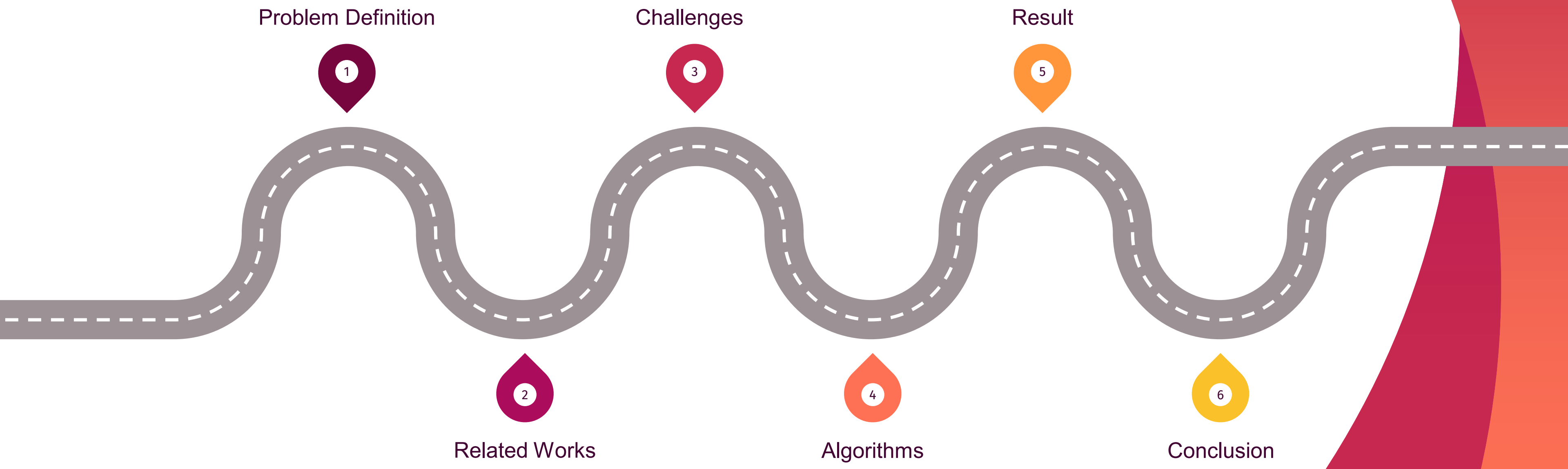
In The Name of God

Simple and Efficient Pattern Matching Algorithms for Biological Sequences

Presenters:
Nazanin Fereydoonizade – Sarvin Saravi

Suppervisor:
Dr. Shakibian

Road Map


Problem Definition
1
Related Works
2
Challenges
3
Algorithms
4
Result
5
Conclusion
6

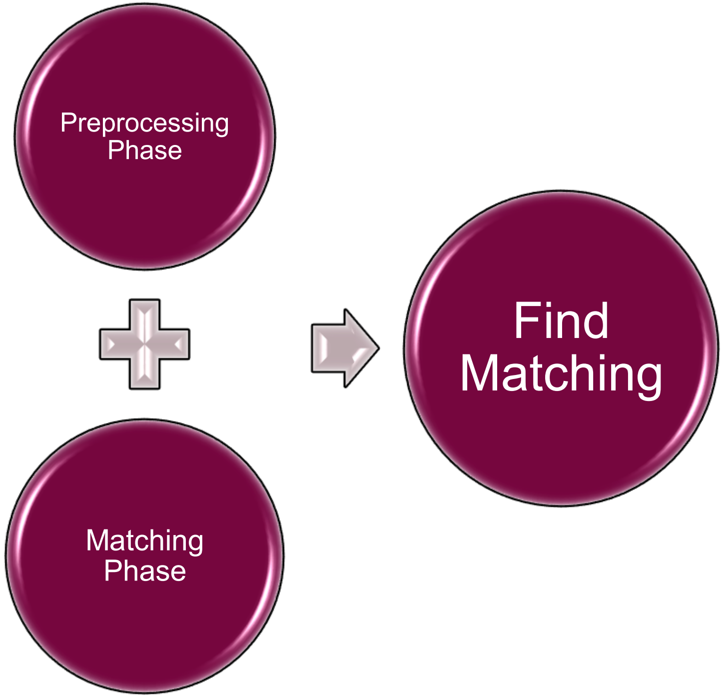
What is Pattern Matching?

Scan a sequence or a database to detect the locations of a pattern

Applications

1. Image & Signal Processing
2. Information Retrieval
3. Search Engine
4. Text Processing
5. Parsers
6. NLP

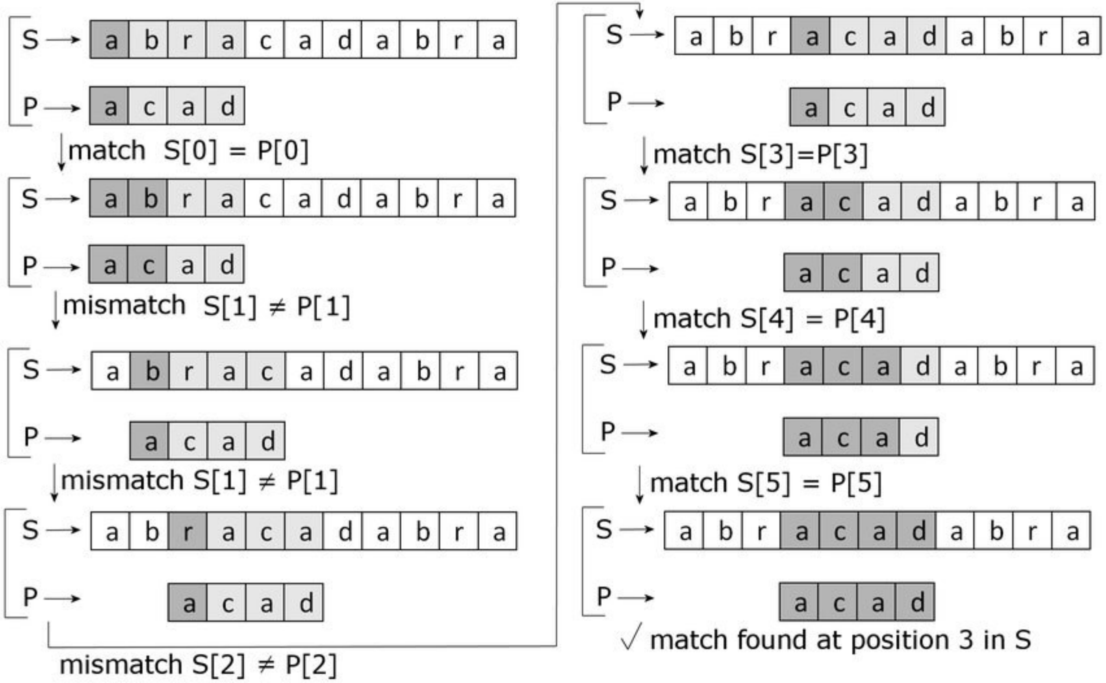
Pattern Matching Phases



Related Works

1. Brute Force — Time problem, O(mn)
2. DFA-bases — Not Scalable, O(n)
3. KMP — Not Suitable for Small Strings, O(n)
4. Boyer Moore — Depend on String size, O(m+n) / O(mn)
5. DCPM

1. Brute Force



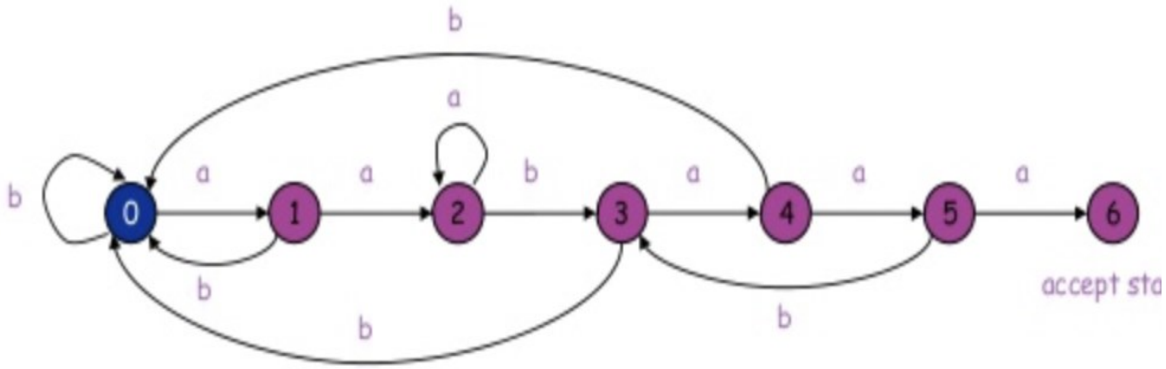
2. DFA-based

- Build DFA from Pattern
- Run DFA on Text

example

Text: aaabaabaaab

Pattern: aabaaa



Challenges

1. Library
2. Optimal File Size
3. Execution Time
4. Performance Analysis Using Different Encoding Techniques

Algorithm.1: First-Last Pattern Matching (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.

$count \leftarrow 0$

$num\_window \leftarrow 0$

WHILE $count \leq n - m$ **DO**

IF $t[count] = p[0]$ **THEN**

IF $t[count + m - 1] = p[m-1]$ **THEN**

$window\_index[num\_window] \leftarrow count$

$num\_window \leftarrow num\_window + 1$

END-IF

END-IF

$count \leftarrow count + 1$

END-WHILE

Algorithm.1: First-Last Pattern Matching (Matching Phase)

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.

```
count ← 0
num_match ←
WHILE count < num_window DO
    s ← window_index[count]
    c ← 1
    WHILE c ≤ m − 2 DO
        IF p[c] ≠ t[s + c] THEN
            BREAK /*Exit the current loop*/
        END-IF
        c ← c + 1
    END-WHILE
    IF c = m − 1 THEN
        match_index[num_match] ← s
        num_match ← num_match + 1
    END-IF
    count ← count + 1
END-WHILE
```

Algorithm.1: First-Last Pattern Matching
(Example)

Array t

A	G	C	C	C	A	A	C	A	T	T	T	A	A	G	T	T	T	A	A	A	A	A	T	C	A
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

A	G	C	G	T	A	A	A	A	T	A	C	A	G	A	A	G	C	T	G	G	A	A	G	C	A
26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51

Array p

A	A	G	C	G	T	A
0	1	2	3	4	5	6

Array $window\_index$

0	6	12	13	19	20	25	26	32	34	41
0	1	2	3	4	5	6	7	8	9	10

Array $match\_index$

25
0

Algorithm.2: Processor-Aware Pattern Matching

(Preprocessing Phase)

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** */

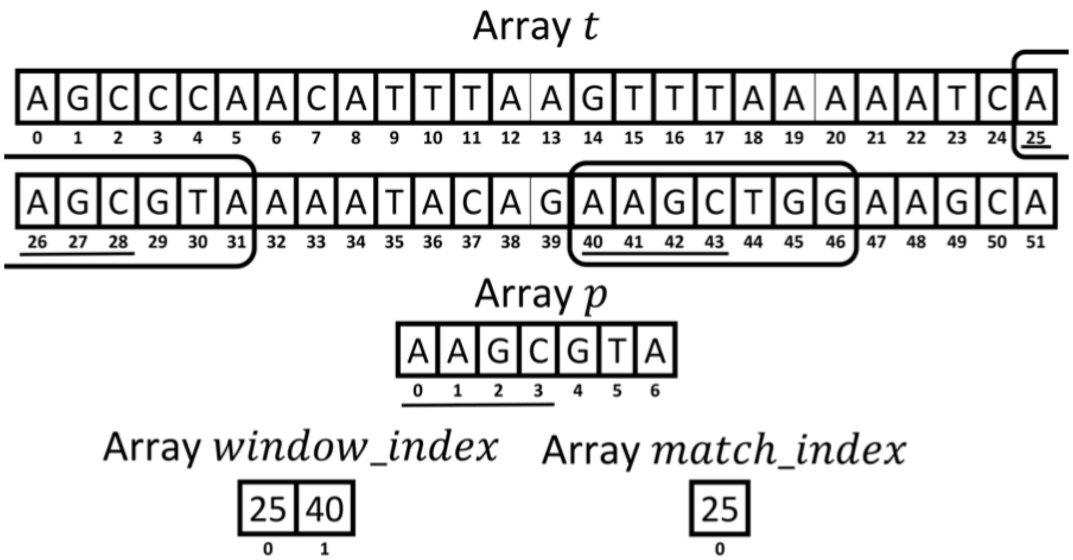
$count \leftarrow 0$
$num\_window \leftarrow 0$
WHILE $count \leq n - m$ **DO**
    IF $t\,[count..count + word\_len - 1]$
        $= p[0..word\_len - 1]$ **THEN**
          $window\_index[num\_window] \leftarrow count$
          $num\_window \leftarrow num\_window + 1$
    END-IF
    $count \leftarrow count + 1$
END-WHILE

Algorithm.2: Processor-Aware Pattern Matching
(Matching Phase)

```
k ← m mod word_len
IF  k = 0  THEN
        start_index ← word_len
ELSE
        start_index ← k
END-IF
count ← 0
num_match ← 0
WHILE  count < num_window   DO
        s ← window_index[count]
        c ← start_index
        WHILE  c ≤ m − word_len   DO
            IF  p[c..c + word_len − 1]
              ≠ t[s + c..s + c + word_len − 1]  THEN
                        BREAK
            END-IF
            c ← c + word_len
        END-WHILE
        IF  c = m  THEN
            match_index[num_match] ← s
            num_match ← num_match + 1
        END-IF
        count ← count + 1
END-WHILE
```

Algorithm.2: Processor-Aware Pattern Matching

(Example)



Algorithm.3: Least frequency pattern matching algorithm

- LFPM
- an enhancement of PAPM
- specialized for DNA
- many patterns vs. few patterns

Least frequency pattern matching algorithm

The first step before the preprocessing phase

- Σ = { A, C, G, T } HRG
- word_len = b/8 for b-bit computer
- all possible words = 4^{word_len}
- Create freq_table

Least frequency pattern matching algorithm

The first step before the preprocessing phase

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	$n_{4^{word\_len}-1}$

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* $\Sigma = \{ACGT\}$
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**

Least frequency pattern matching algorithm

The preprocessing & Matching phase

- words in the pattern ➡ least frequent word ➡ start index
- similar to PAPM

Least frequency pattern matching algorithm

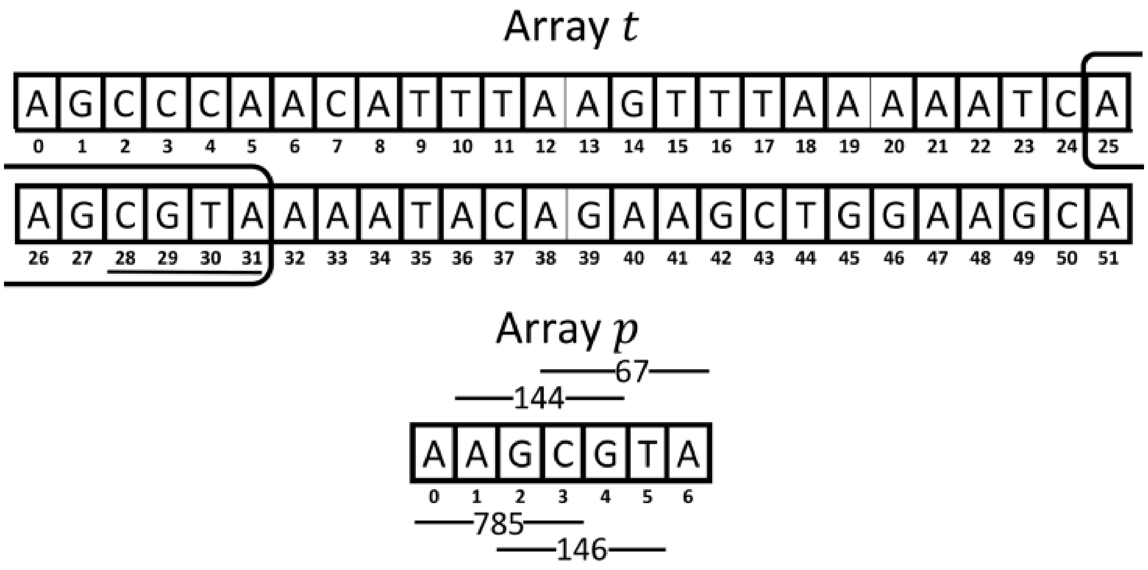


FIGURE 3. **An example for the operation of LFPM algorithm.**

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*/

$count \leftarrow 0$
$min\_value \leftarrow \infty$
$min\_index \leftarrow -1$
WHILE $count \leq m - word\_len$ **DO**
  $row\_count \leftarrow 0$, $col\_count \leftarrow 0$
  $w[0..word\_len-1] \leftarrow p[count..count + word\_len - 1]$
  WHILE $col\_count < word\_len$ **DO**
    SWITCH $w[col\_count]$
      CASE'A': nothing
      CASE'C': Increase row_count by $1 \times 4^{word\_len-col\_count-1}$
      CASE'G': Increase row_count by $2 \times 4^{word\_len-col\_count-1}$
      CASE'T': Increase row_count by $3 \times 4^{word\_len-col\_count-1}$
    END- SWITCH
    $col\_count \leftarrow col\_count + 1$
  END-WHILE
  IF $freq\_table[row\_count][word\_len] < min\_value$ **THEN**
    $min\_value = freq\_table[row\_count][word\_len]$
    $min\_index \leftarrow count$
  END-IF
  $count \leftarrow count + 1$
END-WHILE

Time complexity :
$\theta(word\_len \times (m - word\_len)) \approx \theta(m)$

/*preprocessing phase*/

/*Step 2: finding the windows*/

```
count ← min_index
num_window ← 0
WHILE count ≤ n − (m − min_index) DO
        IF t[count..count + word_len − 1]
            = p[min_index..min_index + word_len − 1]
            THEN
              window_index[num_window] ←
              count − min_index
              num_window ← num_window + 1
        END-IF
count ← count + 1
END-WHILE
```

/*MATCHING PHASE*/

```
count ← 0
num_match ← 0
k ← mmodword_len
WHILE count < num_window DO
        s ← window_index[count]
        c ← 0
        w ← word_len
        WHILE c ≤ m − 1 DO
            IF c > m − word_len THEN
                    w ← k
            END-IF
            IF p [c..c + w − 1]
               ≠ t[s + c..s + c + w − 1] THEN
                  BREAK
            END-IF
            c ← c + w
        END-WHILE
        IF c = m THEN
             match_index[num_match] ← s
             num_match ← num_match + 1
        END-IF
        count ← count + 1
END-WHILE
```

Results

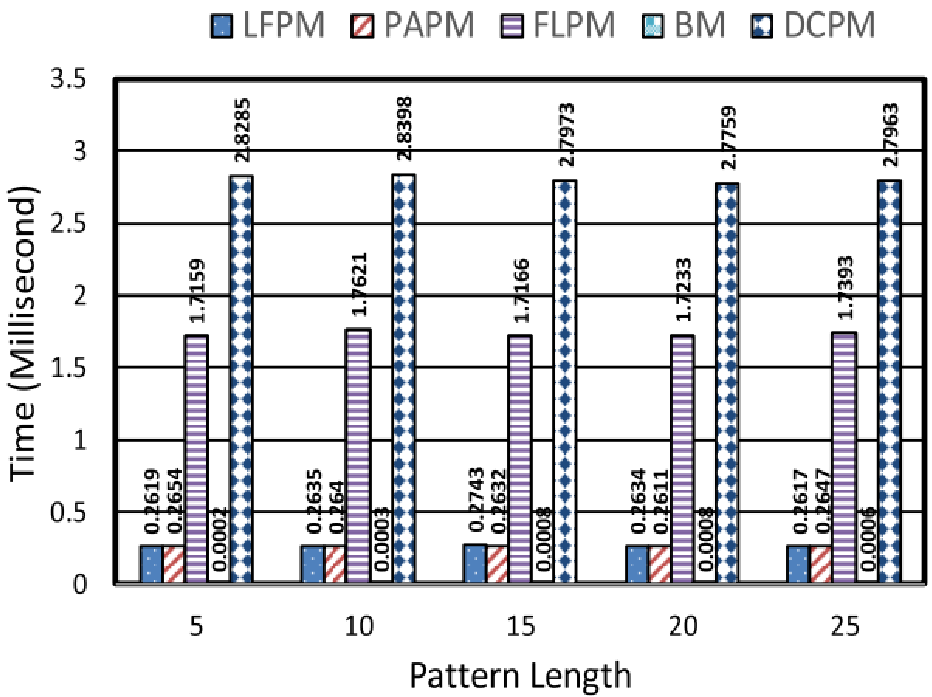
- Brute Force (BF)
- Boyer-Moore (BM)
- Divide and Conquer Pattern Matching (DCPM)
- FLPM
- PAPM
- LFPM

The specifications of the computing environment →

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

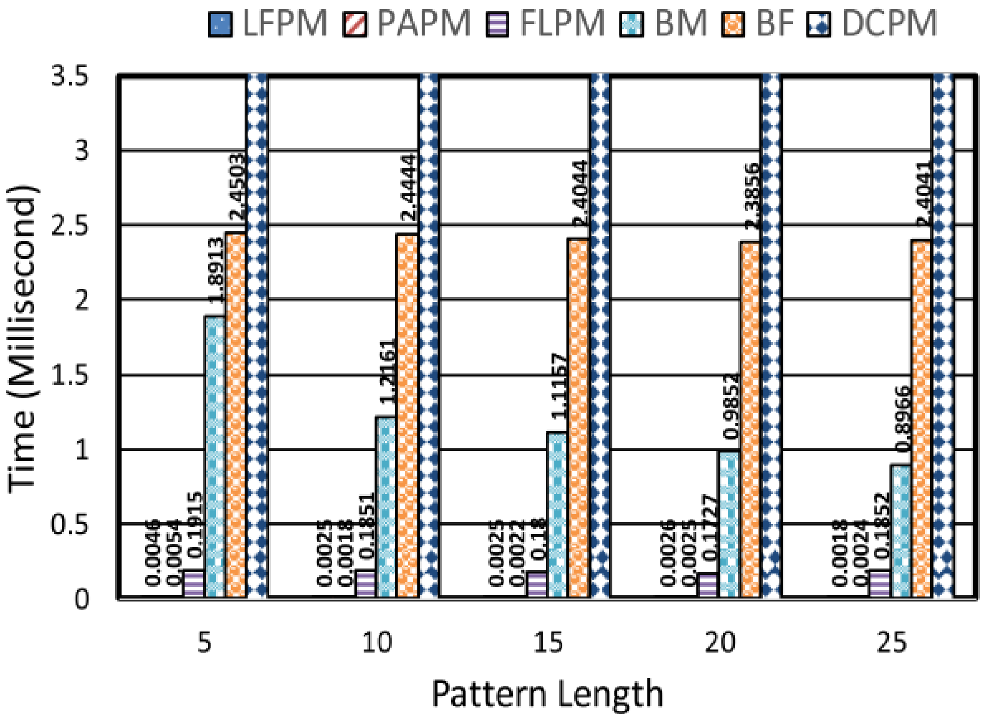
Results

A. THE TIME OF PREPROCESSING PHASE



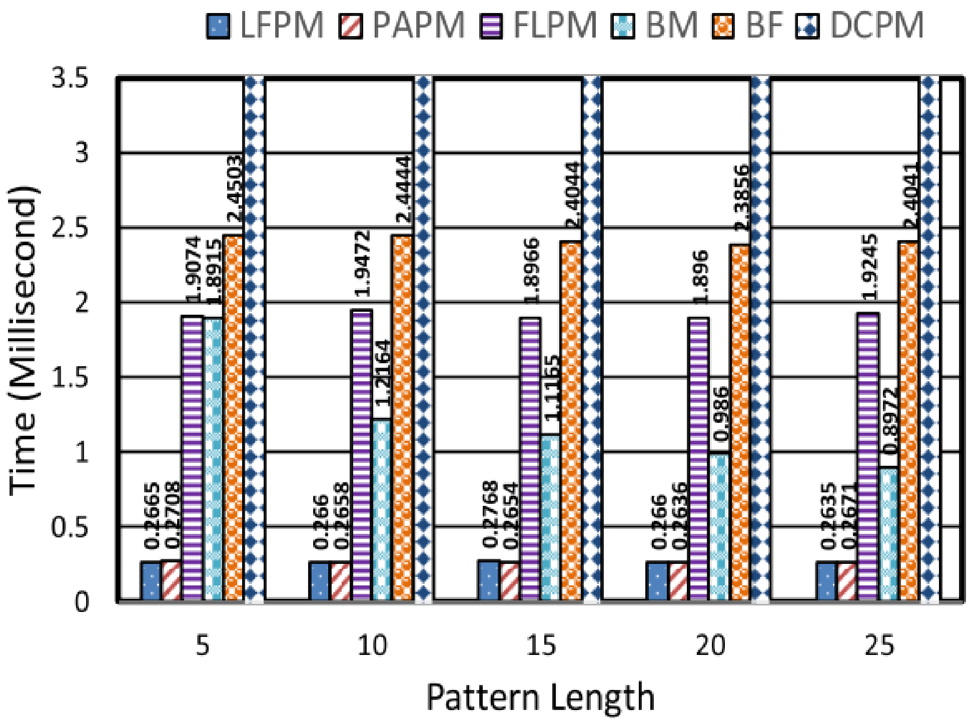
Results

B. THE TIME OF MATCHING PHASE

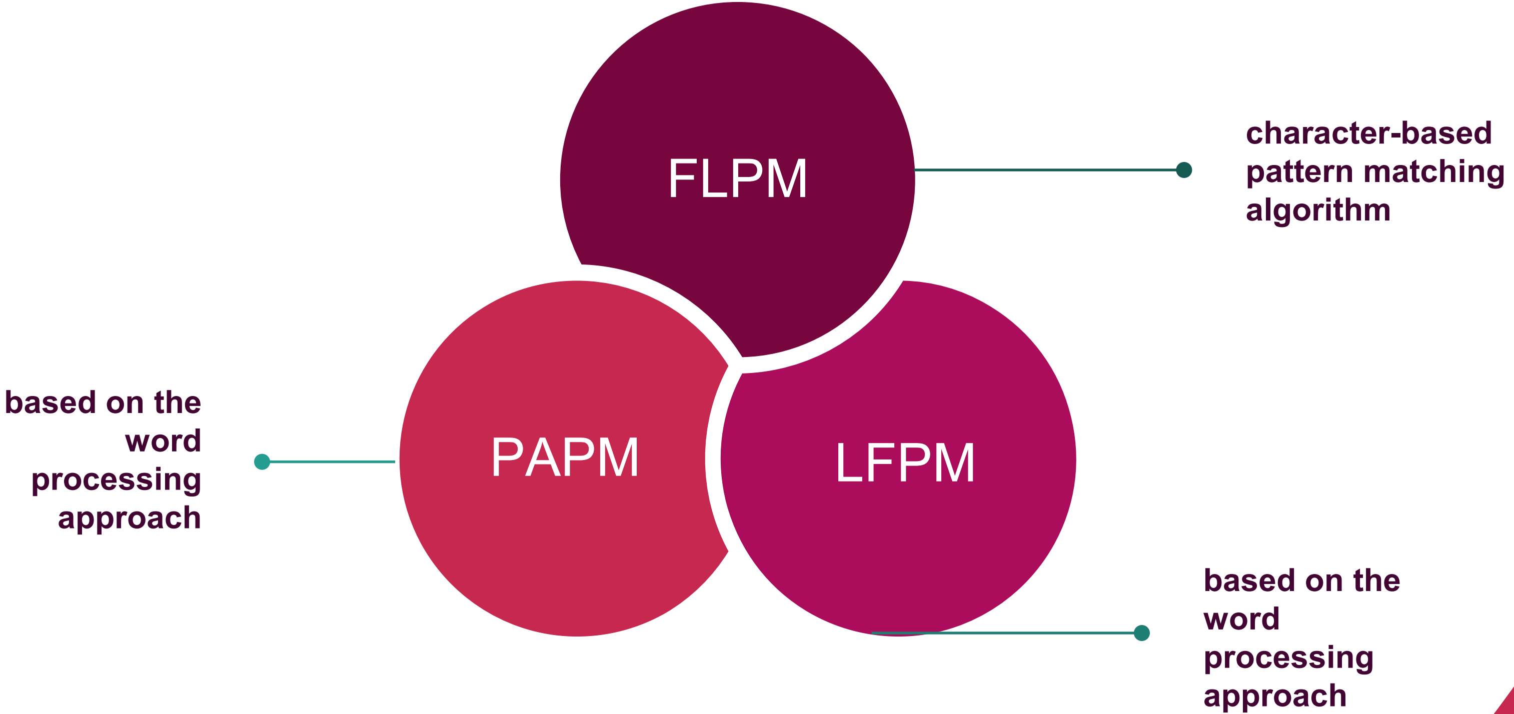


Results

C. TOTAL TIME OF PATTERNS



Conclusion


FLPM
character-based pattern matching algorithm
PAPM
based on the word processing approach
LFPM
based on the word processing approach

Future Research

- A parallel version of the presented algorithms
- The algorithms supporting approximate matching

Reference

- Neamatollahi, Peyman, Montassir Hadi, and Mahmoud Naghibzadeh. "Simple and efficient pattern matching algorithms for biological sequences." IEEE Access 8 (2020): 23838-23846.
- SAQIB HAKAK1, Amirrudin Kamsin1, PALAIAHNAKOTE SHIVAKUMARA1, GULSHIN “Exact String Matching Algorithms: Survey, Issues, and Future Research Directions”
- https://en.wikipedia.org/wiki/Boyer–Moore_string-search_algorithm
- https://en.wikipedia.org/wiki/Knuth–Morris–Pratt_algorithm
- https://en.wikipedia.org/wiki/Brute-force_attack

Thanks!

Any questions?

- Nazanin Fereydoonizade
- Sarvin Saravi