An Improvement of the Franek-Jennings-Smyth Pattern Matching Algorithm

Satoshi Kobayashi, Diptarama Hendrian, Ryo Yoshinaka, and Ayumi Shinohara

Graduate School of Information Sciences, Tohoku University
6-6-05 Aramaki Aza Aoba, Aoba-ku, Sendai, Japan
satoshi_kobayashi@shino.ecei.tohoku.ac.jp
{diptarama, ryoshinaka, ayumis}@tohoku.ac.jp

Abstract. In this paper, we propose a new pattern matching algorithm based on the Franek-Jennings-Smyth (FJS) algorithm. The FJS algorithm is a hybrid of the Knuth-Morris-Pratt (KMP) and the Sunday algorithms. The worst case time complexity of the KMP algorithm is linear time and the Sunday algorithm is quadratic time. However, the Sunday algorithm is faster than the KMP algorithm in practice. Inheriting the virtues of those algorithms, the FJS algorithm runs in linear time in the worst case and fast in practice. We improve the FJS algorithm by further taking an idea inspired by the Quite-Naive algorithm by Cantone and Faro. The experimental results show that our algorithm is faster than the FJS algorithm in general except when a pattern is extremely short.

Keywords: exact pattern matching, Knuth-Morris-Pratt algorithm, Sunday algorithm

1 Introduction

The pattern matching problem is a very fundamental problem in string processing. Given a text T of length n and a pattern P of length m, the pattern matching problem is a task to find all occurrences of P in T. A naive solution of this problem is to compare P with all the substrings of T of length m, which takes $O(mn)$ time. One approach to perform pattern matching more efficiently is to avoid comparing position pairs of the pattern and the text as much as possible based on the property of the pattern obtained by preprocessing it. Among the previously proposed pattern matching algorithms in this approach, the Knuth-Morris-Pratt (KMP) algorithm [9] is well known, which exploits the periodicity of prefixes of P to perform pattern matching in $O(n)$ time with $O(m)$ preprocessing time. On the other hand, the Boyer-Moore (BM) algorithm [2] is famous as an algorithm that can perform pattern matching fast in practice while it has $O(nm)$ worst case time complexity. The BM algorithm uses occurrence positions of each symbol in P and periodicity of suffixes of P rather than prefixes.

Many BM-type algorithms have been proposed. Typical BM-type algorithms are the Horspool algorithm [8] and the Sunday algorithm [10]. The Horspool algorithm is a simpler version of the BM algorithm. The Sunday algorithm is a slight improvement of the Horspool algorithm and known to be practically fast. While the BM algorithm checks characters from right to left of the pattern, the Sunday algorithm can check the characters of the pattern in any order. Franek et al. [7] proposed a hybrid algorithm of the KMP and the Sunday algorithms called the Franek-Jennings-Smyth (FJS) algorithm. Inheriting the virtues of those algorithms, the FJS algorithm runs in linear time in the worst case and fast in practice. A comprehensive survey of the exact online string matching problem is written by Faro and Lecroq [5].

Satoshi Kobayashi, Diptarama Hendrian, Ryo Yoshinaka, Ayumi Shinohara: An Improvement of the Franek-Jennings-Smyth Pattern Matching Algorithm, pp. 56–68.
Proceedings of PSC 2019, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06618-8

In this paper, we propose a new pattern matching algorithm based on the FJS algorithm. The FJS algorithm uses the shift functions of the KMP and Sunday algorithms. Our algorithm additionally uses a new shift function inspired by the Quite-Naive algorithm by Cantone and Faro [3]. The time complexity of the preprocessing phase of our algorithm is $O(m+\sigma)$ and the search phase runs in $O(n)$ time, where σ denotes the alphabet size. Our algorithm is faster than the FJS algorithm in general except when a pattern is extremely short. It is not faster than the state-of-the-art in general, but it is effective when a pattern frequently appears in a text.

This paper is organized as follows. Section 2 briefly reviews the KMP, Sunday, and FJS algorithms, which are the basis of the proposed algorithm. Section 3 proposes our algorithm. Section 4 shows experimental results comparing the proposed algorithm with several other algorithms using artificial and real data.

2 Preliminaries

2.1 Notation

Let Σ be a set of characters called an *alphabet* and $\sigma = |\Sigma|$ be the size of the alphabet. 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 $x = w[1:i]$ is called a *prefix* of w and a string $z = w[i:|w|]$ is called a *suffix* of w. In particular, a prefix x (resp. suffix z) of w is a *proper prefix* (resp. *proper suffix*) of w when $x \neq w$ (resp. $z \neq w$). 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, it is a *proper border* of w if $v \neq w$. The length of the longest proper border of $w[1:i]$ for $1 \leq i \leq |w|$ is given by

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

The exact pattern 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 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 shift functions obtained by preprocessing the pattern.

2.2 Knuth-Morris-Pratt algorithm

The Knuth-Morris-Pratt (KMP) algorithm [9] is a pattern matching algorithm that has linear worst case time complexity. When the KMP algorithm has confirmed that

Algorithm 1: Calculating *KMP_Shift*

```
1  Function PreKMPShift(P)
2      m ← |P|;
3      i ← 1; j ← 0;
4      KMP_Shift[1] = 1;
5      while i ≤ m do
6          while j > 0 and P[i] ≠ P[j] do
7              j ← j − KMP_Shift[j];
8          i ← i + 1; j ← j + 1;
9          if i ≤ m and P[i] = P[j] then
10             KMP_Shift[i] ← i − (j − KMP_Shift[j])
11         else
12             KMP_Shift[i] ← i − j
13     return KMP_Shift
```

$T[i : i+j-2] = P[1 : j-1]$ 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. Thus, the pattern can be shifted by $j-k-1$ for $k = \mathit{Bord}_P[j-1]$. However, if $P[k+1] = P[j]$, the same mismatch occurs again after the shift. In order to avoid this kind of mismatch, we use $\mathit{Strong\_Bord}[1 : m+1]$ given by

$$\mathit{Strong\_Bord}_P[j] = \begin{cases} -1 & \text{if } j = 1, \\ \mathit{Strong\_Bord}_P[k] & \text{if } j \leq m \text{ and } P[k+1] = P[j], \\ k & \text{otherwise,} \end{cases}$$

where $k = \mathit{Bord}_P[j-1]$. The amount $\mathit{KMP\_Shift}[j]$ of the shift is given by

$$\mathit{KMP\_Shift}[j] = j - \mathit{Strong\_Bord}_P[j] - 1.$$

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 $k_j = \mathit{KMP\_Shift}[j]$. Moreover, there is no positive integer $k < \mathit{KMP\_Shift}[j]$ 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 $\mathit{KMP\_Shift}[m+1]$ after reporting the occurrence of the pattern. Algorithm 1 shows pseudocode to compute the array *KMP_Shift*. Clearly, it runs in $O(m)$ time. By using *KMP_Shift*, the KMP algorithm finds all occurrences of P in T in $O(n)$ time.

2.3 Sunday algorithm

In the Sunday algorithm [10], the amount of the shift when a mismatch occurs between P and $T[i : i+m-1]$ depends on the character $T[i+m] \in \Sigma$. It shifts the pattern so that the character $T[i+m]$ will match the rightmost occurrence of the same character in P. If $T[i+m]$ does not occur in P, by skipping the position $i+m$ of T, we check whether $T[i+m+1 : i+2m] = P$. The preprocessing phase of the Sunday algorithm calculates an array *Sunday_Shift* of size σ given by

$$\mathit{Sunday\_Shift}[c] = m + 1 - \max(\{\, i \mid P[i] = c \,\} \cup \{0\})$$

for $c \in \Sigma$.

Algorithm 2: Calculating *Sunday_Shift*

```
1 Function PreSundayShift(P, Σ)
2     m ← |P|;
3     for c in Σ do
4         Sunday_Shift[c] ← m + 1;
5     for i ← 1 to m do
6         Sunday_Shift[P[i]] ← m − i + 1;
7     return Sunday_Shift;
```

Fact 2 *For any i, then there is no positive integer $k < Sunday\_Shift[T[i+m]]$ such that $P = T[i+k : i+k+m-1]$.*

In principle, the order of comparison of characters in P and $T[i : i+m-1]$ is arbitrary. Wherever a mismatch is found, the Sunday algorithm shifts the pattern by $Sunday\_Shift[T[i+m]]$. Algorithm 2 shows pseudocode to calculate the array *Sunday_Shift*, which runs in $O(m+\sigma)$ time. Although the searching time is $O(nm)$ in the worst case, it is practically very fast for large alphabets.

2.4 Franek-Jennings-Smyth algorithm

The Franek-Jennings-Smyth (FJS) algorithm [7] (Algorithm 3) is a hybrid of the KMP and Sunday algorithms. It computes both arrays *KMP_Shift* and *Sunday_Shift* in the preprocessing phase. In the matching phase, it shifts a pattern using the better array depending on the situation where a mismatch has been found. We call the shift based on *KMP_Shift* and *Sunday_Shift* the KMP-shift and the Sunday-shift, respectively. Practically using the Sunday-shift is very effective, but it does not guarantee a linear time execution if we consistently use the Sunday-shift when we have found some *partial match* between the pattern and a text substring, i.e., when it has been confirmed that some nonempty prefixes of the pattern and the text substring match. When a partial match has been found, the FJS algorithm uses the KMP-shift so that it runs in linear time.

When comparing the pattern P with a text substring $T[i : i+m-1]$, the FJS algorithm checks whether the last characters of the pattern and the substring match. If $P[m] \neq T[i+m-1]$, it performs the Sunday-shift repeatedly until we find a position in T that has the character $P[m]$. We call this procedure (**while** loop at Line 8 in Algorithm 3) the *Sunday-phase*. Once we find a position j such that if $P[m] = T[j+m-1]$, we go into the *KMP-phase*, where we use the KMP-shift. As long as the algorithm sees a partial match between the pattern and a text substring, it behaves just like the KMP algorithm. Otherwise, it goes back to the Sunday-phase.

The preprocessing time and the searching time of the algorithm are $O(m+\sigma)$ and $O(n)$, respectively.

Example 1. The arrays *KMP_Shift* and *Sunday_Shift* for $P =$ `abaaca` are calculated as follows:

j	1	2	3	4	5	6	7
$P[j]$	a	b	a	a	c	a	
$KMP\_Shift[j]$	1	1	3	2	3	6	5

c	a	b	c
$Sunday\_Shift[c]$	1	5	2

Algorithm 3: The FJS algorithm

```
1  Function FJS(P, T, Σ)
2      KMP_Shift ← PreKMPShift(P);
3      Sunday_Shift ← PreSundayShift(P, Σ);
4      n ← |T|; m ← |P|;
5      i ← 1; j ← 1; ip ← m;
6      while ip ≤ n do
7          if j ≤ 1 then                                   // No partial match
8              while P[m] ≠ T[ip] do
9                  ip ← ip + Sunday_Shift[T[ip + 1]];
10                 if ip > n then
11                     halt;
12             j ← 1; i ← ip − m + 1;
13             while j < m and P[j] = T[i] do
14                 i ← i + 1; j ← j + 1;
15             if j = m then
16                 i ← i + 1; j ← j + 1;
17                 output i − m;
18             j ← j − KMP_Shift[j];
19         else                                            // Partial match is found
20             while j ≤ m and P[j] = T[i] do
21                 i ← i + 1; j ← j + 1;
22             if j = m + 1 then
23                 output i − m;
24             j ← j − KMP_Shift[j];
25         ip ← i + m − j;
```

Figure 1 illustrates an example run of the FJS algorithm for finding P in $T =$ `abababcababbbca`.

Attempt 1 The first attempt compares the character at the end of P with the character at the corresponding position in T. Since $P[6] \neq T[6]$, the pattern is shifted by $Sunday\_Shift[T[7]] = Sunday\_Shift[\mathtt{c}] = 2$

Attempt 2 Since $P[6] = T[8]$, we compare the characters from left to right. The first mismatch occurs at $P[4] \neq T[6]$, so the pattern is shifted by $KMP\_Shift[4] = 2$.

Attempt 3 Because there is a partial match $P[1:1] = T[5:5]$ due to *KMP_Shift* of Attempt 2, the character comparison is started from $P[2]$. A mismatch $P[3] \neq T[7]$ occurs, so the pattern is shifted by $KMP\_Shift[3] = 3$.

Attempt 4 Since there is no partial match, we compare the last character of the pattern $P[6]$ with $T[13]$. Since $P[6] \neq T[13]$, the pattern is shifted by $Sunday\_Shift[T[14]] = Sunday\_Shift[\mathtt{c}] = 2$.

Attempt 5 Since $P[6] = T[15]$, we compare the characters of the pattern from left to right. A mismatch $P[3] \neq T[12]$ occurs and the search ends.

3 Proposed algorithm

In this section, we propose to improve the FJS algorithm by introducing another shift function. To make the shift amount bigger on Line 18 of Algorithm 3, we employ and

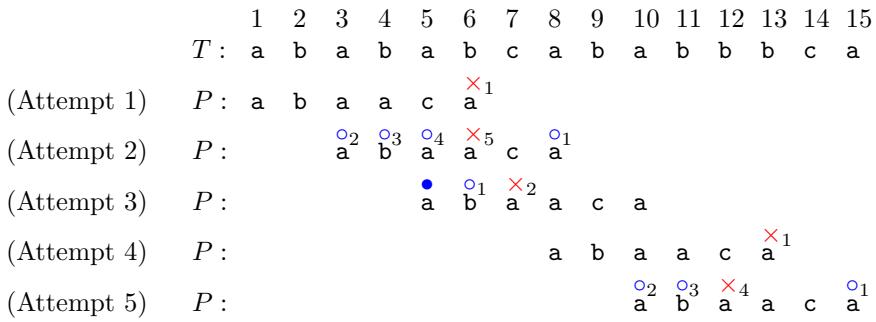

Figure 1. An example run of the FJS algorithm for a pattern P = `abaaca` and a text T = `abababcababbbca`. For each alignment of the pattern, ○ and × indicate a match and a mismatch between the text and the pattern, respectively. The character with • is known to match the character at the corresponding position in the text without comparison. Subscript numbers show the order of character comparisons in each attempt.

modify the shift function used in Cantone and Faro's Quite-Naive algorithm [3]. Their algorithm compares the pattern P and a text substring $T[i : i+m-1]$ from right to left. When a mismatch has been found in the middle after at least it has been confirmed that $P[m] = T[i+m-1]$,[1] the pattern is shifted by δ so that $P[m-\delta] = T[i+m-1]$ hold if $P[m]$ occurs in $P[1 : m-1]$. The shift amount is given by $\delta = \min(\{\, j \mid P[m-j] = P[m],\ 1 \le j < m \,\} \cup \{m\})$, which is the distance between the rightmost and the second rightmost occurrences of the rightmost character. If $P[m]$ does not occur in $P[1 : m-1]$, we shift $\delta = m$ to skip the position $T[i+m-1]$.

We generalize their idea to make the shift amount bigger by focusing on characters as well as the last one in P. Define an array d of size m by

$$d[i] = \min(\{\, j \mid P[i-j] = P[i],\ 1 \le j < i \,\} \cup \{i\})$$

for $1 \le i \le m$. Obviously $d[m]$ is the shift amount used in the Quite-Naive algorithm. We take the maximum value md and the position mdp that gives md by

$$md = \max_{1 \le j \le m} d[j],$$
$$mdp = \arg\max_{1 \le j \le m} d[j].$$

If more than one position gives the maximum value md, we simply take the largest index as mdp.

Fact 3 *If $P[mdp] = T[i+mdp-1]$, then there is no positive integer $k < md$ such that $P = T[i+k : i+k+m-1]$.*

While the FJS algorithm uses the rightmost character of P for triggering the Sunday-shift, we use mdp in the Sunday-phase. That is, when comparing P and $T[i : i+m-1]$, we first check whether $P[mdp] = T[i+mdp-1]$ holds. If $P[mdp] \neq T[i+mdp-1]$, we perform the Sunday-shift like the FJS algorithm. This is not an improvement from the FJS algorithm, since, as we have explained in Section 2.3, the shift amount is always determined by the character $T[i+m]$. On the other hand, when we have $P[mdp] = T[i+mdp-1]$, we compare the corresponding characters of

[1] The Quite-Naive algorithm uses a different shift amount in other situations, which we refrain from explaining in this paper.

Algorithm 4: Calculating *md* and *mdp*

```
1  Function GetMdp(P, Σ)
2      md ← 0, mdp ← 1;
3      for c in Σ do
4          prevpos[c] ← 0;
5      for j ← 1 to |P| do
6          if md ≤ j − prevpos[P[j]] then
7              md ← j − prevpos[P[j]];
8              mdp ← j;
9          prevpos[P[j]] ← j;
10     return mdp, md;
```

P and $T[i : i+m-1]$ from left to right. When a mismatch is found in the middle, we may perform the KMP-shift, just like the FJS algorithm does. Yet, it is also possible to shift the pattern by md by taking the advantage of the knowledge $P[mdp] = T[i+mdp-1]$, due to Fact 3. We take the bigger one between md and $KMP\_Shift[j]$, provided that the choice guarantees the theoretical linear time performance of the algorithm. 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 md, we simply start matching $T[i+md]$ and $P[1]$. Therefore, we should use $KMP\_Shift[j]$ rather than md when either $KMP\_Shift[j] < j$ and $md < j-1$ or $KMP\_Shift[j] = j > md$. Summarizing the discussion, we obtain the following shift function:

$$MAX\_Shift[j] = \begin{cases} md & \text{if } md \geq \max\{KMP\_Shift[j],\ j-1\}, \\ KMP\_Shift[j] & \text{otherwise,} \end{cases}$$

where $1 \leq j \leq m+1$. When the shift amount is $MAX\_Shift[j] = md$, we have no partial match between P and the substring of T at the alignment obtained by the shift and thus go to the Sunday-phase. If $MAX\_Shift[j] = KMP\_Shift[j]$, our algorithm behaves just like the FJS algorithm.

We replace *KMP_Shift* on Line 18 of Algorithm 3 by our shift function *MAX_Shift*. On the other hand, the other occurrence of *KMP_Shift* on Line 24 of Algorithm 3 cannot be replaced by *MAX_Shift*, where $P[mdp] = T[i+mdp-1]$ is not guaranteed. Algorithms 4 and 5 calculate the values mdp, md and the shift function *MAX_Shift* in $O(m+|\Sigma|)$ and $O(m)$ time, respectively. Our proposed searching algorithm is shown as Algorithm 6, where differences from Algorithm 3 are highlighted. The correctness of our algorithm is supported by Facts 1, 2 and 3. Clearly it runs in linear in $O(n)$ time apart from the preprocessing phase.

Example 2. We use the same pattern and text as Example 1 for an example run of our algorithm. The arrays *KMP_Shift*, *MAX_Shift* and *Sunday_Shift* for $P =$ `abaaca` are calculated as follows. We have $md = 5$ and $mdp = 5$ for $d[5] = \max_{1 \leq j \leq 6} d[j] = 5$.

Algorithm 5: Calculating *MAX_Shift*

```
1  Function PreMaxShift(P, KMP_Shift, md)
2      for j ← 1 to |P| + 1 do
3          if md ≥ j − 1 and md ≥ KMP_Shift[j] then
4              MAX_Shift[j] ← md;
5          else
6              MAX_Shift[j] ← KMP_Shift[j];
7      return MAX_Shift;
```

Algorithm 6: The proposed algorithm

```
1  Function ImprovedFJS(P, T, Σ)
2      KMP_Shift ← PreKMPShift(P);
3      mdp, md ← GetMdp(P, Σ);
4      MAX_Shift ← PreMaxShift(P, KMP_Shift, md);
5      Sunday_Shift ← PreSundayShift(P, Σ);
6      n ← |T|; m ← |P|;
7      i ← 1; j ← 1; ip ← m;
8      while ip ≤ n do
9          if j ≤ 1 then                                   // No partial match
10             while P[mdp] ≠ T[ip − (m − mdp)] do
11                 ip ← ip + Sunday_Shift[T[ip + 1]];
12                 if ip > n then
13                     halt;
14             j ← 1; i ← ip − m + 1;
15             while j ≤ m and P[j] = T[i] do
16                 i ← i + 1; j ← j + 1;
17             if j = m + 1 then
18                 output i − m;
19             j ← j − MAX_Shift[j];
20         else                                            // Partial match is found
21             while j ≤ m and P[j] = T[i] do
22                 i ← i + 1; j ← j + 1;
23             if j = m + 1 then
24                 output i − m;
25             j ← j − KMP_Shift[j];
26         ip ← i + m − j;
```

j	1	2	3	4	**5**	6	7
$P[j]$	a	b	a	a	c	a	
$d[j]$	1	2	2	1	**5**	2	
$KMP\_Shift[j]$	1	1	3	2	3	6	5
$MAX\_Shift[j]$	5	5	5	5	5	6	5

c	a	b	c
$Sunday\_Shift[c]$	1	5	2

Figure 2 shows an example run of our algorithm finding P in $T =$ `abababcababbbca`.

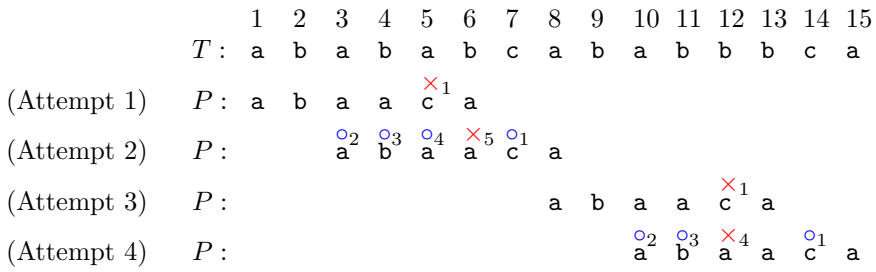

Figure 2. Running our algorithm for $P =$ `abaaca` and $T =$ `abababcababbbca`

Table 1. Algorithms used for the experiment

Name	Description
KMP	Knuth-Morris-Pratt [9]
QS	Quick-Search [10]
FJS	Franek-Jennings-Smyth [7]
LWFR2	Linear-Weak-Factor-Recognition implemented with a 2-chained-loop [4]
Ours	Proposed algorithm

Attempt 1 First we compare $P[mdp]$ with the corresponding text character $T[mdp]$. Since $P[5] \neq T[5]$, we shift the pattern by $\mathit{Sunday\_Shift}[T[7]] = \mathit{Sunday\_Shift}[\mathtt{c}] = 2$.

Attempt 2 Again, $P[mdp]$ is compared with the character at the corresponding position of the text. Because $P[5] = T[7]$, the letters of the pattern are compared from left to right. Finding a mismatch $P[4] \neq T[6]$, the pattern is shifted by $\mathit{MAX\_Shift}[4] = md = 5$.

Attempt 3 Since we are aware of no partial match at this alignment, we compare $P[mdp] = P[5]$ with $T[12]$. By $P[5] \neq T[12]$, the pattern is shifted by $\mathit{Sunday\_Shift}[T[14]] = \mathit{Sunday\_Shift}[\mathtt{c}] = 2$.

Attempt 4 We compare $P[mdp] = P[5]$ and $T[14]$. For $P[5] = T[14]$, we compare the characters of the pattern from left to right and find a mismatch $P[3] \neq T[12]$. The shift amount lets the pattern go beyond the end of the text and thus the algorithm halts.

4 Experiments

In this section, we compare the execution times of the proposed algorithm with several other algorithms using various texts and patterns. Table 1 shows the algorithms we used. The experiments were performed on AOBA [11], an integrated online platform evaluating string processing algorithms, with a PC with Xeon E3-1220 V2, 8 GB RAM, Ubuntu 16.04 and Docker 18.09.0. The evaluations are executed on a Docker container, and resources provided by the sandbox are limited to 1 CPU and 1 GB RAM. We used the implementations in SMART [6] for all algorithms except for our and the FJS algorithms, with modification to conform to the AOBA execution format. The implementation of the FJS algorithm we used is the original [7].[2] All implementations are in the C language, compiled using GCC 5.4.0 with the optimization option `-O3`. We used the best performance result among three trials for each experiment.

[2] The SMART implementation of the FJS algorithm is in error.

4.1 Random strings

We prepared a random text of length $n = 1000000$ and random patterns of length $m = 2, 4, 8, 16, 32, 64, 128, 256$ and 512 over alphabets of size $\sigma = 2, 4, 8, 16, 32, 64$ and 95. We measured the total time of 100 runs. Table 2 compares the performance of our algorithm with the ones in Table 1. Our algorithm runs faster than the FJS algorithm in most cases due to the bigger shift on Line 19 of Algorithm 6 than that on Line 18 of Algorithm 3. The difference becomes quite significant when the alphabet is rather small, since both algorithms easily exit the Sunday-phase and execute those lines more often. Exceptional cases, where ours is a little slower than the FJS algorithm, are when the pattern is extremely short. Recall that after the FJS algorithm confirms $P[m] = T[i+m-1]$ in the Sunday-phase, it does not compare $P[m]$ and $T[i+m-1]$ any more in the succeeding KMP-phase, while our algorithm may compare the same positions twice on Lines 10 and 15 in Algorithm 6. The redundancy will be relatively large when the pattern is extremely short. On the other hand, LWFR2 is the fastest for almost all random data.

4.2 Artificial strings

We also experimented with the following strings.

1. Fibonacci strings are generated by the following recurrence:

$$Fib_1 = \mathtt{b},\ Fib_2 = \mathtt{a} \text{ and } Fib_n = Fib_{n-1} \cdot Fib_{n-2} \text{ for } n > 2.$$

   The text is fixed to $T = Fib_{32}$ of length $n = 2178309$, and the patterns are randomly extracted from T of length $m = 2, 4, 8, 16, 32, 64, 128$ and 256. Fibonacci strings are known to be highly repetitive. We measured the total time after 100 executions.
2. Texts with frequent pattern occurrences are generated by intentionally embedding a lot of patterns. We embedded 32768 occurrences of a pattern of length $m = 2, 4, 8, 16, 32$ and 64 into a text of length $n = 4000000$ over an alphabet of size $\sigma = 8$. 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 at random positions without overlapping. We measured the total time after 25 executions.

Our proposed algorithm is the fastest for the Fibonacci string (Table 3). Since patterns appear so frequently in a Fibonacci string, algorithms that perform verification after hash-filtering, such as LWFR, could be slow. Ours performed the best for texts with frequent occurrences of long patterns (Table 4). Therefore, our algorithm seem to work efficiently when patterns frequently appear in the text.

4.3 Practical strings

We also made similar measurements on practical data. The total time of 25 runs was measured. We used the following data as texts.

1. Genome sequence: the genome sequence of E. coli of length $n = 4641652$ with $\sigma = 4$, from NCBI[3].

[3] `https://www.ncbi.nlm.nih.gov/genome/167?genome_assembly_id=161521`

Table 2. The execution time of the algorithms in Table 1 using the random strings

$\sigma = 2$

m	2	4	8	16	32	64	128	256	512
KMP	815.84	816.46	813.49	814.84	817.05	815.29	816.96	817.65	816.14
QS	669.15	709.66	735.67	761.96	775.29	763.89	774.25	729.14	754.32
LWFR2	629.74	**549.76**	**342.04**	**238.98**	**188.72**	**169.70**	**158.24**	**151.89**	**151.89**
FJS	**608.69**	629.12	693.13	757.15	759.19	757.78	763.55	737.30	755.06
Ours	689.81	603.26	520.00	467.80	444.09	408.40	386.28	361.21	344.48

$\sigma = 4$

m	2	4	8	16	32	64	128	256	512
KMP	725.49	746.48	750.65	746.55	742.36	748.35	746.22	751.83	746.07
QS	544.44	441.19	376.97	343.46	350.00	341.49	353.08	360.16	344.43
LWFR2	**329.47**	**273.56**	**239.52**	**193.17**	**162.70**	**153.32**	**148.54**	**146.78**	**144.95**
FJS	532.69	477.87	445.55	418.53	425.92	417.98	431.15	439.43	420.17
Ours	547.44	408.54	329.00	275.47	255.39	241.96	233.55	227.61	220.60

$\sigma = 8$

m	2	4	8	16	32	64	128	256	512
KMP	591.71	589.70	588.05	588.54	585.03	589.40	589.90	590.50	588.08
QS	406.26	318.84	256.1	224.52	217.62	217.83	217.91	219.07	219.32
LWFR2	**292.49**	**213.36**	**189.96**	**181.20**	**166.41**	**151.78**	**148.68**	**146.11**	**146.44**
FJS	393.45	322.11	269.27	240.90	236.04	235.12	234.38	237.45	236.64
Ours	395.28	300.18	243.03	211.71	196.09	190.36	184.00	184.13	181.70

$\sigma = 16$

m	2	4	8	16	32	64	128	256	512
KMP	499.51	493.94	495.27	490.56	492.42	490.36	491.20	493.83	492.84
QS	345.50	269.98	219.15	190.88	178.53	174.37	173.40	174.93	174.68
LWFR2	**260.03**	**193.25**	**172.27**	**164.07**	**160.09**	**153.35**	**146.72**	**144.97**	**144.88**
FJS	327.49	263.10	217.15	192.90	182.34	178.80	177.77	178.92	178.82
Ours	327.14	256.38	211.15	185.85	174.03	168.53	165.59	165.38	163.69

$\sigma = 32$

m	2	4	8	16	32	64	128	256	512
KMP	440.08	436.27	435.02	433.58	436.51	434.87	433.98	435.30	435.02
QS	319.45	249.25	202.80	176.55	164.43	158.70	157.47	157.51	157.20
LWFR2	**247.72**	**182.07**	**164.72**	**155.89**	**153.35**	**152.02**	**148.13**	**145.45**	**145.00**
FJS	299.48	238.50	198.54	175.64	165.54	162.61	161.32	160.53	161.08
Ours	299.54	236.41	196.95	173.16	162.98	156.66	154.97	154.47	155.43

$\sigma = 64$

m	2	4	8	16	32	64	128	256	512
KMP	405.14	403.63	405.03	403.81	404.46	402.91	402.73	403.97	404.15
QS	307.45	239.90	197.00	171.20	158.06	153.09	149.10	148.35	148.46
LWFR2	**243.00**	**177.85**	**158.95**	**151.39**	**148.76**	**147.38**	**147.05**	**146.53**	**145.62**
FJS	282.46	229.53	190.89	169.34	157.98	155.70	153.75	153.41	151.86
Ours	284.43	226.99	188.94	167.47	157.18	152.22	149.44	148.78	149.27

$\sigma = 95$

m	2	4	8	16	32	64	128	256	512
KMP	394.58	394.42	392.13	394.58	393.01	392.00	394.32	395.56	393.35
QS	303.62	237.59	194.73	169.84	155.15	151.36	147.39	145.69	**145.23**
LWFR2	**241.08**	**176.17**	**157.90**	**150.18**	**147.23**	**146.94**	**145.41**	**145.59**	146.12
FJS	278.63	224.24	188.01	166.81	155.56	153.84	151.51	150.28	149.10
Ours	280.46	224.78	187.13	167.33	155.23	149.86	148.36	147.49	146.93

Table 3. The execution time of the algorithms in Table 1 using the Fibonacci strings

m	2	4	8	16	32	64
KMP	806.02	782.25	771.26	783.94	733.25	656.00
QS	809.51	824.72	835.41	954.10	1119.16	1280.31
LWFR2	842.78	968.64	847.70	774.92	676.61	648.86
FJS	**616.63**	582.88	505.26	449.33	425.88	427.02
Ours	821.58	**581.54**	**472.51**	**405.69**	**377.28**	**372.04**

Table 4. The execution time of the algorithms in Table 1 using the texts with 32768 occurrences of a pattern

m	2	4	8	16	32	64
KMP	578.23	609.99	605.00	591.71	554.09	476.73
QS	397.98	340.59	285.62	259.85	267.48	284.01
LWFR2	**271.54**	**242.86**	**236.93**	**236.70**	244.92	291.22
FJS	383.25	340.95	291.77	261.48	251.58	234.06
Ours	380.58	325.98	272.63	240.43	**233.28**	**229.03**

Table 5. The execution time of the algorithms in Table 1 using the genome sequences

m	2	4	8	16	32	64
KMP	841.13	860.16	836.45	869.12	869.73	847.45
QS	630.21	501.58	417.28	398.56	387.16	399.61
LWFR2	**373.87**	**318.24**	**257.32**	**214.21**	**178.39**	**166.65**
FJS	604.83	541.89	488.88	480.37	465.15	478.47
Ours	619.65	462.60	359.56	310.54	285.58	272.14

Table 6. The execution time of the algorithms in Table 1 using the English texts

m	2	4	8	16	32	64
KMP	528.59	516.48	534.79	514.70	515.38	533.08
QS	377.53	288.36	237.96	199.98	184.02	172.70
LWFR2	**270.67**	**197.57**	**178.91**	**169.13**	**163.86**	**157.35**
FJS	351.93	286.69	229.52	202.25	184.61	174.37
Ours	359.03	292.18	221.54	190.36	175.27	166.16

2. English text: the King James version of the Bible of length $n = 4017009$ with $\sigma = 62$, from the Large Canterbury Corpus[4] [1]. We removed the line break from the text.

In both cases, the patterns are randomly extracted from the text of length $m = 2, 4, 8, 16, 32, 64, 128$ and 256.

Table 5 shows that our algorithm is faster than the FJS algorithm for genome sequences. For English texts, Table 6 shows that our and the FJS algorithm have almost the same speed, probably because the Sunday-shift is dominant. For both texts, LWFR2 is the fastest for all pattern lengths.

[4] `http://corpus.canterbury.ac.nz/`

5 Conclusion

We propose a new algorithm modifying the FJS algorithm. Our experimental results show that it runs faster than the FJS algorithm in general except when a pattern is extremely short. Moreover, our algorithm outperformed representative existing algorithms for data where patterns appear frequently in text.

Acknowledgment

This work is partially supported by JSPS KAKENHI Grant Numbers JP19K20208 and JP15H05706.

References

1. R. Arnold and T. Bell: *A corpus for the evaluation of lossless compression algorithms*, in Proceedings DCC '97. Data Compression Conference, 1997, pp. 201–210.
2. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Commun. ACM, 20(10) Oct. 1977, pp. 762–772.
3. D. Cantone and S. Faro: *Searching for a substring with constant extra-space complexity*, in Proc. of Third International Conference on Fun with algorithms, 2004, pp. 118–131.
4. D. Cantone, S. Faro, and A. Pavone: *Linear and Efficient String Matching Algorithms Based on Weak Factor Recognition.* Journal of Experimental Algorithmics, 24(1) feb 2019, pp. 1–20.
5. S. Faro and T. Lecroq: *The exact online string matching problem: A review of the most recent results.* ACM Comput. Surv., 45(2) 2013, pp. 13:1–13:42.
6. S. Faro, T. Lecroq, S. Borzì, S. D. Mauro, and A. Maggio: *The string matching algorithms research tool*, in Proceedings of the Prague Stringology Conference 2016, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2016, pp. 99–113.
7. F. Franek, C. G. Jennings, and W. Smyth: *A simple fast hybrid pattern-matching algorithm.* Journal of Discrete Algorithms, 5(4) dec 2007, pp. 682–695.
8. R. N. Horspool: *Practical fast searching in strings.* Software: Practice and Experience, 10(6) jun 1980, pp. 501–506.
9. D. E. Knuth, J. H. Morris Jr., and V. R. Pratt: *Fast pattern matching in strings.* SIAM Journal on Computing, 6(2) 1977, pp. 323–350.
10. D. M. Sunday: *A very fast substring search algorithm.* Communications of the ACM, 33(8) aug 1990, pp. 132–142.
11. R. Wakimoto, S. Kobayashi, Y. Igarashi, J. Davaajav, D. Hendrian, R. Yoshinaka, and A. Shinohara: *AOBA: An online benchmark tool for algorithms in stringology*, 2019, http://aoba.iss.is.tohoku.ac.jp/.