KMP algorithm. The KMP-phase prevents the character comparison position from returning to the left and guarantees the linear time behavior of our algorithm. If we have no partial match, we return to the Alignment-phase.

► **Theorem 1.** *The worst-case time complexity of the $DIST_q$ algorithm is $O((n+m)q)$.*

■ **Algorithm 4** DIST q algorithm.

```

1 Function DIST $q(P, T, q)$ 
2    $KMP\_Shift \leftarrow \text{PreKMPShift}(P);$ 
3    $HQ\_Shift \leftarrow \text{PreHqShift}(P, q);$ 
4    $dist \leftarrow \text{PreDistArray}(P, q);$ 
5    $n \leftarrow |T|; m \leftarrow |P|; i \leftarrow 1; j \leftarrow 1; k \leftarrow m;$ 
6   while  $k \leq n$  do
7     if  $j \leq 1$  then
8       while True do // Alignment-phase
9          $sh \leftarrow HQ\_Shift[h(T[k - m + 1 : k])]; k \leftarrow k + sh;$ 
10        if  $sh \neq m - q + 1$  then
11           $pos \leftarrow m - 1 - sh;$ 
12          if  $P[1] = T[k - m + 1]$  then break;
13           $k \leftarrow k + dist[pos];$ 
14          if  $k > n$  then halt;
15         $j \leftarrow 2; i \leftarrow k - m + 2;$  // Comparison-phase
16        while  $j \leq m$  and  $P[j] = T[i]$  do
17           $i \leftarrow i + 1; j \leftarrow j + 1;$ 
18        if  $j = m + 1$  then
19          output  $i - m;$ 
20        if  $dist[pos] \geq j - 1$  and  $dist[pos] \geq KMP\_Shift[j]$  then
21           $j \leftarrow j - dist[pos];$ 
22        else
23           $j \leftarrow j - KMP\_Shift[j];$ 
24      else
25        while  $j \leq m$  and  $P[j] = T[i]$  do // KMP-phase
26           $i \leftarrow i + 1; j \leftarrow j + 1;$ 
27        if  $j = m + 1$  then
28          output  $i - m;$ 
29           $j \leftarrow j - KMP\_Shift[j];$ 
30       $k \leftarrow i + m - j;$ 

```

Proof. Since the proposed algorithm uses Fact 1 on the KMP algorithm to prevent the character comparison position from going back to the left, the number of character comparisons is at most $2n - m$ times like the KMP algorithm. In addition, the hash value of q -gram is calculated to perform the shift using HQ_Shift . Since the hash value calculation requires $O(q)$ time and it is calculated at the maximum of $n - q + 1$ places in the text, the hash value calculation takes $O(nq)$ time in total. Therefore, the worst-case time complexity of the searching phase is $O(nq)$. In the preprocessing, $O(mq)$ time is required to calculate the hash value of q -gram at $m - q + 1$ locations. ◀

▶ **Example 2.** Let $P = \text{abaabbaaa}$. The shifting functions $dist$, KMP_Shift , and HQ_Shift are shown below. The hash values are calculated by treating each character as its ASCII value, e.g. **a** is calculated as 97.

j	1	2	3	4	5	6	7	8	9	10
P	a	b	a	a	b	b	a	a	a	
$dist$	-	-	1	2	3	4	5	4	7	
KMP_Shift	1	1	3	2	4	3	7	6	7	8

x	aba	baa	aab	abb	bba	aaa	others
$h(x)$	2041	2053	2038	2042	2057	2037	
$HQ_Shift[h(x)]$	6	1	4	3	2	0	7

Figure 2 illustrates an example run of the $DIST_q$ algorithm ($q = 3$) for finding $P = \text{abaabbaaa}$ in $T = \text{abbaabbaababbabbbaaabaabaabbaaa}$.

Attempt 1 We shift the pattern by one character for $HQ_Shift[h(T[7 : 9])] = HQ_Shift[h(\text{baa})] = HQ_Shift[2053] = 1$. Since the position of the q -gram aligned by this shift is 8, pos is updated to 8.

Attempt 2 We check whether the first character of the pattern matches the corresponding character of the text. Finding $P[1] \neq T[8]$, the pattern is shifted by $dist[pos] = dist[8] = 4$.

Attempt 3 We shift the pattern by $HQ_Shift[h(T[12 : 14])] = HQ_Shift[h(\text{bba})] = 2$ and update pos to 7.

Attempt 4 We check whether the first character of the pattern matches the corresponding character of the text. From $P[1] = T[8]$, we compare the characters of $P[2 : 9]$ and $T[9 : 16]$ from left to right. Since $P[2] \neq T[9]$, the pattern is shifted by $KMP_Shift[2]$ or $dist[pos] = dist[7]$. From $KMP_Shift[2] = 1$, $dist[7] = 5$, $dist[pos] \geq 2 - 1$ and $dist[pos] \geq KMP_Shift[2]$ are satisfied. Therefore, we shift the pattern by $dist[pos] = dist[7] = 5$.

Attempt 5 We shift the pattern by $HQ_Shift[h(T[19 : 21])] = HQ_Shift[h(\text{aba})] = HQ_Shift[2041] = 6$ and update pos to 3.

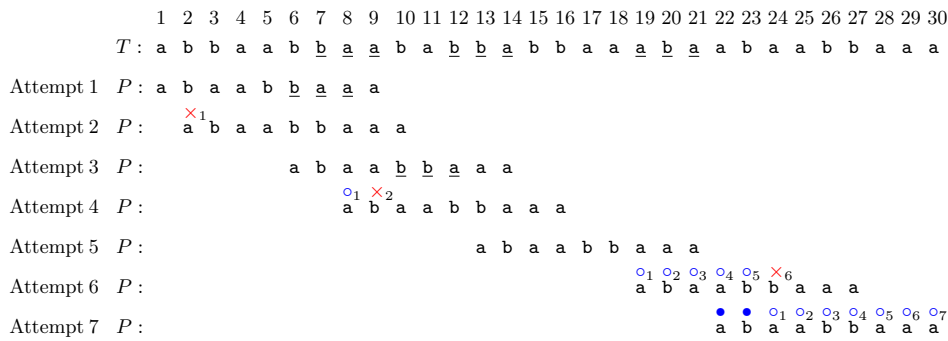
Attempt 6 We check whether the first character of the pattern matches the corresponding character of the text. By $P[1] = T[19]$, the characters of $P[2 : 9]$ and $T[20 : 27]$ are compared from left to right. Since $P[6] \neq T[24]$, the pattern is shifted by $KMP_Shift[6]$ or $dist[pos] = dist[3]$. By $KMP_Shift[6] = 3 > dist[3] = 1$, the pattern is shifted by $KMP_Shift[6] = 3$.

Attempt 7 Attempt 6 shows that $P[1 : 2] = T[22 : 23]$, that is, there is a partial match, so we continue the comparison of $T[24 : 30]$ and $P[3 : 9]$. Since $T[24 : 30] = P[3 : 9]$, the pattern occurrence position 22 is reported.

3.2 LDIST $_q$ algorithm

The LDIST $_q$ algorithm modifies the $DIST_q$ algorithm so that the worst-case time complexity is independent of q . In the $DIST_q$ algorithm, if strings such as $T = \mathbf{a}^n$ and $P = \mathbf{ba}^{m-1}$ are given, $O(nq)$ time is required for searching phase because the hash values of each q -gram are calculated in the text. Since the hash function h defined in Section 3.1 is a rolling hash, if the hash value of $w[i : i + q - 1]$ has already been obtained for a string w , the hash value of $w[i + 1 : i + q]$ can be computed in constant time by $h(w[i + 1 : i + q]) = (4 \cdot (h(w[i : i + q - 1]) - 4^{q-1} \cdot w[i]) + w[i + q]) \bmod 2^{16}$. The LDIST $_q$ algorithm modifies Line 9 of Algorithm 4 so that we calculate the hash value of the q -gram using the previously calculated value of the other q -gram in the incremental way, if they overlap. Similarly, the time complexity of the preprocessing phase can be reduced.

► **Theorem 3.** *The worst-case time complexity of the LDIST $_q$ algorithm is $O(n + m)$.*



■ **Figure 2** An example run of the DIST_q algorithm for a pattern $P = \text{abaabbaaa}$ and a text $T = \text{abbaabbaababbabbbaabaabaabbaaa}$. For each alignment of the pattern, \circ and \times indicate a match and a mismatch between the text and the pattern, respectively. The character with \bullet is known to match the character at the corresponding position in the text without comparison. Subscript numbers show the order of character comparisons in each attempt.

Proof. Like the DIST_q algorithm, we compare characters at most $2n - m$ times. To calculate the hash value of a q -gram, if it is overlapped with the q -gram for which the hash value has been calculated one step before, the incremental update is performed using the rolling hash. Therefore, the calculation of the hash value of q -grams takes $O(n)$ time in total. Thus, the worst-case time complexity of matching is $O(n)$. Calculating the hash values of q -grams in the preprocess is performed in the same way, so it is done in $O(m)$ time. ◀

4 Experiments

In this section, we compare the execution times of the proposed algorithms with the existing algorithms listed below, where algorithms that run in linear time in the input string size are marked with \star .

- BNDM_q [20]: Backward Nondeterministic DAWG Matching algorithm using q -grams with $q = 2, 4$ and 6 ,
- SBNDM_q [1]: Simplified version of the Backward Nondeterministic DAWG Matching algorithm using q -grams with $q = 2, 4, 6$ and 8 ,
- KBNDM [7]: Factorized variant of the BNDM algorithm,
- BSDM_q [10]: Backward SNR DAWG Matching algorithm using condensed alphabets with groups of q characters with $1 \leq q \leq 8$,
- $\star\text{FJS}$ [14]: Franek-Jennings-Smyth algorithm,
- $\star\text{FJS}+$ [18]: Modification of the FJS algorithm,
- HASH_q [19]: Hashing algorithm using q -grams with $2 \leq q \leq 8$ (see Section 2.4),
- $\text{FS-}w$ [11]: Multiple Windows version of the Fast Search algorithm [5] implemented using w sliding windows with $w = 1, 2, 4, 6$ and 8 ,
- IOM [6]: Improved Occurrence Matcher,
- WOM [6]: Worst Occurrence Matcher,
- SKIP_q [9]: Skip-Search algorithm using q -grams with $2 \leq q \leq 8$,
- WFR_q [8]: Weak-Factors-Recognition algorithm implemented with a q -chained loop with $2 \leq q \leq 8$,
- $\star\text{LWFR}_q$ [8]: Linear-Weak-Factors-Recognition algorithm implemented with a q -chained loop with $2 \leq q \leq 8$,
- $\star\text{DIST}_q$: Our algorithm proposed in Section 3.1 (Algorithm 4) with $1 \leq q \leq 8$,
- $\star\text{LDIST}_q$: Our algorithm proposed in Section 3.2 with $1 \leq q \leq 8$.

All algorithms are implemented in C language, compiled by GCC 9.2.0 with the optimization option `-O3`. We used the implementations in SMART [13] for all algorithms except for the FJS, FJS+ and our algorithms. The implementations of our algorithms are available at <https://github.com/ushitora/distq>. We experimented with the following strings:

1. Genome sequence (Table 1): the genome sequence of *E. coli* of length $n = 4641652$ with $\sigma = 4$, from NCBI¹. The patterns are randomly extracted from T of length $m = 2, 4, 8, 16, 32, 64, 128, 256, 512$ and 1024. We measured the total running time of 25 executions.
2. English text (Table 2): the King James version of the Bible of length $n = 4017009$ with $\sigma = 62$, from the Large Canterbury Corpus² [2]. We removed the line breaks from the text. The patterns are randomly extracted from T of length $m = 2, 4, 8, 16, 32, 64, 128, 256, 512$ and 1024. We measured the total running time of 25 executions.
3. Fibonacci string (Table 3): generated by the following recurrence

$$Fib_1 = \mathbf{b}, Fib_2 = \mathbf{a} \text{ and } Fib_k = Fib_{k-1} \cdot Fib_{k-2} \text{ for } k > 2.$$

The text is fixed to $T = Fib_{32}$ of length $n = 2178309$. The patterns are randomly extracted from T of length $m = 2, 4, 8, 16, 32, 64, 128, 256, 512$ and 1024. We measured the total running time of 100 executions.

4. Texts with frequent pattern occurrences (Tables 4, 5): generated by intentionally embedding a lot of patterns. We embedded $occ = 0, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536$, and 131072 occurrences of a pattern of length $m = 8$ into a text of length $n = 4000000$ over an alphabet of size $\sigma = 4$ and 95. More specifically, we first randomly generate a pattern and a provisional text, which may contain the pattern. Then we randomly change characters of the text until the pattern does not occur in the text. Finally we embed the pattern occ times at random positions without overlapping. We measured the total running time of 25 executions.

The best performance among three trials is recorded for each experiment. For the algorithms using parameter q or w , we report only the best results. The value of q or w giving the best performance is shown in round brackets.

Experimental results show that when the pattern is short, the SBNDM q and BSDM q algorithms have good performance in general. For the genome sequence text, WFR q , LWFR q and our algorithms are the fastest algorithms except when the pattern is very short. On the English text, SBNDM q and LWFR q run fastest for short and long patterns, respectively. On the other hand, DIST q runs almost as fast as the best algorithm on both short and long patterns. In fact, it runs faster than SBDDM q and LWFR q for long and short patterns, respectively. In the experiments on the Fibonacci string, the FJS algorithm and our algorithms have shown good results as the pattern length increases. Differently from the previous two sorts of texts, our algorithms clearly outperformed the LWFR q algorithm. Since the Fibonacci strings have many repeating structures and patterns are randomly extracted from the text, the number of occurrences of the pattern is very large in this experiment. Therefore, we hypothesize that the efficiency of DIST q algorithms does not decrease when the number of pattern occurrences is large. We fixed the pattern length and alphabet size and prepared data with the number of pattern occurrences intentionally changed. From the experimental result, it is found that our algorithms become more advantageous as the number of pattern occurrences increases. The results show that the LDIST q algorithm is generally slower than the DIST q algorithm. This should be due to the overhead of the process of determining whether to update the hash value difference by the rolling hash in the LDIST q algorithm.

¹ https://www.ncbi.nlm.nih.gov/genome/167?genome_assembly_id=161521

² <http://corpus.canterbury.ac.nz/>

■ **Table 1** Genome sequence ($\sigma = 4, n = 4641652$).

m	2	4	8	16	32	64	128	256	512	1024
BNDM q	218.34 ⁽²⁾	174.55 ⁽²⁾	84.14 ⁽⁴⁾	64.89 ⁽⁶⁾	60.39 ⁽⁶⁾	61.07 ⁽⁶⁾	60.85 ⁽⁶⁾	62.17 ⁽⁶⁾	61.20 ⁽⁶⁾	60.16 ⁽⁶⁾
SBNDM q	179.90⁽²⁾	154.20 ⁽²⁾	80.94 ⁽⁴⁾	73.29 ⁽⁶⁾	57.10 ⁽⁸⁾	61.01 ⁽⁶⁾	61.74 ⁽⁶⁾	61.20 ⁽⁶⁾	61.67 ⁽⁶⁾	60.80 ⁽⁶⁾
KBNDM	311.78	201.99	150.15	113.84	83.23	67.83	75.65	75.39	76.58	74.72
BSDM q	195.38 ⁽²⁾	118.86⁽³⁾	84.23 ⁽⁵⁾	63.03 ⁽⁶⁾	61.26 ⁽⁶⁾	58.75 ⁽⁶⁾	57.50 ⁽⁷⁾	56.99 ⁽⁶⁾	57.01 ⁽⁶⁾	56.58 ⁽⁶⁾
*FJS	407.02	353.60	311.96	279.13	308.42	297.00	266.12	317.79	317.34	296.19
*FJS+	388.44	296.70	203.03	171.17	149.52	136.59	128.55	130.39	122.51	112.99
HASH q	571.44 ⁽²⁾	272.86 ⁽³⁾	126.00 ⁽³⁾	88.12 ⁽³⁾	68.22 ⁽³⁾	58.84 ⁽⁶⁾	55.09 ⁽⁶⁾	59.48 ⁽⁷⁾	57.34 ⁽⁷⁾	57.96 ⁽⁷⁾
FS- w	332.32 ⁽⁴⁾	245.99 ⁽⁴⁾	184.72 ⁽⁴⁾	158.72 ⁽⁴⁾	143.79 ⁽⁶⁾	125.05 ⁽⁴⁾	123.52 ⁽⁶⁾	117.90 ⁽⁶⁾	108.03 ⁽⁶⁾	100.72 ⁽⁶⁾
IOM	377.25	275.36	215.72	220.97	219.86	218.12	210.61	221.31	230.15	211.69
WOM	381.54	301.46	220.34	182.30	166.27	143.24	136.20	133.75	127.40	114.74
SKIP q	250.89 ⁽²⁾	136.18 ⁽³⁾	91.51 ⁽⁴⁾	63.96 ⁽⁶⁾	56.79 ⁽⁷⁾	53.09 ⁽⁷⁾	52.10 ⁽⁷⁾	57.12 ⁽⁷⁾	56.97 ⁽⁸⁾	58.00 ⁽⁶⁾
WFR q	219.50 ⁽²⁾	168.39 ⁽²⁾	88.86 ⁽⁴⁾	65.82 ⁽⁴⁾	57.19 ⁽⁸⁾	55.12 ⁽²⁾	51.77 ⁽³⁾	50.04 ⁽³⁾	55.02 ⁽⁶⁾	54.64⁽⁸⁾
*LWFR q	216.10 ⁽²⁾	173.48 ⁽³⁾	88.71 ⁽⁴⁾	60.75 ⁽⁵⁾	53.84⁽⁵⁾	50.48⁽⁶⁾	49.65⁽⁸⁾	48.71 ⁽⁶⁾	54.90 ⁽⁶⁾	54.97 ⁽⁷⁾
*DIST q	186.10 ⁽²⁾	125.44 ⁽³⁾	78.56⁽⁴⁾	60.48⁽⁵⁾	55.21 ⁽⁶⁾	52.05 ⁽⁷⁾	51.26 ⁽⁸⁾	50.44 ⁽⁸⁾	54.81 ⁽⁷⁾	55.58 ⁽⁸⁾
*LDIST q	295.55 ⁽²⁾	181.99 ⁽³⁾	86.58 ⁽⁴⁾	65.29 ⁽⁶⁾	56.74 ⁽⁶⁾	52.31 ⁽⁶⁾	50.39 ⁽⁶⁾	48.70⁽⁷⁾	54.33⁽⁴⁾	55.22 ⁽⁷⁾

■ **Table 2** English text ($\sigma = 62, n = 4017009$).

m	2	4	8	16	32	64	128	256	512	1024
BNDM q	140.48 ⁽²⁾	93.39 ⁽²⁾	70.84 ⁽⁴⁾	54.10 ⁽⁴⁾	45.88⁽⁴⁾	47.61 ⁽⁴⁾	47.64 ⁽⁴⁾	46.88 ⁽⁴⁾	47.40 ⁽⁶⁾	45.58 ⁽⁴⁾
SBNDM q	108.32⁽²⁾	73.50⁽²⁾	64.87⁽²⁾	50.73⁽⁴⁾	47.85 ⁽⁴⁾	48.45 ⁽⁴⁾	48.55 ⁽⁴⁾	47.56 ⁽⁴⁾	47.08 ⁽⁴⁾	45.98 ⁽⁴⁾
KBNDM	192.76	126.56	92.03	71.04	64.17	56.79	49.56	49.09	50.24	47.68
BSDM q	117.22 ⁽²⁾	79.96 ⁽²⁾	70.36 ⁽³⁾	57.72 ⁽⁴⁾	49.91 ⁽⁸⁾	48.00 ⁽⁸⁾	44.99 ⁽⁸⁾	43.62 ⁽⁸⁾	43.06 ⁽⁸⁾	42.70 ⁽⁸⁾
*FJS	192.52	158.61	106.40	89.04	78.43	68.14	64.80	61.15	60.48	55.38
*FJS+	196.88	155.86	101.77	77.74	67.40	60.31	54.77	52.44	51.62	47.74
HASH q	312.91 ⁽²⁾	218.95 ⁽²⁾	107.38 ⁽²⁾	70.85 ⁽²⁾	56.90 ⁽³⁾	49.33 ⁽⁶⁾	46.63 ⁽⁵⁾	45.27 ⁽⁸⁾	44.77 ⁽³⁾	42.98 ⁽⁷⁾
FS- w	129.50 ⁽⁶⁾	104.45 ⁽⁶⁾	71.57 ⁽⁶⁾	61.08 ⁽⁴⁾	54.19 ⁽⁶⁾	49.53 ⁽⁴⁾	48.95 ⁽⁶⁾	47.02 ⁽⁴⁾	46.96 ⁽⁶⁾	43.00 ⁽⁴⁾
IOM	193.37	148.51	104.36	86.22	75.09	69.08	63.63	60.62	58.83	51.76
WOM	199.23	153.81	112.45	86.61	75.26	62.80	60.54	62.74	58.91	55.65
SKIP q	161.38 ⁽²⁾	100.52 ⁽²⁾	72.33 ⁽³⁾	55.71 ⁽⁴⁾	49.06 ⁽⁴⁾	48.73 ⁽⁴⁾	49.87 ⁽⁴⁾	48.81 ⁽⁸⁾	45.75 ⁽²⁾	45.32 ⁽²⁾
WFR q	137.40 ⁽²⁾	85.22 ⁽²⁾	68.88 ⁽²⁾	52.85 ⁽⁵⁾	46.67 ⁽⁵⁾	44.39⁽⁸⁾	42.09⁽⁸⁾	41.66 ⁽⁵⁾	41.65 ⁽⁶⁾	40.53 ⁽²⁾
*LWFR q	121.08 ⁽²⁾	85.47 ⁽²⁾	70.38 ⁽²⁾	53.83 ⁽³⁾	47.89 ⁽⁵⁾	44.49 ⁽⁵⁾	42.38 ⁽⁶⁾	41.09⁽⁸⁾	41.23⁽⁸⁾	40.30⁽⁸⁾
*DIST q	115.61 ⁽²⁾	80.14 ⁽²⁾	65.84 ⁽³⁾	52.25 ⁽⁴⁾	48.13 ⁽⁴⁾	46.26 ⁽⁴⁾	43.32 ⁽⁴⁾	42.84 ⁽⁸⁾	42.98 ⁽⁴⁾	41.47 ⁽⁷⁾
*LDIST q	229.62 ⁽²⁾	102.15 ⁽²⁾	69.70 ⁽³⁾	55.34 ⁽⁴⁾	49.46 ⁽⁵⁾	45.93 ⁽⁵⁾	44.37 ⁽³⁾	43.15 ⁽⁴⁾	42.21 ⁽⁵⁾	41.11 ⁽⁷⁾

■ **Table 3** Fibonacci string ($\sigma = 2, n = 2178309$).

m	2	4	8	16	32	64	128	256	512	1024
BNDM q	343.25 ⁽²⁾	308.44 ⁽²⁾	283.26 ⁽⁴⁾	257.64 ⁽⁶⁾	233.94 ⁽⁶⁾	285.63 ⁽⁴⁾	284.37 ⁽⁴⁾	293.00 ⁽⁴⁾	307.82 ⁽⁴⁾	315.47 ⁽⁴⁾
SBNDM q	286.02⁽²⁾	292.15⁽²⁾	272.98 ⁽⁴⁾	276.35 ⁽⁶⁾	306.42 ⁽⁶⁾	372.03 ⁽⁶⁾	432.53 ⁽⁶⁾	493.20 ⁽⁶⁾	546.94 ⁽⁶⁾	602.09 ⁽⁶⁾
KBNDM	541.70	405.78	411.08	422.85	382.25	402.45	425.60	437.67	461.07	451.12
BSDM q	482.47 ⁽²⁾	500.43 ⁽³⁾	397.29 ⁽⁵⁾	362.52 ⁽⁸⁾	330.76 ⁽⁶⁾	736.89 ⁽¹⁾	766.57 ⁽¹⁾	782.98 ⁽¹⁾	790.26 ⁽¹⁾	508.80 ⁽³⁾
*FJS	402.44	362.23	276.97	237.87	218.38	206.07	203.86	202.94	196.49	194.01
*FJS+	456.20	396.04	335.70	319.93	295.64	300.36	296.48	295.37	288.56	289.00
HASH q	645.48 ⁽²⁾	406.70 ⁽²⁾	257.69⁽⁴⁾	251.91 ⁽⁷⁾	279.81 ⁽⁷⁾	344.99 ⁽⁷⁾	415.16 ⁽⁷⁾	470.71 ⁽⁷⁾	514.05 ⁽⁷⁾	579.57 ⁽⁷⁾
FS- w	383.23 ⁽¹⁾	396.23 ⁽¹⁾	347.72 ⁽¹⁾	289.15 ⁽¹⁾	253.36 ⁽¹⁾	246.82 ⁽¹⁾	248.73 ⁽¹⁾	235.66 ⁽¹⁾	235.35 ⁽¹⁾	230.61 ⁽¹⁾
IOM	381.92	414.42	453.54	497.84	543.93	641.13	751.42	839.92	899.19	1019.59
WOM	552.38	555.43	564.67	617.93	664.47	732.05	852.06	926.35	1036.31	1126.17
SKIP q	470.93 ⁽²⁾	394.09 ⁽²⁾	332.66 ⁽⁵⁾	336.16 ⁽⁸⁾	374.91 ⁽⁸⁾	464.23 ⁽³⁾	460.54 ⁽³⁾	450.18 ⁽³⁾	451.05 ⁽³⁾	464.13 ⁽³⁾
WFR q	442.32 ⁽²⁾	497.95 ⁽³⁾	528.45 ⁽³⁾	652.48 ⁽⁶⁾	2132.38 ⁽⁷⁾	3762.19 ⁽⁸⁾	6762.67 ⁽⁸⁾	12624.63 ⁽⁸⁾	24416.65 ⁽⁸⁾	48596.02 ⁽⁸⁾
*LWFR q	552.64 ⁽²⁾	504.77 ⁽³⁾	428.34 ⁽⁵⁾	342.80 ⁽⁷⁾	297.07 ⁽⁷⁾	304.57 ⁽²⁾	274.47 ⁽⁶⁾	265.28 ⁽⁶⁾	258.06 ⁽⁶⁾	254.92 ⁽⁶⁾
*DIST q	438.96 ⁽¹⁾	359.56 ⁽²⁾	293.80 ⁽²⁾	235.61⁽²⁾	213.07⁽⁷⁾	206.92 ⁽²⁾	201.18 ⁽⁸⁾	196.56 ⁽⁵⁾	194.86 ⁽⁴⁾	193.62 ⁽⁵⁾
*LDIST q	565.13 ⁽²⁾	412.15 ⁽³⁾	300.98 ⁽⁴⁾	245.38 ⁽⁷⁾	215.89 ⁽⁸⁾	208.40 ⁽⁷⁾	200.45⁽⁴⁾	193.74⁽⁴⁾	190.85⁽³⁾	193.48⁽⁸⁾

■ **Table 4** Texts with frequent pattern occurrences ($\sigma = 4$, $n = 4000000$, $m = 8$).

<i>occ</i>	0	128	256	512	1024	2048	4096	8192	16384	32768	65536	131072
BNDM $_q$	73.53 ⁽⁴⁾	72.28 ⁽⁴⁾	72.87 ⁽⁴⁾	73.89 ⁽⁴⁾	74.39 ⁽⁴⁾	75.75 ⁽⁴⁾	77.60 ⁽⁴⁾	81.63 ⁽⁴⁾	82.67 ⁽⁴⁾	88.98 ⁽⁴⁾	108.56 ⁽⁴⁾	146.13 ⁽⁶⁾
SBNDM $_q$	68.58⁽⁴⁾	68.92⁽⁴⁾	71.17 ⁽⁴⁾	77.26 ⁽⁴⁾	81.04 ⁽⁴⁾	80.76 ⁽⁴⁾	78.33 ⁽⁴⁾	82.82 ⁽⁴⁾	91.01 ⁽⁴⁾	96.83 ⁽⁴⁾	111.75 ⁽⁴⁾	140.21 ⁽⁴⁾
KBNDM	127.73	128.41	128.41	126.92	128.74	131.17	129.74	135.70	140.99	152.46	164.59	186.32
BSDM $_q$	69.85 ⁽⁴⁾	69.04 ⁽⁴⁾	68.85⁽⁴⁾	69.19⁽⁴⁾	70.63⁽⁴⁾	72.00⁽⁴⁾	72.48⁽⁴⁾	76.25⁽⁴⁾	79.95⁽⁴⁾	89.91 ⁽⁴⁾	110.22 ⁽⁴⁾	143.91 ⁽⁴⁾
*FJS	246.26	253.35	263.94	247.73	255.94	276.44	262.83	256.36	251.33	269.39	259.93	251.06
*FJS+	174.38	173.86	180.46	173.05	178.65	182.98	177.56	181.17	182.48	190.71	197.69	199.60
HASH $_q$	121.09 ⁽³⁾	121.36 ⁽³⁾	122.64 ⁽³⁾	122.50 ⁽³⁾	119.49 ⁽³⁾	123.98 ⁽³⁾	124.31 ⁽³⁾	124.93 ⁽³⁾	126.81 ⁽³⁾	128.94 ⁽³⁾	143.75 ⁽³⁾	153.08 ⁽⁴⁾
FS- w	154.89 ⁽⁴⁾	155.56 ⁽⁴⁾	165.18 ⁽⁴⁾	156.67 ⁽⁴⁾	159.03 ⁽⁴⁾	166.44 ⁽⁴⁾	165.62 ⁽⁴⁾	160.46 ⁽⁴⁾	167.46 ⁽⁴⁾	178.53 ⁽²⁾	185.86 ⁽²⁾	191.38 ⁽²⁾
IOM	185.31	181.07	195.53	187.98	194.18	200.99	195.98	195.89	197.67	202.61	214.90	209.52
WOM	192.59	192.28	207.57	189.21	196.02	203.86	196.98	199.79	203.59	215.79	219.94	230.59
SKIP $_q$	79.32 ⁽⁴⁾	77.14 ⁽⁴⁾	79.19 ⁽⁴⁾	82.08 ⁽⁴⁾	81.82 ⁽⁴⁾	82.55 ⁽⁴⁾	83.83 ⁽⁴⁾	87.08 ⁽⁴⁾	89.82 ⁽⁴⁾	93.09 ⁽⁴⁾	106.41 ⁽⁴⁾	126.22 ⁽³⁾
WFR $_q$	76.68 ⁽⁴⁾	76.38 ⁽⁴⁾	77.63 ⁽⁴⁾	83.21 ⁽⁴⁾	80.93 ⁽⁴⁾	81.42 ⁽⁴⁾	83.86 ⁽⁴⁾	88.55 ⁽⁴⁾	93.16 ⁽⁴⁾	106.07 ⁽⁴⁾	123.90 ⁽⁴⁾	164.77 ⁽⁴⁾
*LWFR $_q$	76.99 ⁽⁴⁾	76.33 ⁽⁴⁾	78.83 ⁽⁴⁾	77.57 ⁽⁴⁾	78.74 ⁽⁴⁾	76.46 ⁽⁴⁾	84.65 ⁽⁴⁾	89.87 ⁽⁴⁾	96.17 ⁽⁴⁾	108.67 ⁽⁴⁾	129.35 ⁽³⁾	168.70 ⁽³⁾
*DIST $_q$	69.67 ⁽⁴⁾	73.38 ⁽⁴⁾	74.35 ⁽⁴⁾	74.31 ⁽⁴⁾	75.12 ⁽⁴⁾	74.96 ⁽⁴⁾	77.21 ⁽⁴⁾	77.56 ⁽⁴⁾	80.09 ⁽⁴⁾	86.25⁽⁴⁾	101.12⁽⁴⁾	120.98⁽³⁾
*LDIST $_q$	75.82 ⁽⁴⁾	74.45 ⁽⁴⁾	74.89 ⁽⁴⁾	76.77 ⁽⁴⁾	74.62 ⁽⁴⁾	75.47 ⁽⁴⁾	77.10 ⁽⁴⁾	80.18 ⁽⁴⁾	83.40 ⁽⁴⁾	88.86 ⁽⁴⁾	103.73 ⁽⁴⁾	122.27 ⁽⁴⁾

■ **Table 5** Texts with frequent pattern occurrences ($\sigma = 95$, $n = 4000000$, $m = 8$).

<i>occ</i>	0	128	256	512	1024	2048	4096	8192	16384	32768	65536	131072
BNDM $_q$	56.58 ⁽²⁾	55.41 ⁽²⁾	56.26 ⁽²⁾	56.89 ⁽²⁾	56.68 ⁽²⁾	57.42 ⁽²⁾	58.80 ⁽²⁾	60.72 ⁽²⁾	67.37 ⁽²⁾	74.57 ⁽²⁾	98.52 ⁽²⁾	125.65 ⁽²⁾
SBNDM $_q$	52.76⁽²⁾	52.37⁽²⁾	53.09⁽²⁾	53.23⁽²⁾	55.56 ⁽²⁾	52.82⁽²⁾	56.41 ⁽²⁾	56.50⁽²⁾	61.58 ⁽²⁾	69.96 ⁽²⁾	91.72 ⁽²⁾	112.41 ⁽²⁾
KBNDM	81.98	80.22	77.61	79.42	79.55	81.88	82.61	82.25	89.19	98.65	111.59	142.65
BSDM $_q$	54.89 ⁽²⁾	56.07 ⁽²⁾	54.55 ⁽²⁾	55.99 ⁽²⁾	55.78 ⁽²⁾	56.78 ⁽²⁾	58.38 ⁽²⁾	61.48 ⁽²⁾	66.76 ⁽²⁾	76.66 ⁽³⁾	96.79 ⁽³⁾	131.76 ⁽³⁾
*FJS	81.93	81.41	81.14	81.83	82.04	82.16	82.10	84.30	86.02	90.72	100.41	111.55
*FJS+	80.26	84.27	81.05	80.58	80.21	80.84	83.52	85.66	87.40	94.03	102.16	112.23
HASH $_q$	101.34 ⁽²⁾	103.34 ⁽²⁾	102.15 ⁽²⁾	105.02 ⁽²⁾	101.55 ⁽²⁾	104.07 ⁽²⁾	105.84 ⁽²⁾	107.46 ⁽²⁾	107.79 ⁽²⁾	112.15 ⁽²⁾	118.52 ⁽²⁾	126.53 ⁽²⁾
FS- w	52.77 ⁽⁸⁾	52.42 ⁽⁶⁾	53.51 ⁽⁶⁾	53.74 ⁽⁶⁾	53.07⁽⁶⁾	53.95 ⁽⁶⁾	55.33⁽⁶⁾	57.61 ⁽⁶⁾	61.36⁽⁶⁾	68.86 ⁽⁶⁾	81.66 ⁽⁴⁾	102.11 ⁽²⁾
IOM	82.07	83.86	84.89	85.58	82.64	82.45	83.90	86.98	87.28	91.63	99.30	106.37
WOM	84.37	84.42	86.39	85.05	85.36	85.35	86.20	86.74	90.22	93.31	105.36	114.68
SKIP $_q$	60.93 ⁽²⁾	59.34 ⁽²⁾	63.66 ⁽³⁾	62.97 ⁽³⁾	62.11 ⁽²⁾	62.42 ⁽²⁾	64.08 ⁽²⁾	65.12 ⁽²⁾	69.11 ⁽²⁾	76.18 ⁽³⁾	83.05 ⁽³⁾	101.43 ⁽³⁾
WFR $_q$	59.39 ⁽²⁾	59.57 ⁽²⁾	59.24 ⁽²⁾	60.12 ⁽²⁾	61.16 ⁽²⁾	55.51 ⁽²⁾	58.24 ⁽²⁾	62.38 ⁽²⁾	68.35 ⁽²⁾	81.99 ⁽²⁾	104.92 ⁽²⁾	139.15 ⁽²⁾
*LWFR $_q$	55.99 ⁽²⁾	58.63 ⁽²⁾	54.16 ⁽²⁾	53.75 ⁽²⁾	56.99 ⁽²⁾	60.87 ⁽²⁾	58.45 ⁽²⁾	63.39 ⁽²⁾	72.48 ⁽²⁾	85.61 ⁽²⁾	112.47 ⁽³⁾	152.54 ⁽³⁾
*DIST $_q$	58.60 ⁽²⁾	59.30 ⁽²⁾	58.94 ⁽²⁾	58.46 ⁽²⁾	58.47 ⁽³⁾	59.91 ⁽²⁾	61.55 ⁽²⁾	62.29 ⁽²⁾	65.21 ⁽²⁾	65.99⁽²⁾	77.36⁽²⁾	95.05⁽²⁾
*LDIST $_q$	62.56 ⁽³⁾	61.52 ⁽²⁾	66.62 ⁽³⁾	66.90 ⁽²⁾	67.28 ⁽³⁾	63.66 ⁽²⁾	65.22 ⁽²⁾	67.15 ⁽²⁾	67.59 ⁽²⁾	73.93 ⁽²⁾	85.10 ⁽²⁾	100.33 ⁽²⁾

5 Conclusion

We proposed two new algorithms for the exact string matching problem: the DIST $_q$ algorithm and the LDIST $_q$ algorithm. We confirmed that our algorithms are as efficient as the state-of-the-art algorithms in many cases. Particularly when a pattern frequently appears in a text, our algorithms outperformed existing algorithms. The DIST $_q$ algorithm runs in $O(q(n+m))$ time and the LDIST $_q$ algorithm runs in $O(n+m)$ time. Their performances were not significantly different in our experiments and rather the former ran faster than the latter in most cases, where the optimal value of q was relatively small.

References

- 1 Cyril Allauzen, Maxime Crochemore, and Mathieu Raffinot. Factor oracle: A new structure for pattern matching. In Jan Pavelka, Gerard Tel, and Miroslav Bartošek, editors, *SOFSEM'99: Theory and Practice of Informatics*, pages 295–310. Springer Berlin Heidelberg, 1999.
- 2 Ross Arnold and Tim Bell. A corpus for the evaluation of lossless compression algorithms. In *Proceedings of DCC '97. Data Compression Conference*, pages 201–210, 1997.
- 3 Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977. doi:10.1145/359842.359859.
- 4 Domenico Cantone and Simone Faro. Searching for a substring with constant extra-space complexity. In *Proceedings of Third International Conference on Fun with algorithms*, pages 118–131, 2004.

- 5 Domenico Cantone and Simone Faro. Fast-search algorithms: New efficient variants of the Boyer-Moore pattern-matching algorithm. *Journal of Automata, Languages and Combinatorics*, 10:589–608, 2005. doi:10.1007/3-540-44867-5_4.
- 6 Domenico Cantone and Simone Faro. Improved and self-tuned occurrence heuristics. *Journal of Discrete Algorithms*, 28:73–84, 2014. doi:10.1016/j.jda.2014.07.006.
- 7 Domenico Cantone, Simone Faro, and Emanuele Giaquinta. A compact representation of nondeterministic (suffix) automata for the bit-parallel approach. *Information and Computation*, 213:3–12, 2012. Special Issue: Combinatorial Pattern Matching (CPM 2010). doi:10.1016/j.ic.2011.03.006.
- 8 Domenico Cantone, Simone Faro, and Arianna Pavone. Linear and Efficient String Matching Algorithms Based on Weak Factor Recognition. *Journal of Experimental Algorithmics*, 24(1):1–20, 2019. doi:10.1145/3301295.
- 9 Simone Faro. A very fast string matching algorithm based on condensed alphabets. In Riccardo Dondi, Guillaume Fertin, and Giancarlo Mauri, editors, *Algorithmic Aspects in Information and Management - 11th International Conference, AAIM 2016, Bergamo, Italy, July 18-20, 2016, Proceedings*, volume 9778 of *Lecture Notes in Computer Science*, pages 65–76. Springer, 2016. doi:10.1007/978-3-319-41168-2_6.
- 10 Simone Faro and Thierry Lecroq. A fast suffix automata based algorithm for exact online string matching. In Nelma Moreira and Rogério Reis, editors, *Implementation and Application of Automata*, pages 149–158. Springer Berlin Heidelberg, 2012.
- 11 Simone Faro and Thierry Lecroq. A multiple sliding windows approach to speed up string matching algorithms. In Ralf Klasing, editor, *Experimental Algorithms*, pages 172–183. Springer Berlin Heidelberg, 2012.
- 12 Simone Faro and Thierry Lecroq. The exact online string matching problem. *ACM Computing Surveys*, 45(2):1–42, 2013. doi:10.1145/2431211.2431212.
- 13 Simone Faro, Thierry Lecroq, Stefano Borzì, Simone Di Mauro, and Alessandro Maggio. The string matching algorithms research tool. In Jan Holub and Jan Žďárek, editors, *Proceedings of the Prague Stringology Conference 2016*, pages 99–113, Czech Technical University in Prague, Czech Republic, 2016.
- 14 Frantisek Franek, Christopher G. Jennings, and W.F. Smyth. A simple fast hybrid pattern-matching algorithm. *Journal of Discrete Algorithms*, 5(4):682–695, 2007. doi:10.1016/J.JDA.2006.11.004.
- 15 Saqib I. Hakak, Amirrudin Kamsin, Palaiahnakote Shivakumara, Gulshan A. Gilkar, Wazir Z. Khan, and Muhammad Imran. Exact string matching algorithms: Survey, issues, and future research directions. *IEEE Access*, 7:69614–69637, 2019.
- 16 R. Nigel Horspool. Practical fast searching in strings. *Software: Practice and Experience*, 10(6):501–506, 1980. doi:10.1002/spe.4380100608.
- 17 Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977. doi:10.1137/0206024.
- 18 Satoshi Kobayashi, Diptarama Hendrian, Ryo Yoshinaka, and Ayumi Shinohara. An improvement of the Franek-Jennings-Smyth pattern matching algorithm. In *Proceedings of the Prague Stringology Conference 2019*, pages 56–68, 2019.
- 19 Thierry Lecroq. Fast exact string matching algorithms. *Information Processing Letters*, 102(6):229–235, 2007. doi:10.1016/j.ipl.2007.01.002.
- 20 Gonzalo Navarro and Mathieu Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In Martin Farach-Colton, editor, *Combinatorial Pattern Matching*, pages 14–33. Springer Berlin Heidelberg, 1998.
- 21 Daniel M. Sunday and Daniel M. A very fast substring search algorithm. *Communications of the ACM*, 33(8):132–142, 1990. doi:10.1145/79173.79184.
- 22 Sun Wu and Udi Manber. A fast algorithm for multi-pattern searching. Technical Report TR-94-17, Department of Computer Science, Chung-Cheng University, 1994.