IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received November 14, 2019, accepted January 13, 2020, date of publication January 23, 2020, date of current version February 6, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2969038

Simple and Efficient Pattern Matching Algorithms for Biological Sequences

PEYMAN NEAMATOLLAHI, MONTASSIR HADI, AND MAHMOUD NAGHIBZADEH, (Senior Member, IEEE)
Computer Engineering Department, Faculty of Engineering, Ferdowsi University of Mashhad, Mashhad 9177948974, Iran
Corresponding author: Mahmoud Naghibzadeh (naghibzadeh@um.ac.ir)

ABSTRACT The remarkable growth of biological data is a motivation to accelerate the discovery of solutions in many domains of computational bioinformatics. In different phases of the computational pipelines, pattern matching is a very practical operation. For example, pattern matching enables users to find the locations of particular DNA subsequences in a database or DNA sequence. Furthermore, in these expanding biological databases, some patterns are updated over time. To perform faster searches, high-speed pattern matching algorithms are needed. The present paper introduces three pattern matching algorithms that are specially formulated to speed up searches on large DNA sequences. The proposed algorithms raise performance by utilizing word processing (in place of the character processing presented in previous works) and also by searching the least frequent word of the pattern in the sequence. In terms of time cost, the experimental results demonstrate the superiority of the presented algorithms over the other simulated algorithms.

INDEX TERMS Bioinformatics, string matching, DNA sequence, frequent pattern, exact algorithm.

I. INTRODUCTION

In the pattern matching problem, a text, sequence or database is scanned to detect the locations of a pattern in the text [1], [2]. It is imperative that this kind of problem be addressed mainly because of its applications in diverse and important areas, such as image and signal processing, information retrieval, text processing, search engines, question-answer systems, and chemistry [3]–[6].

Notably, the pattern matching problem arises in the different scopes of computational bioinformatics, which include the basic local alignment search, biomarker discovery, sequence alignment, proteogenomic mapping, and homologous series detection. In these disciplines, there is a need to recognize the locations of multiple patterns, including those of amino acids and nucleotides in databases [7], [8]. In biotechnology, forensics, medicine, and agriculture research, the knowledge of gene analysis and DNA sequences may be applied when exploring possible disease or abnormality diagnoses [4]. The comparison of a particular gene with similar genes of the same or different organisms and the prediction of its function can also employ DNA sequence analysis. In another application, the functionality of a recently discovered DNA sequence can be prespecified by investigating its similarity to known sequences of DNA. This approach has been used in various research studies and medical applications.

Although there are some generalized and specialized DNA pattern matching algorithms in the literature [8]–[17], the development of efficient algorithms is still required. This is mainly necessary because many current algorithms [18] may not be well scalable for databases or large DNA sequences due to high-computational costs. In contrast to approximate pattern matching [19], [20], the current paper focuses on the exact pattern matching problem which finds all the occurrences of a pattern in a text. The present study introduces three algorithms to mitigate the drawbacks of previous works. Similar to the literature [21]–[26], the operation of the proposed algorithms is divided into a preprocessing and a matching phase. In the preprocessing phase, the potential intervals of the text to be matched with the pattern are recognized. These candidate intervals are called windows. Next, during the matching phase, the windows are carefully scanned in order to be matched with the pattern. The fewer windows found in the preprocessing phase, the less time taken for verifying the windows in the matching phase. Accordingly,

The associate editor coordinating the review of this manuscript and approving it for publication was Liangxiu Han.

the present work's primary aim is to decrease the number of recognized windows. The main contributions of the current study are:

- For diagnosing the windows, some algorithms in the literature, such as in [22], search the text to separately discover the first and last character of the pattern. In contrast, the current study's first proposed algorithm finds the windows by simultaneously considering the first and last character of the pattern.
- Nowadays, the computational length of almost all processors is 32 or 64 bits in each execution cycle. In more precise terms, they can process 4 or 8 bytes of data in an instant. Therefore, 4 or 8 characters, indicated by a word, may be simultaneously compared to 4 or 8 other characters. As opposed to the character-based comparisons applied in many previous works, such as in [21]–[24], the present paper introduces a second algorithm to conduct word-based comparisons. The word processing is performed by utilizing the processing power of the processor. This approach creates a new class of string-matching algorithms that improve the performance of character-based algorithms. By employing this method, the current work decreases the number of detected windows and speeds up the comparisons. As a result, the performance improves in terms of time cost.
- The present study introduces a third algorithm that focuses on the word of the pattern having the fewest repetitions in the text. In other words, the algorithm searches the text for a low-frequency word of the pattern. This technique further advances the algorithm's efficiency by decreasing the number of discovered windows.

As for the rest of the current paper, Section II reviews related works, while Section III states the problem. Sections IV, V, and VI describe the first, second, and third proposed algorithms, respectively, along with their preprocessing and matching phases. In terms of time requirements, Section VII evaluates the performance of these algorithms in comparison with other previous algorithms. Finally, Section VIII concludes the paper.

II. RELATED WORK

This section presents related work on pattern matching. In the pattern matching literature, Brute Force (BF) [27] is a primary method which preprocesses neither the text nor the pattern. BF carries out a character by character comparison from left to right. After either a match or mismatch, the sliding window is shifted one position to the right and the matching is restarted from the first character of the pattern. The high consumption of time is a significant disadvantage of BF.

There are methods based on Deterministic Finite Automata (DFA) [25], [26] that combine the dynamic programming approach and DFA. Due to the use of a finite automaton, these methods are not often scalable for large sequences. In addition, the memory requirements are tremendously greater because of the usage of dynamic programming.

Knuth et al. [21] presented the KMP algorithm which performs the comparison from the left side. In the event of a mismatch, KMP moves the sliding window to the right by holding the longest overlap of a suffix of the matched text and a prefix of the pattern. This algorithm has a linear performance. Although it performs well when the alphabet size is large, the KMP algorithm requires a long run time when either the alphabet size is small or the length of the pattern is short [4].

The Boyer-Moore algorithm [23] and its variants [24], [28] search the pattern in the text from right to left. In other words, this algorithm first matches the pattern's last character. At the end of the matching phase, it computes the shift increment. To decrease the number of comparisons when a mismatch occurs, two useful rules (bad character and good suffix) are utilized. The disadvantage of the Boyer-Moore algorithm is the dependency of its preprocessing time on the pattern length and alphabet size.

The Divide and Conquer Pattern Matching (DCPM) [22] is a comparison-based algorithm. At the beginning of the DCPM's preprocessing phase, the text is scanned for the rightmost character of the pattern. The index of the findings is stored in the rightmost character table. Then, to detect the leftmost character of the pattern, the text is scanned again. In the case of sameness, the indexes are saved in the leftmost character table. By utilizing these two tables, DCPM identifies the boundary of the windows. In other words, by considering the length of the pattern, the elements of the tables are investigated. A window is found when the distance between the windows' leftmost and rightmost character (extracted earlier from the two tables) is the same as the length of the pattern. Therefore, DCPM requires two passes of the text and some computations to determine the windows. In the matching phase, the algorithm checks the other characters of the windows. If all characters of the pattern and the windows of the text are matched, then complete sameness occurs. The current study's first algorithm promotes DCPM by recognizing the windows with one pass of the text.

III. PROBLEM STATEMENT

In recent years, the size of biological data has significantly grown and yet these large volumes of data must be analyzed within a reasonable time. This issue is encountered in molecular biology because sequences of amino acids or nucleotides are often applied to approximate biological molecules. Another example is the basic information of species maintained by DNA sequences and the challenge of accessing this information via pattern matching. Besides, in a DNA sequence, specifying possible abnormalities or errors often calls for DNA sequence analysis. In addition, pattern matching is appropriate in fields such as phylogenetic and evolutionary biology. In these applications, specific DNA subsequences are extracted from the genomic data of organisms' different species for the purposes of understanding their relatedness, descent, and origin. Therefore, in this context, a pattern matching algorithm must be able to search

in databases spanning gigabytes to terabytes or more and in whole genomes with 3 billion base pairs [13]. On the other hand, the DNA sequences are very long. Therefore, the time consumed for matching with the pattern is considered as the most critical metric.

Consider pattern p of length m and text t of length n over alphabet $\sum$. A sequence of zero or more symbols of the alphabet is nominated as a string. Over alphabet $\sum$, the set of all possible strings is represented by $\sum^*$. If $x = uwv$, where u, v, and $w \in \sum^*$, then w is a substring of string x. Pattern p is stored in finite array $p\,[0..m-1]$, in which $m > 0$. The $(i+1)-$st character of p is represented by $p[i]$, in which $0 \leq i < m$. Besides, $p[i..j]$ signifies a substring of p from the $(i+1)-$st character to the $(j+1)-$st character of p, in which $0 \leq i \leq j < m$. A substring of the form $p[j..m-1]$ and $p[0..i]$, respectively, are called the suffix and prefix of p, in which $0 \leq i, j \leq m-1$.

Whenever character $t[s]$ of the text is aligned with character $p[0]$, substring $t[s..s+m-1]$ is called by the current window of the text. In the preprocessing phase of the proposed pattern matching algorithms, the text is scanned in order to find the windows of a size m. Then, to check the total occurrence of the pattern, the algorithms compare the characters of the pattern one by one against those of the window in the matching phase. After a whole match or mismatch, the other windows are examined for matching the text.

IV. FIRST-LAST PATTERN MATCHING ALGORITHM

This section proposes a simple First-Last Pattern Matching (FLPM) algorithm. FLPM is an enhancement of DCPM [22] and is composed of preprocessing and matching phases. The following explains these phases by using their pseudocodes. An example is then presented to complete the algorithm description.

A. PREPROCESSING PHASE

Since FLPM acts based on comparisons, the FLPM preprocessing phase scans text t to distinguish the windows of the text which are later utilized by the matching phase. At the beginning of the preprocessing phase, the first character of pattern p, i.e., $p[0]$, is searched in text t. As the length of p is m, the search is performed during interval $t[0..n-m]$. In contrast to the DCPM algorithm [22], whenever character $p[0]$ is aligned with the character in position s (in which $0 \leq s \leq n-m$) of text t, i.e., $t[s] = p[0]$, the FLPM algorithm immediately checks the correspondence of $p[m-1]$ to $t[s+m-1]$ in order to determine the end boundary of the window. If these two characters are also the same, the window of $t[s..s+m-1]$ is selected as a candidate interval to be more precisely checked in the next phase. The preprocessing phase then continues on to diagnose more windows. In recognizing windows, although this algorithm focuses on the first and last character of the pattern, it may be extended by considering the characters in other positions, such as $m/4$, $m/2$, and $3m/4$.

Algorithm 1 presents the pseudocode of this phase. In this algorithm, the initial amount of the *while* loop counter, indicated by *count* variable, is zero (Line 1). The *while* loop of this algorithm starts by $count = 0$ and continues up to the time when this counter reaches $n-m$ (Line 3). During the loop (Lines 3-11), the first and last character of the pattern, i.e., $p[0]$ and $p[m-1]$, are compared to $t[count]$ and $t[count+m-1]$, respectively. If the result of both comparisons is correct, the start index of this window, i.e., *count*, is stored in the *window_index* array and the number of windows, saved in the *num_window* variable, increases by one. Therefore, the number of windows and their start indexes are achieved in the preprocessing phase.

Algorithm 1 Preprocessing Phase of FLPM Algorithm

Input: Text t and pattern p stored in the arrays of $t[0..n-1]$ and $p[0..m-1]$, respectively, over finite alphabet $\sum$.

Output: The number of windows identified in this phase and their start indexes.

```
1.  count ← 0
2.  num_window ← 0
3.  WHILE count ≤ n − m DO
4.      IF t[count] = p[0] THEN
5.          IF t[count + m − 1] = p[m − 1] THEN
6.              window_index[num_window] ← count
7.              num_window ← num_window + 1
8.          END-IF
9.      END-IF
10.     count ← count + 1
11. END-WHILE
```

B. MATCHING PHASE

After the preprocessing phase identifies the windows, the matching phase investigates the windows to find all occurrences of the pattern in the text. Therefore, for the start index of each window, say s_i, found in the previous phase, the characters $p[1..m-2]$ must be compared with characters $t[s_i+1..s_i+m-2]$. The first and last character of the pattern and the window, respectively, should not be compared again because their sameness was already checked in the previous phase, i.e., $p[0] = t[s_i]$ and $p[m-1] = t[s_i+m-1]$. If a whole sameness of the pattern with the window of text occurs, then this window is an answer. Otherwise, there is a mismatch. After a match or mismatch, this phase continues by investigating the next windows.

Algorithm 2 provides the pseudocode of this phase and shall be described in detail here. The number of windows identified in the previous phase and their start indexes are considered as the inputs of Algorithm 2. In Lines 3-17, the algorithm's outer *while* loop repeats the instructions for all windows, i.e., $count = 0, 1, \ldots,$ *num_window* $- 1$. Line 4 assigns the start index of the current window to s. The inner *while* loop (Lines 6-11) checks the correspondence of the pattern's non-boundary characters and those of the text window with the start index of s. If alignment succeeds,

Algorithm 2 Matching Phase of FLPM Algorithm

Input: The number of windows identified in the preprocessing phase and their start indexes.
Output: The start index for all occurrences of pattern p in text t.

1. $count \leftarrow 0$
2. $num\_match \leftarrow$
3. **WHILE** $count < num\_window$ **DO**
4. $s \leftarrow window\_index[count]$
5. $c \leftarrow 1$
6. **WHILE** $c \leq m-2$ **DO**
7. **IF** $p[c] \neq t[s+c]$ **THEN**
8. BREAK /∗Exit the current loop∗/
9. **END-IF**
10. $c \leftarrow c+1$
11. **END-WHILE**
12. **IF** $c = m-1$ **THEN**
13. $match\_index[num\_match] \leftarrow s$
14. $num\_match \leftarrow num\_match + 1$
15. **END-IF**
16. $count \leftarrow count + 1$
17. **END-WHILE**

then s is stored in the *match_index* array (Lines 12-15) so as to identify the start index of an occurrence of pattern p in text t. The algorithm then continues to study the next window.

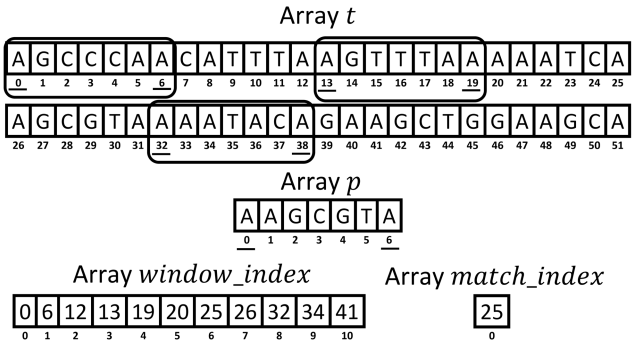


FIGURE 1. Example for the operation of the FLPM algorithm. Note that some windows have been specified in Array *t*.

C. AN EXAMPLE

As an example, Fig. 1 shows text t in array $t[0..51]$ and pattern p, which is *AAGCGTA* in array $p[0..6]$. The preprocessing phase searches for the first and last character of the pattern, i.e., $p[0]$ and $p[6]$, in the text. At the beginning of Algorithm 1, $t[0]$ and $t[6]$ are aligned to $p[0]$ and $p[6]$, respectively. Thus, the start index of the first window, i.e., 0, is stored in the *window_index* array. Following this example, the preprocessing phase identifies ten other windows. After this phase terminates, the matching phase employs Algorithm 2 to check all windows in order to find pattern p in text t. Consequently, the window of $t[25..31]$ and the pattern are the same.

Algorithm 3 PAPM Algorithm

Input: Text t and pattern p stored in the arrays of $t[0..n-1]$ and $p[0..m-1]$, respectively, over finite alphabet $\sum$.
Output: The start index for all occurrences of pattern p in text t.

/* **PREPROCESSING PHASE** */

1. $count \leftarrow 0$
2. $num\_window \leftarrow 0$
3. **WHILE** $count \leq n-m$ **DO**
4. **IF** $t[count..count + word\_len - 1]$
5. $= p[0..word\_len - 1]$ **THEN**
6. $window\_index[num\_window] \leftarrow count$
7. $num\_window \leftarrow num\_window + 1$
8. **END-IF**
9. $count \leftarrow count + 1$
10. **END-WHILE**

/* **MATHING CASE** */

11. $k \leftarrow m \bmod word\_len$
12. **IF** $k = 0$ **THEN**
13. $start\_index \leftarrow word\_len$
14. **ELSE**
15. $start\_index \leftarrow k$
16. **END-IF**
17. $count \leftarrow 0$
18. $num\_match \leftarrow 0$
19. **WHILE** $count < num\_window$ **DO**
20. $s \leftarrow window\_index[count]$
21. $c \leftarrow start\_index$
22. **WHILE** $c \leq m - word\_len$ **DO**
23. **IF** $p[c..c + word\_len - 1]$
24. $\neq t[s + c..s + c + word\_len - 1]$ **THEN**
25. BREAK
26. **END-IF**
27. $c \leftarrow c + word\_len$
28. **END-WHILE**
29. **IF** $c = m$ **THEN**
30. $match\_index[num\_match] \leftarrow s$
31. $num\_match \leftarrow num\_match + 1$
32. **END-IF**
33. $count \leftarrow count + 1$
34. **END-WHILE**

V. PROCESSOR-AWARE PATTERN MATCHING ALGORITHM

This section presents the Processor-Aware Pattern Matching (PAPM) algorithm. This algorithm is different from the FLPM algorithm in the way pattern p characters and text t characters are compared. PAPM performs comparisons based on a word comprised of several characters while FLPM uses a character-based pattern matching algorithm.

With the processing power of a processor, PAPM compares two words concurrently. In the case of a $b-$bit processor, the registers also have a $b-$bit length and, in each

execution cycle, the processor can compare the data of two registers. Since each byte (or character) is composed of eight bits, the number of processable bytes (or word length) for this processor, indicated by *word_len*, is computed as $word\_len = b/8$. In other words, by using its registers each time, the processor can compare a word (including *word_len* characters) with another. For instance, a 32-bit processor is able to simultaneously compare a word of four characters to another.

To apply this processor's ability in the pattern matching problem during the first phase of PAPM, the first word of the pattern is searched in the text. As shown in Lines 1-10 of Algorithm 3, the preprocessing phase of PAPM scans interval $t[0..n-m]$ to find the word $p[0..word\_len-1]$. The start index of the found windows is saved in the *window_index* array. As opposed to FLPM, when windows are sought in PAPM, the higher number of characters (a word) searched in the text yields a lower number of windows, which may decrease the time duration necessary for the next phase.

In the matching phase of Algorithm 3, displayed in Lines 11-34, the words of pattern p are compared with the words of the windows found, respectively. At the beginning of this phase (Lines 11-16), the start index for the word comparison is computed. By setting this start index, the algorithm is performed correctly, even if the length of the pattern and windows are not an integer multiple of the word length. Lines 19-34 of Algorithm 3 are similar to Lines 3-17 of Algorithm 2. However, Algorithm 3 carries out word-based processing in contrast to the character-based processing in Algorithm 2.

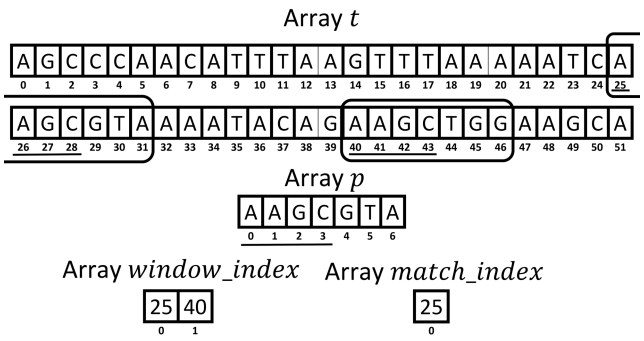


FIGURE 2. An example of the PAPM algorithm operation.

Fig. 2 provides an example employing Algorithm 3 run on a 32-bit processor. In the preprocessing phase, the first word (consisting of the first four characters) of pattern p is searched in text t. After the preprocessing phase of the PAPM algorithm, the *window_index* array is composed of two start indexes of the found windows, i.e., 25 and 4. For this example, it should be noted that the FLPM algorithm identifies 11 start indexes as candidate intervals or windows. Therefore, PAPM reduces the number of identified windows. In the matching phase, since the remainder of pattern length over the word length is 3, the start index for matching is also 3 (Lines 11-16). Therefore, the second word of the pattern to align with the second word of windows is CGTA. After the termination of this phase, only one of the two windows (i.e., $t[25..31]$) is matched with the pattern.

VI. LEAST FREQUENCY PATTERN MATCHING ALGORITHM

The Least Frequency Pattern Matching (LFPM) algorithm is an enhancement of PAPM that is specialized for DNA applications. However, the LFPM algorithm can be extended to other pattern matching applications. LFPM is an appropriate choice when many patterns must be searched in the text, but not an efficient option for few patterns due to the time overhead. To reduce the number of recognized windows, LFPM searches for a low-frequency word of the pattern in the text. In other words, LFPM focuses on a word that is expected to appear less than other words of the pattern appearing in the text.

TABLE 1. THE freq_table.

	0 1 2 3 4 5 . . .	$word\_len-2$	$word\_len-1$	$word\_len$
0	A A A A A A . . .	A	A	n_0
1	A A A A A A . . .	A	C	n_1
2	A A A A A A . . .	A	G	n_2
3	A A A A A A . . .	A	T	n_3
.	. . .			
.	. . .			
.	. . .			
$4^{word\_len}-2$	T T T T T T . . .	T	G	$n_{4^{word\_len}-2}$
$4^{word\_len}-1$	T T T T T T . . .	T	T	$n_{4^{word\_len}-1}$

In the first step before the preprocessing phase, LFPM computes the frequency of all possible words for the related alphabet. Although the computations of this step create a time overhead, it dramatically reduces the time cost of the upcoming phases. To compute the frequency of different words, the current text or a related dataset can be selected as a reference. Here, the Human Reference Genome (HRG) [29] is applied as a reference to count all possible words (i.e., Σ^*) having the length of *word_len* over the finite alphabet $\Sigma = \{A, C, G, T\}$. Note that four nucleotide bases in a molecule of DNA (or a DNA sequence) are Adenine, Guanine, Cytosine, and Thymine. Because *word_len* is the length of a word in a particular computer, the number of all possible words over alphabet Σ is 4^{word_len}. It is worth mentioning that this number is fixed for all $b-$bit computers. For example, in a 32-bit computer, 256 different words having the length of four characters from the related alphabet Σ can be generated. The frequency of all words is stored in a table nominated by *freq_table* and having 4^{word_len} rows and $word\_len + 1$ columns, as indicated in Table 1. In each row, there is a word the length of *word_len* characters in the columns zero to $word\_len-1$. Besides, the frequency of each word is shown in the *word_len* column of that word. Since *word_len* is either four or eight on a 32 or 64-bit computer, respectively, the amount of memory required to maintain the frequency table is reasonable. Therefore, this table can be placed in the main memory.

Algorithm 4 Filling the Table of Frequency

Input: The array of reference and the length of this array.
Output: The filling table of word frequency.

1. $freq\_table[0..4^{word\_len}-1][0..word\_len-1] \leftarrow$
2. *All possible words with word_len length over* Σ={*ACGT*}
3. $freq\_table[0..4^{word\_len}-1][word\_len] \leftarrow 0$
4. $count \leftarrow 0$
5. **WHILE** $count \leq REF\_len - word\_len$ **DO**
6. $row\_count \leftarrow 0, col\_count \leftarrow 0$
7. $w\,[0..word\_len-1] \leftarrow REF\,[count..count+word\_len-1]$
8. **WHILE** $col\_count < word\_len$ **DO**
9. **SWITCH** $w\,[col\_count]$
10. **CASE** 'A': nothing
11. **CASE** 'C': Increase row_count by $1 \times 4^{word\_len-col\_count-1}$
12. **CASE** 'G': Increase row_count by $2 \times 4^{word\_len-col\_count-1}$
13. **CASE** 'T': Increase row_count by $3 \times 4^{word\_len-col\_count-1}$
14. **END- SWITCH**
15. $col\_count \leftarrow col\_count + 1$
16. **END-WHILE**
17. Increase $freq\_table[row\_count][word\_len]$ by one
18. $count \leftarrow count + 1$
19. **END-WHILE**

Algorithm 5 LFPM Algorithm

Input: The filled *freq_table* on reference data. Text t and pattern p stored in the arrays of $t[0..n-1]$ and $p[0..m-1]$, respectively, over finite alphabet $\sum = \{A, C, G, T\}$.
Output: The first indexes for all occurrences of pattern p in text t.

/∗PREPROCESSING PHASE∗/

/∗Step 1: finding the least frequent word in the pattern∗/

1. $count \leftarrow 0$
2. $min\_value \leftarrow \infty$
3. $min\_index \leftarrow -1$
4. **WHILE** $count \leq m - word\_len$ **DO**
5. $row\_count \leftarrow 0, col\_count \leftarrow 0$
6. $w\,[0..word\_len-1] \leftarrow p\,[count..count+word\_len-1]$
7. **WHILE** $col\_count < word\_len$ **DO**
8. **SWITCH** $w\,[col\_count]$
9. **CASE**'A': nothing
10. **CASE**'C': Increase row_count by $1 \times 4^{word\_len-col\_count-1}$
11. **CASE**'G': Increase row_count by $2 \times 4^{word\_len-col\_count-1}$
12. **CASE**'T': Increase row_count by $3 \times 4^{word\_len-col\_count-1}$
13. **END- SWITCH**
14. $col\_count \leftarrow col\_count + 1$
15. **END-WHILE**
16. **IF** $freq\_table[row\_count][word\_len] < min\_value$ **THEN**
17. $min\_value = freq\_table[row\_count][word\_len]$
18. $min\_index \leftarrow count$
19. **END-IF**
20. $count \leftarrow count + 1$
21. **END-WHILE**

/∗Step 2: finding the windows∗/

22. $count \leftarrow min\_index$
23. $num\_window \leftarrow 0$
24. **WHILE** $count \leq n - (m - min\_index)$ **DO**
25. **IF** $t\,[count..count+word\_len-1]$
26. $= p[min\_index..min\_index+word\_len-1]$ **THEN**
27. $window\_index[num\_window] \leftarrow count - min\_index$
28. $num\_window \leftarrow num\_window + 1$
29. **END-IF**
30. $count \leftarrow count + 1$
31. **END-WHILE**

/∗MATCHING PHASE∗/

32. $count \leftarrow 0$
33. $num\_match \leftarrow 0$
34. $k \leftarrow mmodword\_len$
35. **WHILE** $count < num\_window$ **DO**
36. $s \leftarrow window\_index[count]$
37. $c \leftarrow 0$
38. $w \leftarrow word\_len$
39. **WHILE** $c \leq m - 1$ **DO**
40. **IF** $c > m - word\_len$ **THEN**
41. $w \leftarrow k$
42. **END-IF**
43. **IF** $p\,[c..c+w-1]$
44. $\neq t[s+c..s+c+w-1]$ **THEN**
45. BREAK
46. **END-IF**
47. $c \leftarrow c + w$
48. **END-WHILE**
49. **IF** $c = m$ **THEN**
50. $match\_index[num\_match] \leftarrow s$
51. $num\_match \leftarrow num\_match + 1$
52. **END-IF**
53. $count \leftarrow count + 1$
54. **END-WHILE**

For details, Algorithm 4 explains the filling task of *freq_table*. In Lines 1 and 2, all possible words with *word_len* characters over alphabet Σ are assigned to columns $0..word\_len-1$ of all rows in the table. Line 3 initializes the cells of the *word_len* column by zero. In Lines 5-19, the outer loop repeats the instructions for all words in the reference. On the other hand, the inner *while* loop (Lines 8-16) matches the current reference word to one of the *freq_table* words.

When matching occurs, the frequency of that word, saved in the *word_len* column, increases by one (Line 17). It should be mentioned that a total search of the table should be avoided as this is a time-consuming task. Instead, a heuristic method, implemented by Lines 8-16, can be applied. In each iteration of the inner *while* loop, a character of the word is compared to the elements of Σ. According to the result of this comparison, *row_count* increases. This action continues until the time when the location of the current word in the table is achieved. As the length of the reference array is presented by *REF_len* and the word-based comparison is performed on alphabet Σ, the time complexity for filling *freq_table* is $\theta(word\_len \times (REF\_len - word\_len + 1))$. Although filling *freq_table* is a time-consuming task, this is performed only one time on the reference data. After completion, the table can be utilized for searching every pattern over the related alphabet Σ. It should be noted that the table can be constructed for any alphabet based on other data sets or text for other applications. However, in applications with a large alphabet, the size of the table may be enlarged. Note that, if the reference modifies over time, the table can be periodically updated.

Whenever *freq_table* is ready, the preprocessing phase can start. At the beginning of this phase (Lines 1-21 of Algorithm 5), the least frequent word in the pattern must be specified by utilizing the filled *freq_table*. Therefore, in each iteration, the current word frequency of the pattern is obtained from the table. Then, if this frequency is currently the lowest, the start index of this word and its value are saved for comparisons with those of other words in the pattern (Lines 16-19). The time complexity for determining the least frequent word is $\theta(word\_len \times (m - word\_len))$. However, as *word_len* is fixed for each computer, this time complexity may be stated as $\theta(m)$. In the second step of the preprocessing phase, similar to PAPM, the text is scanned to discover windows by finding the least frequent word (See Lines 22-31). In the following, the matching phase (Lines 32-54) of Algorithm 5 is proposed, which is similar to that of Algorithm 3.

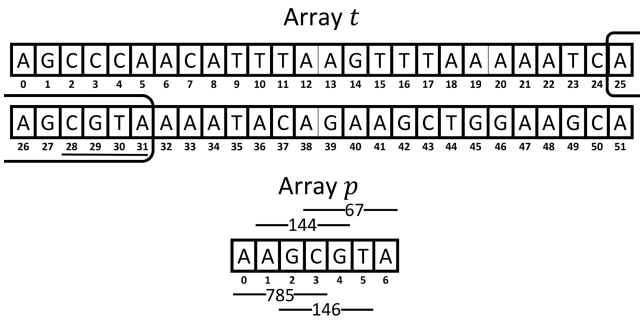


FIGURE 3. An example for the operation of LFPM algorithm.

For example, Fig. 3 illustrates the operation of LFPM on a 32-bit computer. It is assumed that *freq_table* is ready from beforehand. Therefore, the frequency of all words in the pattern is extractable from the table as AAGC: 785, AGCG: 144, GCGT: 146, and CGTA: 67. As a result, the word CGTA is chosen as a word having the least frequency. Then, the search is started to encounter CGTA in the text. The start index of the related window is only 25. As the least frequent word is searched in the text, the number of windows using LFPM is often less than in the previously presented algorithms. After distinguishing the windows, the matching phase checks the sameness of the pattern words with those of the windows. In this case, the words AAGC and GTA, respectively, are compared with the first and second words of the window. This comparison reveals that the recognized window in the index of 25 is the answer. Consequently, the superiority of LFPM is its lowest frequency word approach which tremendously reduces the number of windows that must be checked in the next phase.

VII. RESULTS

This section compares the performance of the presented algorithms (FLPM, PAPM, and LFPM) with that of the Brute Force (BF), Boyer-Moore (BM), and Divide and Conquer Pattern Matching (DCPM) algorithms. The specifications of the computing environment for executing different simulated algorithms were as follows:

- Intell®core™2 Duo CPU T6600 (a 2.2 GHz clock)
- A 2GB Memory
- Acer (Aspire 5738)
- Windows 7 ultimate 32 bit

Due to the use of a 32-bit computer, the word length for the PAPM and LFPM algorithms was considered as four bytes. In LFPM, the HRG [29] was utilized as a reference to construct the table of frequency. The simulation was performed with the C programing language. In each experiment, ten patterns were searched in the reference and the average of the results was then reported. It should be noted that LFPM has a time overhead to construct the table of frequency. As the HRG dataset was employed for all experiments, this time overhead was fixed during the simulation. The amount of overhead was 12 milliseconds. However, this time overhead was ignored in the calculations because the table of frequency is created only once for any text or database worked on by LFPM. The results of the simulated algorithms' performance evaluation in the preprocessing phase, matching phase, and total phases in terms of time cost are presented as follows.

A. THE TIME OF PREPROCESSING PHASE

Fig. 4 illustrates the time of the preprocessing phase for different simulated algorithms over the pattern length. It should be pointed out that the BM's time is negligible as the pattern is only analyzed during the preprocessing phase of this algorithm. This figure reveals the dominance of the present study's algorithms over the other algorithms. In FLPM, one pass across the text is sufficient to concurrently discover the first and last character of the pattern. In contrast, DCPM requires two passes: one to search for the leftmost character of the pattern and another for the rightmost. PAPM searches for the first word of the pattern in the text, for example,

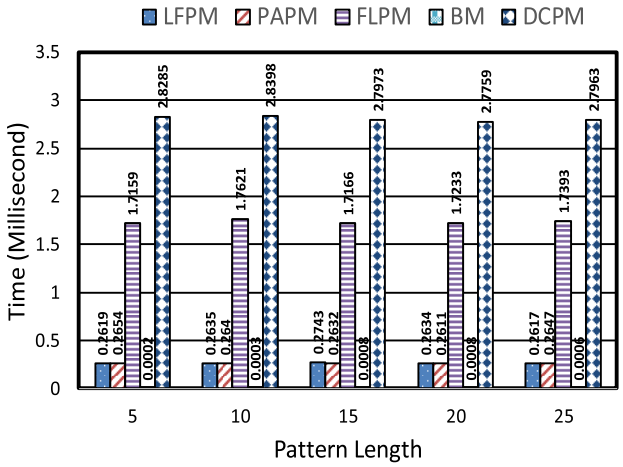


FIGURE 4. Time comparisons in preprocessing phase.

the first four characters in a 32-bit computer. Therefore, the PAPM preprocessing phase also needs one pass. As displayed in Fig. 4, it is expected that PAPM finds fewer windows than FLPM, because it considers a word with some characters in order to recognize the windows, while FLPM only focuses on two characters (the first and last). Because LFPM searches for the least frequent word of the pattern, it finds fewer windows. However, finding the least frequent word from the frequency table in LFPM is time-consuming. As a result, PAPM and LFPM consume the least amount of time among the simulated algorithms in Fig. 4. It is noteworthy that BF has been removed from this figure since it does not have any preprocessing phase.

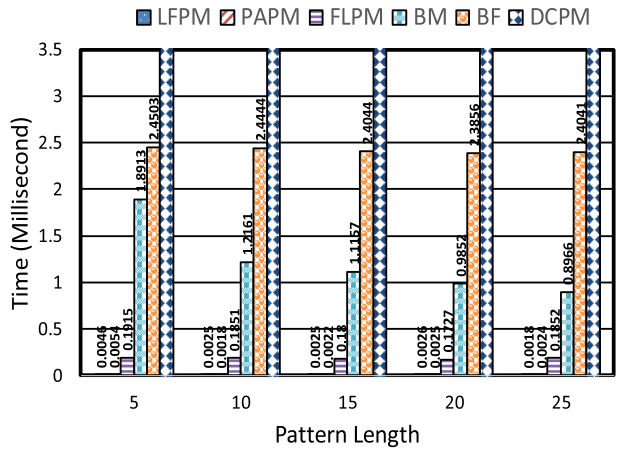


FIGURE 5. Time comparisons in matching phase.

B. THE TIME OF MATCHING PHASE

Fig. 5 presents the time cost of the matching phase for the algorithms. Because DCPM compares each element in the rightmost table to all elements of the leftmost table, its matching phase is very time-consuming. Thus, the DCPM plot has been cut to 3.5 milliseconds. As shown, FLPM, PAPM, and LFPM outperform the other algorithms. The main reason for their superiority is because they specify a low number of

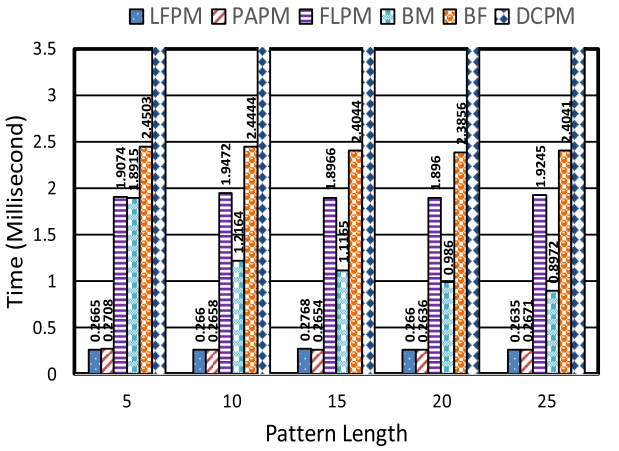


FIGURE 6. Total time comparisons.

windows in the previous phase. In addition, by employing word processing, PAPM and LFPM's inquiry to match the windows and the pattern is much faster than that of the simulated character-based algorithms, i.e., BF, DCPM, BM, and FLPM.

C. TOTAL TIME

Fig. 6 provides the sum of the time costs for the two phases, as illustrated in the two previous figures. This figure reveals that the use of word processing by PAPM and LFPM significantly reduces the time required to execute pattern matching. In LFPM, there is a difference between the repetition number of the least frequent word and those of other words of the pattern. The higher value of this difference leads to more improvement in LFPM performance.

VIII. CONCLUSION

The current paper introduces three new algorithms: FLPM, PAPM, and LFPM. Similar to previous works, LFPM is a character-based pattern matching algorithm, while PAPM and LFPM perform based on the word processing approach. Furthermore, LFPM searches for the lowest frequency word of the pattern in order to minimize the algorithm run time. The present work's experimental results reveal that the proposed algorithms, especially LFPM, surpass the other simulated algorithms in terms of time cost. This improvement is mainly due to having decreased the number of found windows.

The presentation of a parallel version of the presented algorithms is left for future work. Moreover, as this paper provides solutions for exact pattern matching, future research may investigate the algorithms supporting approximate matching.

REFERENCES

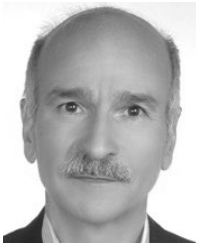
[1] P. Montanari, I. Bartolini, P. Ciaccia, M. Patella, S. Ceri, and M. Masseroli, "Pattern similarity search in genomic sequences," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 11, pp. 3053–3067, Nov. 2016.

[2] V. Abrishami, A. Zaldívar-Peraza, J. M. de la Rosa-Trevín, J. Vargas, J. Otón, R. Marabini, Y. Shkolnisky, J. M. Carazo, and C. O. S. Sorzano, "A pattern matching approach to the automatic selection of particles from low-contrast electron micrographs," *Bioinformatics*, vol. 29, no. 19, pp. 2460–2468, Oct. 2013.

[3] S. Faro and T. Lecroq, "The exact online string matching problem," *CSURACM Comput. Surv.*, vol. 45, no. 2, pp. 1–42, Feb. 2013.
[4] M. Tahir, M. Sardaraz, and A. A. Ikram, "EPMA: Efficient pattern matching algorithm for DNA sequences," *Expert Syst. Appl.*, vol. 80, pp. 162–170, Sep. 2017.
[5] S. I. Hakak, A. Kamsin, P. Shivakumara, G. A. Gilkar, W. Z. Khan, and M. Imran, "Exact string matching algorithms: Survey, issues, and future research directions," *IEEE Access*, vol. 7, pp. 69614–69637, 2019.
[6] M. Sazvar, M. Naghibzadeh, and N. Saadati, "Quick-MLCS: A new algorithm for the multiple longest common subsequence problem," in *Proc. 5th Int. Conf. Comput. Sci. Softw. Eng. (CSE)*, 2012, pp. 61–66.
[7] V. Y. Gudur and A. Acharyya, "Hardware-software codesign based accelerated and reconfigurable methodology for string matching in computational bioinformatics applications," *IEEE/ACM Trans. Comput. Biol. Bioinf.*, to be published.
[8] M. Amit, "Local exact pattern matching for non-fixed RNA structures," *IEEE/ACM Trans. Comput. Biol. Bioinf.*, vol. 11, no. 1, pp. 219–230, Jan. 2014.
[9] D. Cantone, S. Faro, and A. Pavone, "Linear and efficient string matching algorithms based on weak factor recognition," *J. Exp. Algorithmics*, vol. 24, no. 1, pp. 1–20, Feb. 2019.
[10] F. Deng, L. Wang, and X. Liu, "An efficient algorithm for the blocked pattern matching problem," *Bioinformatics*, vol. 31, no. 4, pp. 532–538, Feb. 2015.
[11] C. Ryu and K. Park, "Improved pattern-scan-order algorithms for string matching," *J. Discrete Algorithms*, vol. 49, pp. 27–36, Mar. 2018.
[12] Z. Li, M. Yan, and M. Zhou, "Synthesis of structurally simple supervisors enforcing generalized mutual exclusion constraints in Petri Nets," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 40, no. 3, pp. 330–340, May 2010.
[13] A. Srikantha, A. S. Bopardikar, K. K. Kaipa, P. Venkataraman, K. Lee, T. Ahn, and R. Narayanan, "A fast algorithm for exact sequence search in biological sequences using polyphase decomposition," *Bioinformatics*, vol. 26, no. 18, pp. i414–i419, Sep. 2010.
[14] S. Hakak, A. Kamsin, P. Shivakumara, M. Y. Idna Idris, and G. A. Gilkar, "A new split based searching for exact pattern matching for natural texts," *PLoS ONE*, vol. 13, no. 7, Jul. 2018, Art. no. e0200912.
[15] H. Kim and K.-I. Choi, "A pipelined non-deterministic finite automaton-based string matching scheme using merged state transitions in an FPGA," *PLoS ONE*, vol. 11, no. 10, Oct. 2016, Art. no. e0163535.
[16] C.-L. Lee, Y.-S. Lin, and Y.-C. Chen, "A hybrid CPU/GPU pattern-matching algorithm for deep packet inspection," *PLoS ONE*, vol. 10, no. 10, Oct. 2015, Art. no. e0139301.
[17] C. Otto, "ExpaRNA-P: Simultaneous exact pattern matching and folding of RNAs," *BMC Bioinf.*, vol. 15, no. 1, p. 404, Dec. 2014.
[18] A. M. Al-Ssulami and H. Mathkour, "Faster string matching based on hashing and bit-parallelism," *Inf. Process. Lett.*, vol. 123, pp. 51–55, Jul. 2017.
[19] A. Policriti and N. Prezza, "Fast randomized approximate string matching with succinct hash data structures," *BMC Bioinf.*, vol. 16, no. 9, p. S4, Dec. 2015.
[20] L. A. K. Ayad, S. P. Pissis, and A. Retha, "LibFLASM: A software library for fixed-length approximate string matching," *BMC Bioinf.*, vol. 17, no. 1, p. 454, Dec. 2016.
[21] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt, "Fast pattern matching in strings," *SIAM J. Comput.*, vol. 6, no. 2, pp. 323–350, Jul. 1977.
[22] S. V. Raju, K. K. V. V. S. Reddy, and C. S. Rao, "Parallel string matching with linear array, butterfly and divide and conquer models," *Ann. Data. Sci.*, vol. 5, no. 2, pp. 181–207, Jun. 2018.
[23] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Commun. ACM*, vol. 20, no. 10, pp. 762–772, Oct. 1977.
[24] A. Apostolico and R. Giancarlo, "The boyer–Moore–galil string searching strategies revisited," *SIAM J. Comput.*, vol. 15, no. 1, pp. 98–105, Feb. 1986.
[25] C. Charras and T. Lecroq, *Handbook of Exact String Matching Algorithms*. Princeton, NJ, USA: Citeseer, 2004.
[26] H. Li and R. Durbin, "Fast and accurate long-read alignment with Burrows-Wheeler transform," *Bioinformatics*, vol. 26, no. 5, pp. 589–595, Mar. 2010.
[27] T. H. Cormen C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (Computer Science). Cambridge, MA, USA: MIT Press, 2009. [Online]. Available: https://books.google.com/books?id=aefUBQAAQBAJ
[28] M. Crochemore, T. Lecroq, A. Czumaj, L. Gasieniec, S. Jarominek, W. Plandowski, and W. Rytter, "Speeding up two string-matching algorithms," *Algorithmica*, vol. 12, nos. 4–5, pp. 247–267, Nov. 1992.
[29] (2019). *National Center for Biotechnology Information*. [Online]. Available: https://www.ncbi.nlm.nih.gov/projects/genome/guide/human/index.shtml

PEYMAN NEAMATOLLAHI received the B.S. and M.S. degrees in computer engineering, with a concentration in parallel and distributed systems, in 2007 and 2011, respectively, and the Ph.D. degree in computer engineering from the Ferdowsi University of Mashhad, in 2017. He has published several conference and journal articles. His research interests are in bioinformatics, the Internet of Things (IoT), job scheduling in distributed environments, distributed algorithms, and fuzzy logic control. He also serves as a Reviewer of the IEEE Sensors Journal, *Ad Hoc Networks*, the *Journal of Networks and Computer Applications*, *Computers and Electrical Engineering*, *Wireless Networks*, and the *Journal of Supercomputing*.

MONTASSIR HADI received the M.S. degree in computer engineering from the Ferdowsi University of Mashhad, Iran. His thesis was on string matching algorithms for bioinformatics applications. His research interests are in bioinformatics and string-matching algorithms. Besides, he can work well with the simulation tools as a software developer.



MAHMOUD NAGHIBZADEH (Senior Member, IEEE) received the M.S. and Ph.D. degrees in computer science and computer engineering from the University of Southern California (USC), USA. He has taught Undergraduate and Graduate courses at USC and the University of South Florida, USA. He was a Visiting Professor with the University of California at Irvine (UCI), USA, in 1991, and a Visiting Professor with Monash University, Australia, from 2003 to 2004. He is currently a Full Professor with the Ferdowsi University of Mashhad, Iran. He has published numerous articles and eight books. His research interests include scheduling aspects of real-time systems, grid, cloud, multiprocessors, multicores, and GPGPUs and also bioinformatics algorithms, especially genomics and proteomics. He was a recipient of many awards including M.S. and Ph.D. study scholarship. He has been the chairman of two international conferences and technical chair of many others. He is the Reviewer of many journals and a member of many computer societies.