String matching algorithm based on the middle of the pattern

Bakoss Janan Jan
Al-Zaytoonah University, Jordan
jenanbakoss@yahoo.com

Oqeili Saleh
Al-Balqa' Applied University, Jordan
saleh@bau.edu.jo

ABSTRACT

In this paper, three new algorithms for string matching that depend on comparison starting at the middle of the pattern are proposed. In case of matching, then we compare the left middle and the right middle to obtain matching string. In addition, the pattern is preprocessed to obtain a failure array similar to that in the KMP algorithm. The proposed algorithms are compared with other algorithms by simulation. Different sets of data of variable size are used and recommendations are given to show when to use each of the proposed methods.

Key Words: String matching algorithms; Pattern matching; Finite automaton; KMP algorithm

1. Introduction

The context of the problem is to find out whether one string (called "pattern") is contained in another string. This problem corresponds to a part of more general one, called "pattern recognition". The strings considered are sequences of symbols, and symbols are defined by an alphabet. The size and other features of the alphabet are important factors in the design of string-processing algorithms.

2. Brute-force algorithm

Starting at the beginning of each string, we compare characters, one after the other, until either the pattern is exhausted or a mismatch is found. In the former case we are done; a copy of the pattern has been found in the text. In the latter case we start again, comparing the first pattern character with the second text character. In general, when a mismatch is found, we (figuratively) slide the pattern one more place forward over the text and start again, comparing the first pattern character with the next text character. The time complexity of this searching phase is O(mn).

3. THE Knuth – Morris Pratt (KMP) Algorithm

The Knuth–Morris–Pratt string searching algorithm searches for occurrences of a "word" *W* within a main "text string" *S* by employing the simple observation that when a mismatch occurs, the word itself embodies sufficient information to determine where the next match could begin, thus bypassing re-examination of previously matched characters. The searching phase can be performed in O(m+n) time.

4. Propose work

In this paper, we will discuss three new algorithms that depend on comparison starting at the integer of the middle of the pattern. In case of matching, we will compare the left middle and the right middle to obtain matching string.

4.1 Middle Pattern that involve preprocessing function in Left and Right side (MPLR) Algorithm

MPLR Algorithm that depends on middle of the pattern compared with the text to matching string, if the middle of the pattern is matching with a text then we compare left middle and right middle to obtain matching string, if the mismatch occurred in the left or right side middle of the pattern, we would use preprocessing function (failure function) like KMP algorithm. This algorithm is constructed by starting comparison in the integer middle of pattern with the position of the text. At the beginning, the position of the text is equal to the position of the pattern, if the integer middle of the pattern matches then we will compare left middle beginning with utmost left to (middle-1) and right middle beginning with (middle+1) to rightmost to obtain matching string, figure 1 shows the flowchart of this algorithm and figures 2 illustrated the shift of window to the right for this algorithm when mismatch occurs.

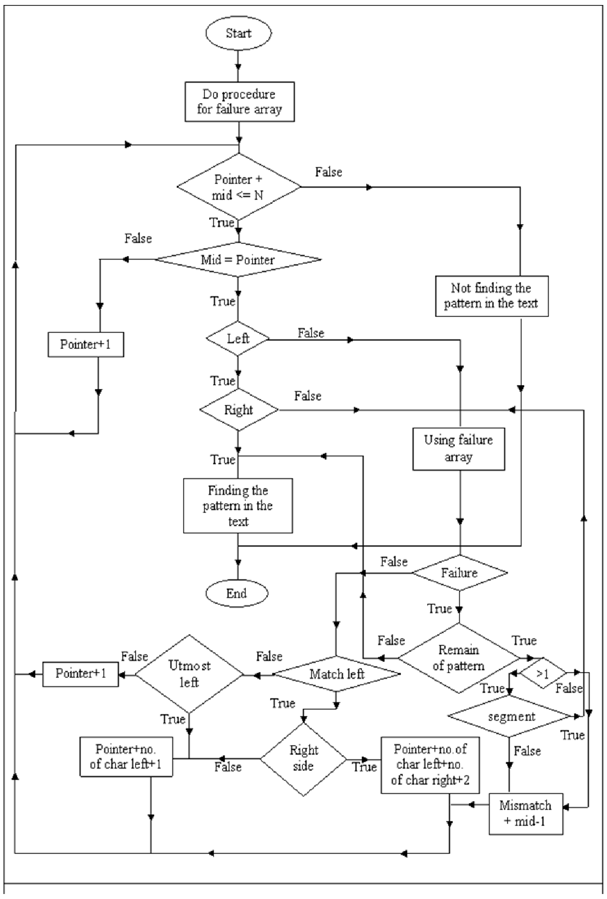
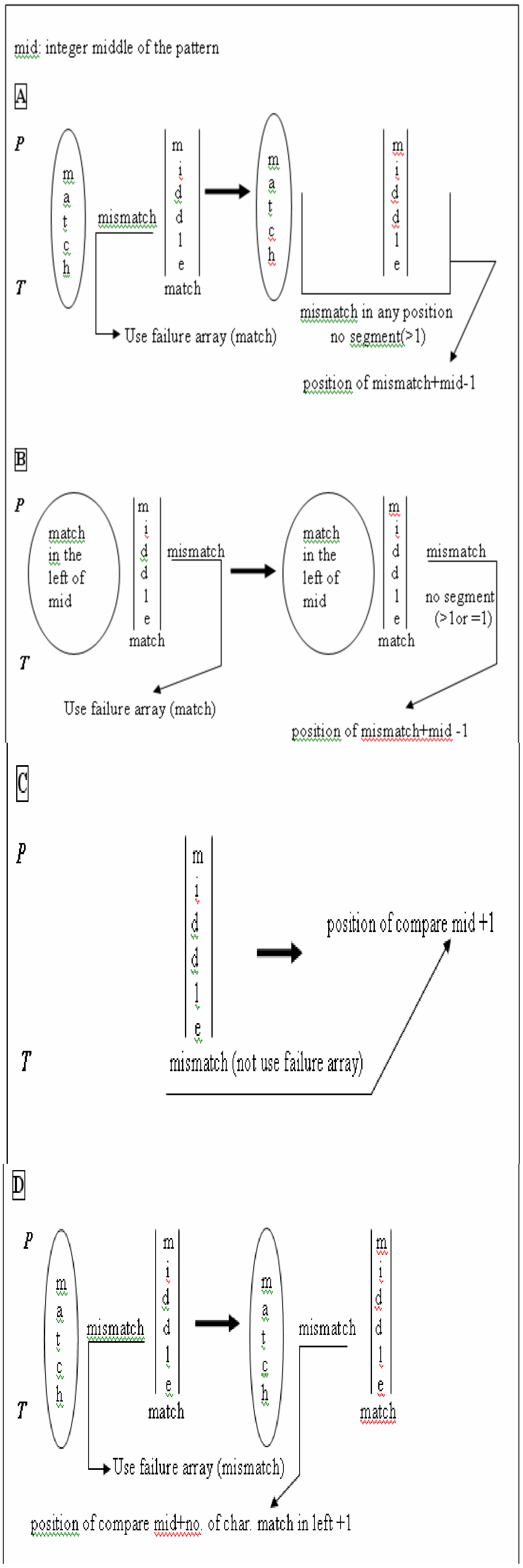
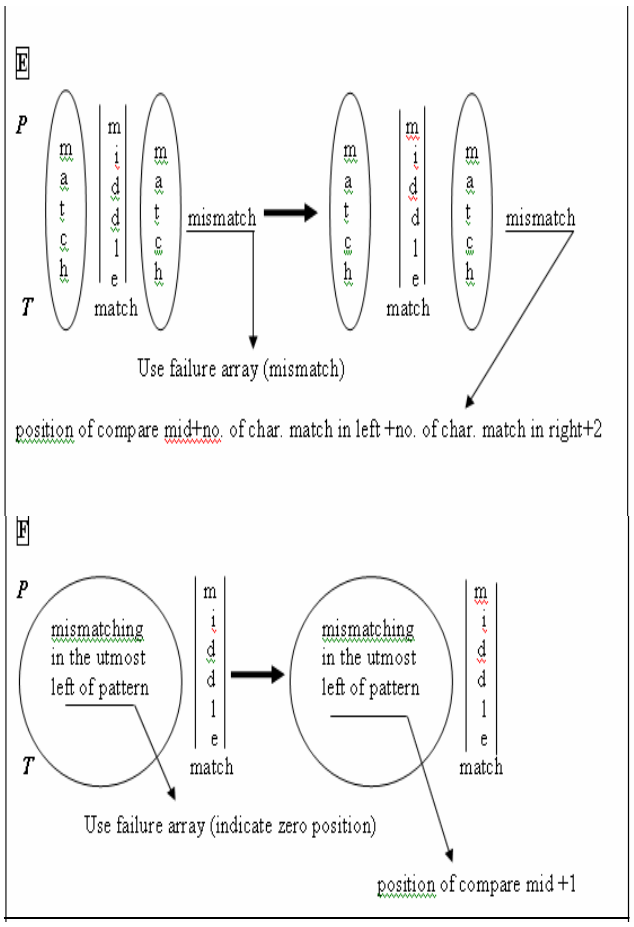


Figure 1: Flowchart of MPLR algorithm.



mid: integer middle of the pattern
A
P
T
match
mismatch
middle
match
match
middle
Use failure array (match)
mismatch in any position
no segment(>1)
position of mismatch+mid-1
B
P
T
match in the left of mid
middle
mismatch
match
match in the left of mid
middle
mismatch
no segment (>1or =1)
match
Use failure array (match)
position of mismatch+mid -1
C
P
T
middle
position of compare mid +1
mismatch (not use failure array)
D
P
T
match
mismatch
middle
match
match
mismatch
middle
match
Use failure array (mismatch)
position of compare mid+no. of char. match in left +1



E
P
match
middle
match
mismatch
match
middle
match
mismatch
T
match
match
Use failure array (mismatch)
position of compare mid+no. of char. match in left +no. of char. match in right+2
F
P
mismatching in the utmost left of pattern
middle
mismatching in the utmost left of pattern
middle
T
match
match
Use failure array (indicate zero position)
position of compare mid +1

Figure 2: Sliding the pattern to line up a matched substring of MPLR algorithm.

We show MPLR algorithm how can it work by taken text (1 0 0 1 1 1 0 1 0 0 1 0 1 0 0 0 1 0 1 0 0 1 1 1 0 0 0 1 1 1) and pattern (1 0 1 0 0 1 1 1), the failure array illustrates in table 1 for this pattern.

Table1: Fail table for *P* '10100111'

Fail	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
	0	**1**	**1**	**2**	**3**	**1**	**2**	**2**

Text: 100111010010100010100111000111
Pattern: 10100111

100111010010100010100111000111
101$\boxed{0}$0111
101$\boxed{0}$0111
101$\boxed{0}$0111
1$\boxed{0}$100111
1$\boxed{0}$100111
101001$\boxed{1}$1
10100$\boxed{1}$11
101$\boxed{0}$0111
1001110100101000$\boxed{10100111}$000111

Analysis

In this algorithm, we have confronted the problem that the pattern contains one segment of the characters. This problem was solved by the continuous use of the preprocessing function (failure array) until mismatching happens that changes the status from one segment to no segment of characters so the algorithm is designed, in this case, it will shift the window to the right magnitude of the position of the text that the mismatch is happened in it adding for it the integer middle of the pattern minus one, this case is similar to the text "baabababababacaab" with the pattern "ababaca".

The first part for computed failure array takes O(2m) ≈ O(m) time, the second part when scanning the pattern for MPLR algorithm, the time complexity takes firstly if the integer middle of the pattern is not existed between the text so the number of comparisons is (n-(m-1)) = (n-m+1), the time complexity is O(n-m) like this case of the text 'abcdefghijklmnopq' with the pattern 'abczdefg', secondly if the integer middle of the pattern is matching but the utmost left is not matched so the number of comparisons is (n-(m-1))*2 = (n-m+1)*2, the time complexity is O(n-m) like this case of the text 'aaaaaaaaaaaaaa' with the pattern 'baaaaaaa', and the worst-case running time for this algorithm is O(m+n) like KMP algorithm the case of text 'aaaaaaabaaaaaaabaaaaaaabaaaaaaaa' with the pattern 'aaaaaaaa'.

4.2 Middle Pattern that involve preprocessing function in Left side (MPL) Algorithm

The idea of comparison for MPL algorithm, firstly, the algorithm compares the integer middle of the pattern, if the attempt successfully matches, the algorithm will be designed to compare the left side beginning from the utmost left to (middle-1), if mismatch happened there, we would use the preprocessing function (failure array), otherwise, if the attempt successfully matches in the left of the middle of the pattern, the beginning of the right side of the middle of the pattern will be compared from (middle+1) to the rightmost, if the mismatch happens there, it will shift the window to the right magnitude one, otherwise, the attempt successfully matches.

Figure 3 shows the flowchart of this algorithm and figure 4 illustrating the shift of window to the right of this algorithm when mismatch occurs.

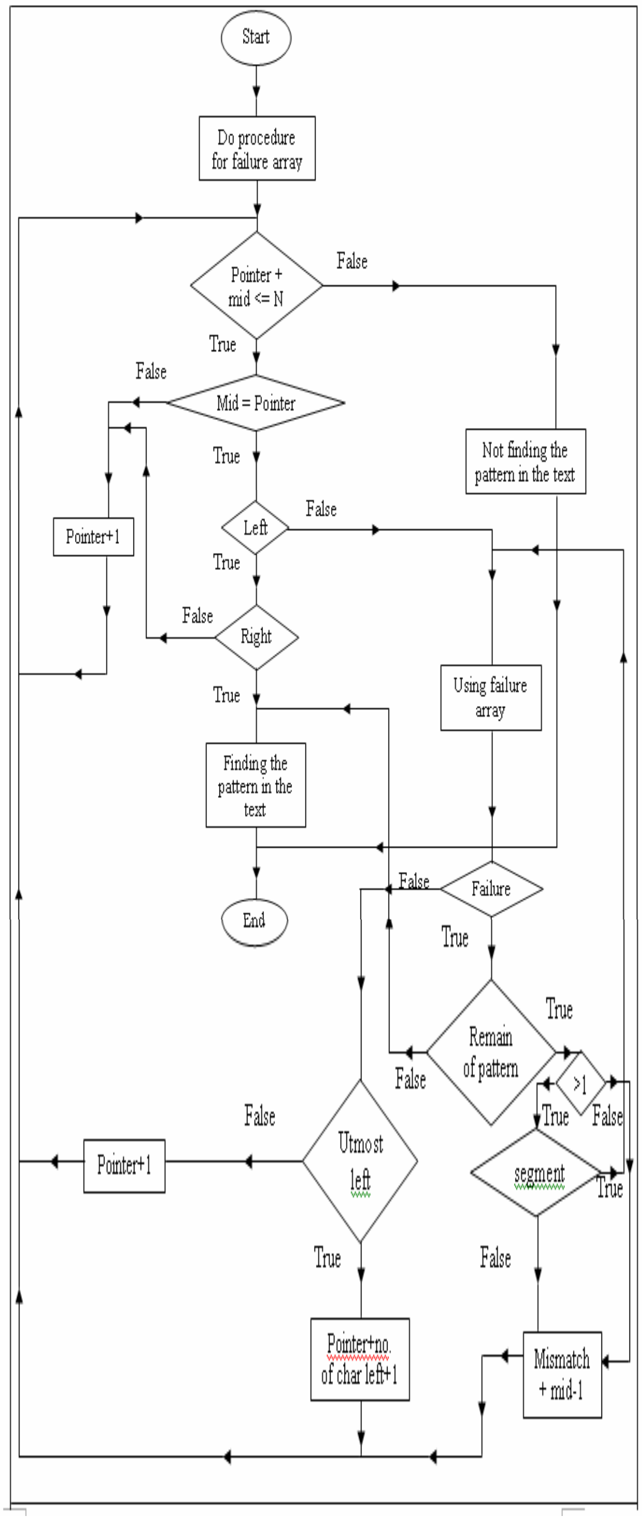
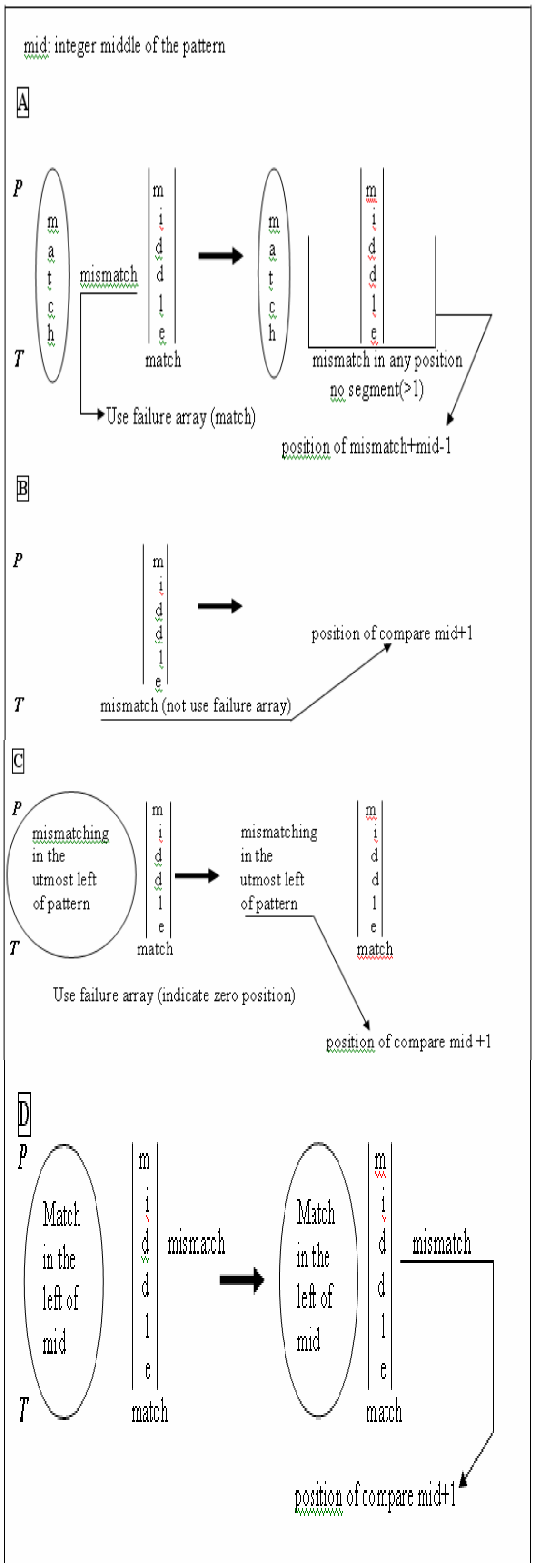
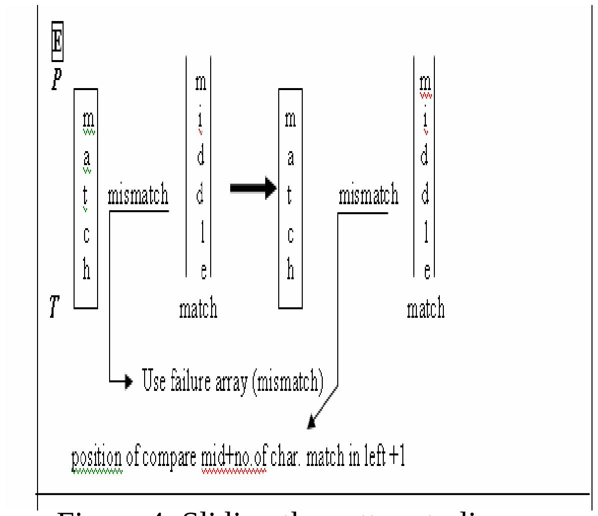


Figure 3: Flowchart of MPL algorithm.



mid: integer middle of the pattern
A
P
T
match
mismatch
middle
match
mismatch in any position
no segment(>1)
Use failure array (match)
position of mismatch+mid-1
B
P
T
middle
mismatch (not use failure array)
position of compare mid+1
C
P
T
mismatching in the utmost left of pattern
middle
match
Use failure array (indicate zero position)
position of compare mid +1
D
P
T
Match in the left of mid
middle
mismatch
match
position of compare mid+1


E
P
match
mismatch
middle
match
T
match
mismatch
middle
match
Use failure array (mismatch)
position of compare mid+no.of char. match in left+1
Figure 4: Sliding the pattern to line up a matched substring of MPL algorithm.

We show MPL algorithm how can it work by taken text (1 0 0 1 1 1 0 1 0 0 1 0 1 0 0 0 1 0 1 0 0 1 1 1 0 0 0 1 1 1) and pattern (1 0 1 0 0 1 1 1).

Text: 100111010010100010100111000111
Pattern: 10100111

100111010010100010100111000111
101$\boxed{0}$0111
101$\boxed{0}$0111
101$\boxed{0}$0111
1$\boxed{0}$100111
1$\boxed{0}$100111
101001$\boxed{1}$1
$\boxed{1}$0100111
101$\boxed{0}$0111
$\boxed{1}$0100111
101$\boxed{0}$0111
10100$\boxed{1}$11
$\boxed{1}$0100111
10$\boxed{1}$00111
$\boxed{1}$0100111
101$\boxed{0}$0111
1001110100101000$\boxed{10100111}$000111

Analysis
When the mismatch happens in the right side of the middle of the pattern, the algorithm will shift the window to the right magnitude one, this makes to overload in algorithm and leads the algorithm to worse case.
The first part of computed failure array takes O(2m) ≈ O(m) time, the second part scans the pattern of MPL algorithm the time complexity takes, firstly if the integer middle of the pattern is not existed between the text, the number of comparison will be (n-(m-1)) = (n-m+1) and the time complexity is O(n-m) such as this text 'abcdefghijklmnopq' with pattern 'abczdefg', secondly, if the integer middle of the pattern is matching but the utmost left is not matching, the number of comparison is (n-(m-1))*2 = (n-m+1)*2 and the time complexity is O(n-m) such as this text 'aaaaaaaaaaaaaaaaaa' with pattern 'baaaaaaa', and the worst-case of running time for this algorithm is O(nm) like Brute-Force algorithm, this case is applying on the text 'aaaaaaaaaaaaaaaaaaaaab' with the pattern 'aaaaaaaab'.

4.3 Middle Pattern that involve preprocessing function in Right side (MPR) Algorithm

This algorithm is similar to MPLR algorithm but there are some differences between them, it is begins by comparing from the integer middle of the pattern like MPLR and MPL algorithms, it uses preprocessing function (failure array) when mismatch occurs only in the right side of the middle of the pattern.
Figure 5 shows the flowchart of this algorithm and figure 6 illustrating the shift of window to the right of this algorithm when mismatch occurs.

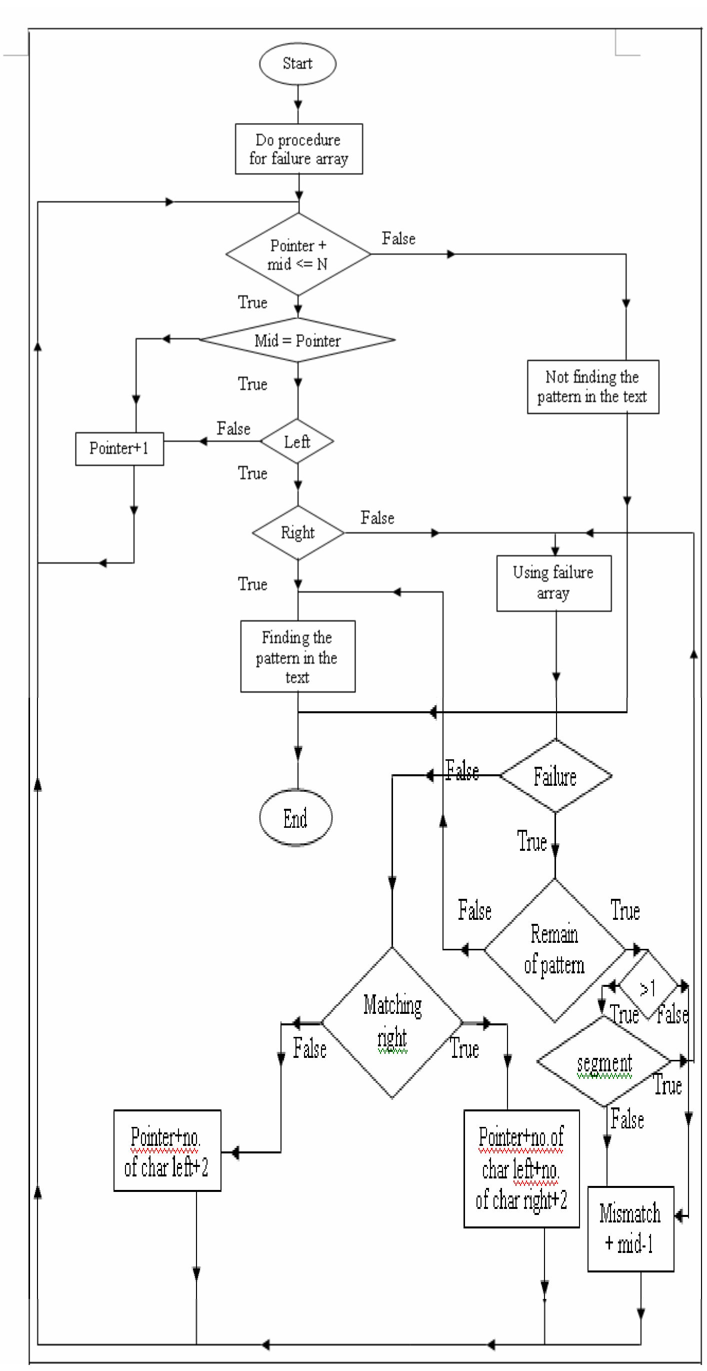
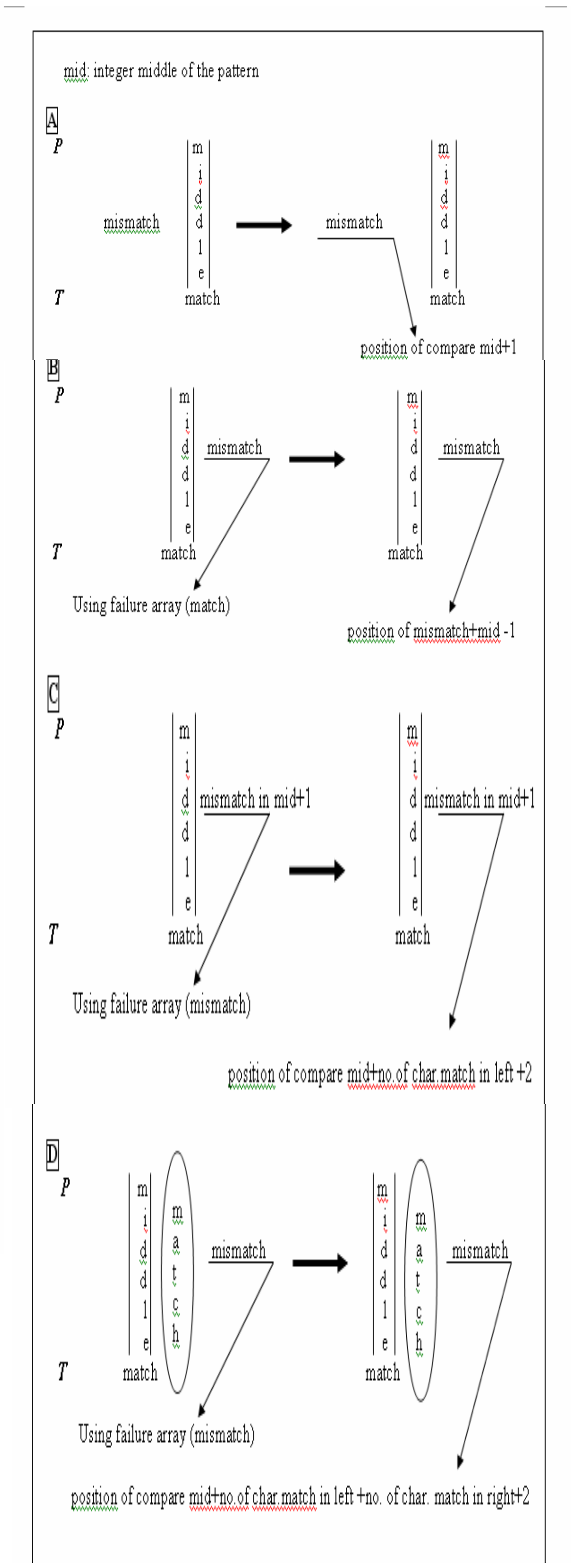


Figure 5: Flowchart of MPR algorithm.


mid: integer middle of the pattern
A
P
T
mismatch
middle
match
mismatch
middle
match
position of compare mid+1
B
P
T
middle
mismatch
match
Using failure array (match)
middle
mismatch
match
position of mismatch+mid -1
C
P
T
middle
mismatch in mid+1
match
Using failure array (mismatch)
middle
mismatch in mid+1
match
position of compare mid+no.of char match in left +2
D
P
T
middle
match
mismatch
match
Using failure array (mismatch)
middle
match
mismatch
match
position of compare mid+no.of char match in left +no. of char. match in right+2

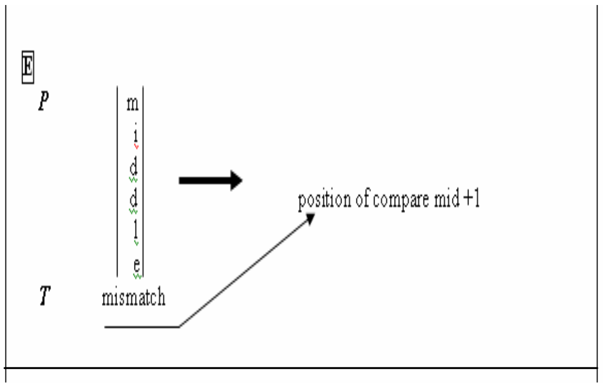


Figure 6: Sliding the pattern to line up a matched substring of MPR algorithm.

We show MPR algorithm how can it work by taken text (1 0 0 1 1 1 0 1 0 0 1 0 1 0 0 0 1 0 1 0 0 1 1 1 0 0 0 1 1 1) and pattern (1 0 1 0 0 1 1 1).

Text: 100111010010100010100111000111
Pattern: 10100111

$\mathbf{100111010010100010100111000111}$
$\mathbf{101\boxed{0}0111}$
$\mathbf{101\boxed{0}0111}$
$\mathbf{101\boxed{0}0111}$
$\mathbf{1\boxed{0}100111}$
$\mathbf{101\boxed{0}0111}$
$\mathbf{101001\boxed{1}1}$
$\mathbf{10100\boxed{1}11}$
$\mathbf{101\boxed{0}0111}$
$\mathbf{1001110100101000\boxed{10100111}000111}$

Analysis

When the mismatch occurs in the left side of the middle of the pattern, the algorithm will shift the window to the right magnitude one, it makes to overload in algorithm and leads the algorithm to worse case.

The first part for computing failure array takes O(2m) ≈ O(m) time, the second part, when scanning the pattern of MPR algorithm the time complexity takes, firstly, if the integer middle of the pattern is not existed between the text so the number of comparisons is (n-(m-1)) = (n-m+1), the time complexity is O(n-m) such as this case of the text 'abcdefghijklmnopq' with the pattern 'abczdefg', secondly, if the integer middle of the pattern is matching but the utmost left is not matching so the number of comparisons is (n-(m-1))*2 = (n-m+1)*2, the time complexity is O(n-m) such as this case of the text 'aaaaaaaaaaaaaaaaaa' with the pattern 'baaaaaaa', we will suppose that the integer middle of the pattern symbolize (µ) so the worst-case of running time for this algorithm is O(nµ), this case is applicable in the text 'aaaaaaaaaaaaaaaabaaaaa' with the pattern 'aabaaaaa'.

5. Results

We tested the new algorithms by taking the data between 26 to 90 character and various sizes of data (1000, 10000 and 20865 character), new algorithms proved that they are better than Brute-force algorithm and Knuth-Morris-Pratt algorithm, especially when we take the various sizes of data (1000, 10000 and 20865 character), the percentage becomes high within the range 1000 character more than any other range of the various sizes of data, new algorithms is more efficient than Brute-force algorithm and Knuth-Morris-Pratt algorithm.

6. Conclusion

The proposed new algorithms are better than Brute-Force algorithm in selected text (26 to 90 character). In general, the new algorithms are better than Brute-Force algorithm and Knuth-Morris-Pratt algorithm, when we take the various sizes of data (1000, 10000 and 20865 character). The MPLR algorithm is often better than Knuth-Morris-Pratt algorithm in selected text (26 to 90 character), in case, if the number of comparisons of Knuth-Morris-Pratt algorithm is less than MPLR algorithm; the differences between them will be less than the length of pattern, but when we take the various sizes of data (1000, 10000 and 20865 character), the differences between them are several, and the MPLR algorithm become better. Whenever the length of pattern is long; so the new algorithms become efficient.

The worst-case of running time of the MPLR algorithm is O(m+n) like KMP algorithm. The worst-case of running time of the MPL algorithm is O(nm) like Brute-Force algorithm. The worst-case of running time of the MPR algorithm is O(nμ).

References

[1] Baase, Sara, Computer algorithms: introduction to design and analysis, Addison – Wesley, Reading, MA, 1988, 2nd edition.

[2] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., and Stein, Clifford, Introduction to algorithms, MIT Press, Boston, 2001, 2nd edition.

[3] Johnsonbaugh, Richard, and Schaefer, Marcus, Algorithms, Pearson/Prentice-Hall, Upper Saddle River,NJ, 2004.

[4] Sedgewick, Robert, Algorithms, Addison – Wesley, Reading MA, 1988, 2nd edition.

[5] Smyth, Bill, Computing patterns in string, Addison – Wesley, Harlow, 2003.