Novel Pattern Matching Algorithm for Single Pattern Matching

Devaki Pendlimarri
Dept. of Computer Applications,
Swarnandhra Institute of Engineering and Technology,
Narsapur, INDIA.

Paul Bharath Bhushan Petlu (corresponding author)
Dept. of Computer Applications,
Swarnandhra College of Engineering and Technology,
Narsapur, INDIA.

***Abstract*– Pattern matching is one of the important issues in the areas of network security and many others. The increase in network speed and traffic may cause the existing algorithms to become a performance bottleneck. Therefore, it is very necessary to develop more efficient pattern matching algorithm, in order to overcome troubles on performance. There are several algorithms in use, in which, some are with different methodology and other are with the improved methodology for the existing algorithms. In this paper, we are proposing a novel pattern matching algorithm, called, DP algorithm (Devaki – Paul algorithm). The algorithm works basing on some novel set of innovated rules, which will endorse the algorithm resulting in better performance and efficiency. In case of unsuccessful search, the DP algorithm has zero character comparisons, irrespective of the sizes of the text and pattern, provided if either the first or the last character was not present in the given input text. Whereas, the Booyer-Moore and Quick Search algorithms will do search as usual. The algorithm also doesn't require any pre-processing phase, if the search is on the same given input text and with different patterns, provided the first and the last characters are same as in the case of first pattern. The algorithm was tested and validated and the results have proved that the performance of DP algorithm is better than BM algorithm (Boyer – Moore algorithm) and Quick Search algorithm. In case of tests with repeated character, its performance is greater than 1%~50% with BM and Quick Search algorithms. In case of tests with the English Text and Random Pattern, it's greater than 33%~91% with BM and 37%~85% with Quick Search algorithms. In case of tests with the English Text and Random Pattern of an unsuccessful search, its performance is greater than BM and Quick Search algorithms with 100%, if either the first and/or the last character of the pattern in the given text were not present.**

***Key words* – DP algorithm, single pattern matching, Boyer – Moore algorithm, Quick Search algorithm**

I. INTRODUCTION

Pattern matching is one of the basic and most important issues, which have been studied, in the research areas of computer science. In a standard problem, we are required to find all occurrences of the pattern in the given input text, known as single pattern matching [6]. Suppose, if more than one pattern are matched against the given input text simultaneously, then it is known as, multiple pattern matching. Here, we are presenting a novel pattern matching algorithm, which will find all the occurrences of a pattern in the given input text. Single pattern matching algorithm is widely used in network security environments. In network security realm, the pattern is a string indicating a network intrusion, attack, virus, and snort, spam or dirty network information, etc [9]. For example, snort[3] and Bro[4] is an open source network intrusion prevention and detection system (IDS/IPS) developed by *sourcefire*.

Since the evolution of the Boyer – Moore (BM) [1] and the Knuth – Morris – Pratt (KMP) [12], [13] algorithms, many more techniques were proposed by many researchers, to find the exact pattern matching, by improving the performance and efficiency. The BM algorithm is considered as one of the most efficient pattern matching algorithm in general applications and is also known as the best average-case performance algorithm of any algorithm [1]-[2], [5], [8]. This algorithm requires a preprocessing phase of $O(m+\sigma)$ time, and a search phase of $O(mn)$ time complexity. The algorithm requires a preprocessing of the pattern before starting the search, which constructs a table. A 256 member table is constructed that is initially filled with the length of the pattern. The 256 members represent the full range of characters in the ASCII character set. A second pass is then made on the table that places a descending count from the original length of the pattern in the ASCII table for each character that occurs [2]. To find the occurrence of a pattern in the given input text, the algorithm scans the characters of the pattern in the text from right to left, beginning with the rightmost character. In case of a match, it continues matching the character of the pattern with the characters of the text sequentially from right to left, until the match is complete. In case of a mismatch, it uses two pre-computed heuristics to shift the window to the right. These two heuristics are: good-suffix and bad character-shift. Both heuristics are triggered on a mismatch.

Quick Search algorithm [2], is one simplification of the BM algorithm, which uses only the bad character heuristic and also easy to implement. This algorithm requires a preprocessing phase of $O(m+\sigma)$ time, and a search phase of $O(mn)$ time complexity. The algorithm uses only the bad-character shift table. After an attemp where the window is positioned on the text factor, $y[j\,..\,j+m-1]$, the length of the shift is atleast equal to one. So, the character $y[j+m]$ is necessarily

involved in the next attempt, and thus can be used for the bad-character shift of the current attempt. The bad-character shift of the BM algorithm is slightly modified to take into account the last character of *x* as follows: for *c* in $\sum$, $qsBc[c] = \min\{i: 0 \leq i < m \; and \; x[m-1-i] = c\}$, if *c* occurs in *x, m* otherwise. The algorithm is very fast in practice for short patterns and large alphabets.

In this paper, we propose a novel methodology, to improve the performance in the pattern matching. This algorithm requires a preprocessing phase of *O(m)* time, where m is size of the given input text, the search phase time is directly proportional to the size of the table of occurrences of preprocessing phase.

II. METHODOLOGY

A novel pattern matching algorithm, called, Devaki – Paul algorithm (DP algorithm), is presented here. This algorithm requires a preprocessing phase, which prepares a table of occurrences of the first and the last characters of the pattern in the given input text. The preprocessing phase time complexity of the DP algorithm is less than the BM and the Quick Search algorithms. The preprocessing phase time complexity of the DP algorithm is compared with the BM and Quick Search algorithms and is presented in the table *tab 1*, where *m* is the size of the given input text.

TABLE 1: Preprocessing phase time complexity

S. No.	Algorithm	Time complexity
1	DP	*O(m)*
2	BM	*O(m+σ)*
3	Quick Search	*O(m+σ)*

A. Preprocessing Phase

In this phase, we find the occurrences of the first and last characters of the pattern in the given input text. Here, we will get two cases: first and last characters of the pattern are similar and the other, dissimilar. In the first case we use algorithm *Similar*, otherwise, *Different* as below:

Similar*(char x[], int m, char y[], int n)*

/ Preparing a table of occurrences of the first character of the pattern in the given input text*/*

Step 1: [initialization]

Initialize the index and other variables

Step 2: [find the first character occurrences]

Find all the occurrences of the first character of the pattern in the given input text

Step 3: [Finish]

return

Different*(char x[], int m, char y[], int n)*

/ Preparing a table of occurrences of the first and the last characters of the pattern in the given input text*/*

Step 1: [initialization]

Initialize all index and other variables

Step 2: [find the first and last character occurrences]

Find all the occurrences of the first and the last characters of the pattern in the given input text

Step 5: [Finish]

return

Then, it performs a search phase based on the pre-computed table with a set of rules.

B. Search Phase

In this phase, we find the probability of having an occurrence of a pattern in the given input text by using the table of occurrences of pre-processing phase.

Search*(char x[], int m, char y[], int n,int a[], int alen, int b[], int blen)*

/ Search Phase: This algorithm will find the chances of getting pattern match */*

Step 1: [initialing the variables]

Initialize all the index and other variables

Step 2: [find the probability of occurrence of a pattern and do match]

Repeat step 3 until the end of the table of last character occurrences

Step 3: [calculate the difference between the last and first character occurrence]

Case 1: difference > n – 1

Update the index of the table of first character occurrence

Go to step 2

Case 2: difference < n – 1

Update the index of the table of last character occurrence

Go to step 2

Case 3: difference = n – 1

Match()

Update the indices of both tables of first and last character occurrence

Go to step 2

Step 3: [finish]

return

In the above algorithm, once we find the probability of occurrence of a pattern in the given input text, we perform an exact pattern matching algorithm.

C. Exact Pattern Matching

This algorithm will find that whether the probability will lead to either successful or unsuccessful search. As already the first and the last characters of the pattern were compared with the given input text and found equal, the number of character comparisons can be reduced by two.

Match*(char x[], char y[], int first, int size)*

/ This algorithm will conclude that, whether a found probability results to an exact pattern match or not. As we already found that the first and the last characters of the pattern are equal in the text, the algorithm reduces two more character comparisons. */*

Step 1: [initializing]

Initialize the index variable

Step 2: [initiate the process of finding the exact match]

Repeat step 3 until the i <= size

Step 3: [perform character comparisons]

Compare each character in the pattern with the character in the text sequentially

If (characters do not match) then

Go to step 4

Step 4: [verify]

If (i >size) then

OUTPUT(pattern)

Step 5: [Finish]

return;

III. IMPLEMENTATION

The DP algorithm requires a preprocessing of the given text to prepare a table of the occurrences of the first and the last characters of the given pattern. This table is used to find the probability of having a match of the pattern in the given text, which reduces the number of comparisons, improving the performance of the pattern matching algorithm. The probability of having a match of the pattern in the given text is mathematically proved.

A. Mathematical Proof:

Here, we get two cases: first and last characters of the pattern in the given input text may be of similar or dissimilar.

Case 1: If the first and the last characters of the Patterns are similar

If the difference between any two occurrences of the first character of the pattern in the pre-computed table is less than the size of the pattern by one, then, it is taken as one probability for occurrence of an exact pattern match.

Let, *x*, be the given text of size *m* and *y*, be the given pattern of size *n*, where $m \geq n$. Let us assume that $A[a1, a2, \ldots, ai]$, is an array, in which, $a1, a2, \ldots, ai$, represents the occurrences of the first character of given pattern, *y*, in the given text, *x*, where, $0 \leq i \leq m$.

We know that,

y[1] = the position of the first character in the pattern

y[n] = the position of the last character in the pattern

Hence, the offset between the last character and the first character in the pattern is:

$$offset = n - 1 \qquad (1)$$

Now, from the pre-computed table,

If *A[i]* = a_i = one of the positions of the first character of the pattern in the given input text then the position of the last character of the pattern in the given input text in case of exact pattern match must be:

$$lastpos = a[i] + offset \qquad (2)$$

Let, if *a[j]* = another position of the last character of the pattern in the given input text, where $i < j \leq alen$ then,

$$lastpos = a[j] \quad (3)$$

From (2) & (3), we get

$$a[j] = a[i] + offset$$

$$=> a[j] - a[i] = offset$$

Substituting (1), we get

$$a[j] - a[i] = n - 1 \quad (4)$$

Hence, we proved that, any condition which satisfies with the (4), is a probability of occurrence of a pattern.

Case 2: If the first and the last characters of the Patterns are dissimilar

If the difference between any two occurrences of the last and the first characters of the pattern in the pre-computed table is less than the size of the pattern by one, then, it is taken as one probability for occurrence of an exact pattern match.

Let, *x*, be the given text of size *m* and *y*, be the given pattern of size *n,* where $m \geq n$. Let us assume that, $A[a1, a2, \ldots, ai]$, is an array, in which, $a1, a2, \ldots, ai$, represents the occurrences of the first character of the given pattern, *y*, in the given input text, *x*, where, $0 \leq i \leq m$. Similarly,$B[b1, b2, \ldots, bj]$, is an array, in which, $b1, b2, \ldots, bj$, represents the occurrences of the last character of the given pattern, *y,* in the given input text, *x*, where, $0 \leq j \leq m$.

We know that,

y[1] = the position of the first character in the pattern

y[n] = the position of the last character in the pattern

then, the offset between the last character and the first character in the pattern is:

$$offset = n - 1 \quad (5)$$

Now, from the pre-computed table,

If *A[i]* = a_i = one of the positions of the first character of the pattern in the given input text then the position of the last character of the pattern in the given input text in case of exact pattern match must be:

$$lastpos = a[i] + offset \quad (6)$$

Let, if *b[j]* = one of the positions of the last character of the pattern in the given input text then,

$$lastpos = b[j] \quad (7)$$

From (6) & (7), we get

$$b[j] = a[i] + offset$$

$$=> b[j] - a[i] = offset$$

Substituting (5), we get

$$b[j] - a[i] = n - 1 \quad (8)$$

Hence, we proved that, any condition which satisfies with the (8), is a probability of occurrence of a pattern.

B. Description

As specified in the methodology, we may have patterns of two cases: *first and last characters of the pattern are similar and the other is different.*

Case 1: If the first and the last characters of the Patterns are similar

Let us assume that the example text and patterns for the above case are as given in Fig 1.

The table of occurrences of the first character of the pattern in the given input text will be calculated using the *Similar* algorithm as in the methodology and was obtained as above. After the pre-processing phase a search phase begins, where we use *Search* algorithm as given in the methodology, to find the probability of a pattern match. Here, we get three possibilities:

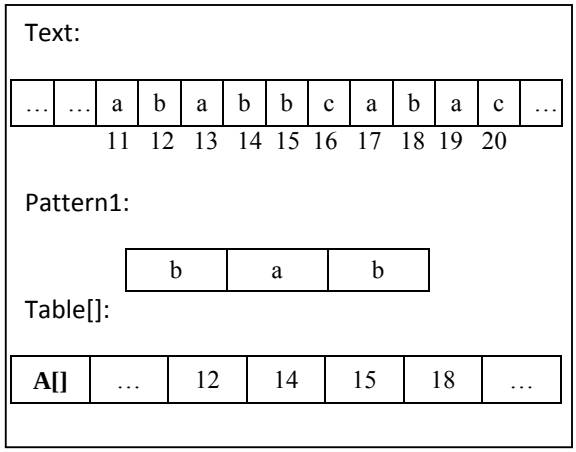

Fig. 1 The example text and pattern with the table of occurrence of the first character of the pattern in the given input text for case 1.

Possibility 1: *A[j] – A[i] = one less than the size of the pattern, i.e., n-1, where j > i at all times*

From the table, we have: *A[j] = 14 and A[i] = 12.* Hence, the condition given in possibility 1 is satisfied. This will be taken as a probability of occurrence of the pattern in the given input text at the current location and then execute the algorithm, *Match(),* which finds whether the pattern exists in that place or not. In this example,

here, we find the pattern. Index variables, *i and j,* are incremented to search for the next possibility.

Possibility 2: *A[j] – A[i] < one less than the size of the pattern, i.e., n-1, where j > i at all times*

From the table, we have: *A[j] = 15 and A[i] = 14* as the index variables, *i and j, were incremented* in the possibility 1. Hence, the condition given in possibility 2 is satisfied. This specifies that there is no possibility of having an occurrence of the pattern in the given text at the current location. Hence, the index, *j,* is incremented to search for the next possibility.

Possibility 3: *A[j] – A[i] > one less than the size of the pattern, i.e., n-1, where j > i at all times*

From the table, we have: *A[j] = 18 and A[i] = 14* as the index variable, *j,* was incremented in the possibility 2. Hence, the condition given in possibility 3 is satisfied. This specifies that there is no possibility of having an occurrence of the pattern in the given input text at the current location. Hence, the index, *i,* is incremented to search for the next possibility.

Case 2: If the first and the last characters of the Patterns are dissimilar

Let us assume that the example text and patterns for the above case are as given in Fig 2.

The table of occurrences of the first and last characters of the pattern in the given input text will be calculated using the *Different* algorithm as in the methodology and was obtained as above. After the pre-processing phase a search phase begins, where we use *Search* algorithm as given in the methodology, to find the probability of a pattern match. Here, we get three possibilities:

Possibility 1: *B[j] – A[i] > one less than the size of the pattern, i.e., n-1, where j ≥ i at all times*

From the table, we have: *B[j] = 16 and A[i] = 12.* Hence, the condition given in possibility 1 is satisfied. This specifies that there is no possibility of having an occurrence of the pattern in the given input text at the current location. Hence, the index, *i,* is incremented to search for the next possibility.

Possibility 2: *B[j] – A[i] < one less than the size of the pattern, i.e., n-1, where j ≥ i at all times*

From the table, we have: *B[j] = 16 and A[i] = 15,* as the index variable, *i,* was incremented two times in the possibility 1. Hence, the condition given in possibility 2 is satisfied. This specifies that there is no possibility of having an occurrence of the pattern in the given input text at the current location. Hence, the index, *j,* is incremented to search for the next possibility.

Possibility 3: *B[j] – A[i] = one less than the size of the pattern, i.e., n-1, where j ≥ i at all times*

From the table, we have: *B[j] = 20 and A[i] = 18* as the index variable, *i and j,* were incremented in the possibilities 1 and 2. Hence, the condition given in possibility 3 is satisfied. This will be taken as a probability of occurrence of a pattern in the given input text at the current location and then execute the algorithm, *Match(),* which finds whether the pattern exists in that place or not. In this

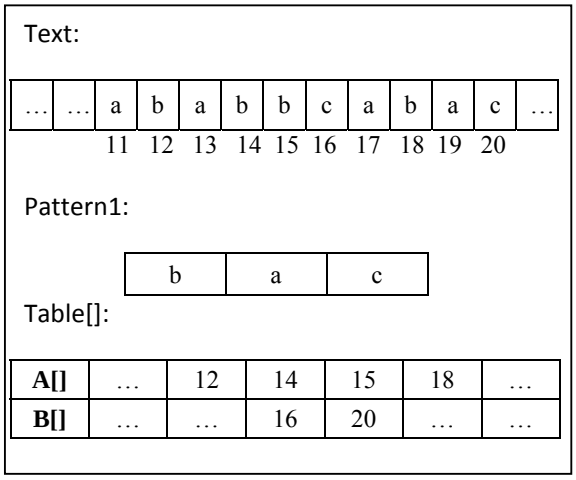

Fig. 2 The example text and pattern with the table of occurrence of the first character of the pattern in the given input text for case 2.

example, here, we find the pattern. Index variables, *i and j,* are incremented to search for the next possibility.

IV. RESULTS AND ANALYSIS

We have implemented and tested the DP algorithm and compared its performance with the BM algorithm and Quick Search Algorithm.

A. Tests with Repeated Characters

The input text and the pattern are given with the same character, or with the repeated set of characters. It provides the worst case situation for the pattern matching algorithm. The text was taken as,

"AAAAAAAAAAAAAAAAAAAAAAAA",

of size 24 characters and the patterns were taken as, "A", "AA", "AAA", "AAAA" … "AAAAAAAAAAAAAAAAAAAAAAA", with the pattern size, 1, 2 … 24, respectively. The results were compared with the BM and Quick Search algorithm and were shown in Fig. 3.

In this example, the DP algorithm has given the best performance than BM and Quick Search algorithms, if the pattern size is less than half of the text size. The performance of DP algorithm has been improved with respect to BM algorithm and Quick Search algorithm with 1%~50%, if the pattern size is less than half of the text size.

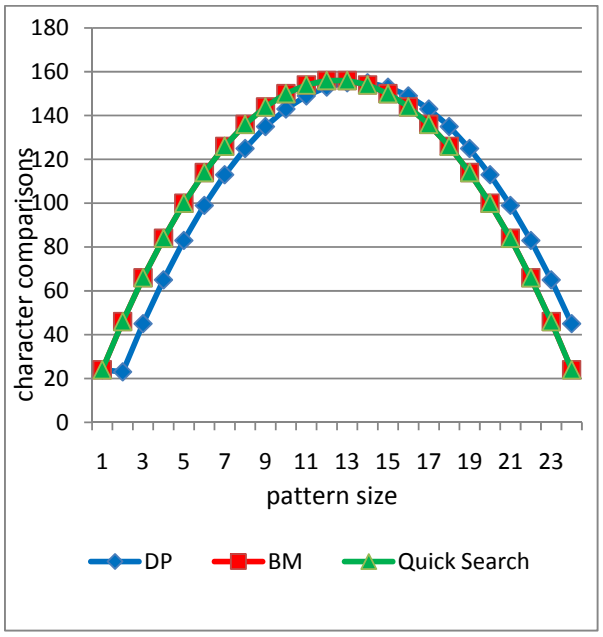

Fig. 3 The Performance of DP algorithm with Repeated characters

B. Tests with an English Text and a Random Pattern:

Let us assume, the given input text as,

"patternmatchingisoneofthebasicandmostimportantissuesintheresearchareasofcomputersciencethemeaningofthepatternmatchingisthatfindingtheoccurrencesofagivenpatterninthegiventext",

and the random patterns as "a", "of", "and", "most", "given", "issues", "pattern", "matching", of sizes, 1, 2 … 8, respectively. The results were compared with the BM and Quick Search algorithm and were shown in Fig. 4.

The performance of DP algorithm has been improved with respect to BM algorithm with 33%~91% and Quick Search algorithm with 37%~85%, varying with the pattern size from 1 to 8 as given in the example above. The time complexity of the DP algorithm is directly proportional to the number of occurrences of the first and the last characters of the pattern in the given input text obtained from the preprocessing phase.

C. Tests with English Text and Random Pattern (Unsuccessful Search):

In this case, we have taken a set of patterns as an example, which leads to an unsuccessful search. Let us assume the given input text as,

"patternmatchingisoneofthebasicandmostimportantissuesintheresearchareasofcomputersciencethemeaningofthepatternmatchingisthatfindingtheoccurrencesofagivenpatterninthegiventext",

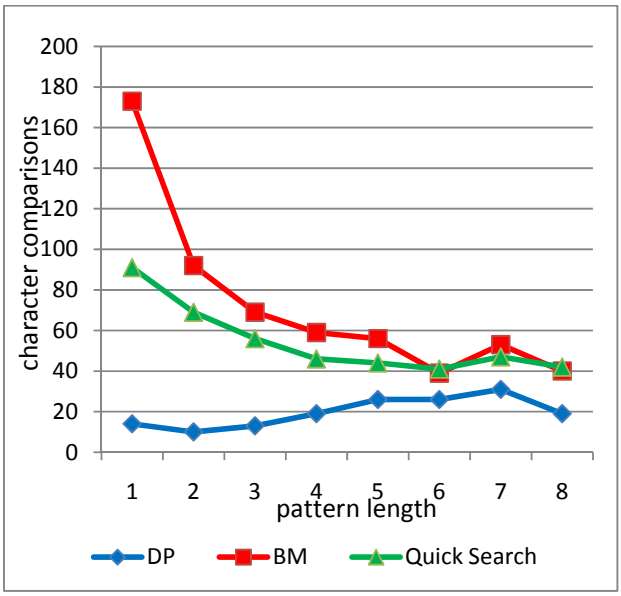

Fig. 4 The performance of DP algorithm with an English Text and Random Pattern

and the random patterns as "z", "lo", "yet", "cute", "given", "hellow", "fantasy", "kindness", of sizes, 1, 2 … 8, respectively. The results were compared with the BM and Quick Search algorithm and were shown in Fig. 5.

Here, irrespective of the pattern and the text sizes, the number of character comparisons is one. For example, if either the first or last characters of the pattern are not present in the given input text, then certainly, there is no possibility of having an occurrence of the pattern in the given input text.

In case of unsuccessful search, irrespective of the sizes of the pattern and the text, the performance of DP algorithm has been improved with respect to BM algorithm and Quick Search algorithms by 100%. Because, if either the first and/or the last character of the pattern were not present in the given input text means, there is no possibility of having a pattern in the given input text. Hence, in case of unsuccessful search (if either the first and/or the last character of the pattern in the given input text was not present), by checking the table of occurrences, we can execute the algorithm in *O(1)* time complexity.

The time complexity of DP algorithm is directly proportional to the total number of occurrences of the first and the last characters of the pattern in the given input text.

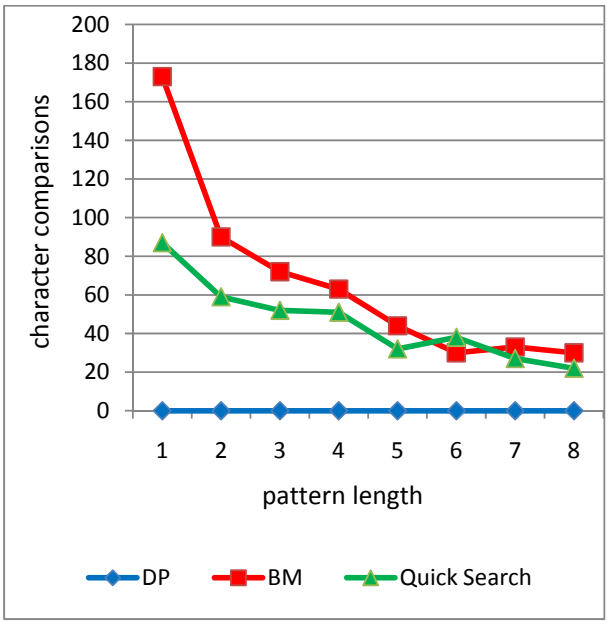

Fig. 5 The performance of DP algorithm with an English Text and Random Pattern (Unsuccessful Search)

V. CONCLUSION

We presented a novel pattern matching algorithm (DP algorithm) with a simple logic which is very easy to implement. We evaluated its performance with different texts and various set of patterns. The results were proved that the performance of the DP algorithm is greater than BM algorithm with 33%~91% and Quick Search algorithm with 37%~85%, in most of the cases. In case of unsuccessful search, the DP algorithm has one comparison with irrespective of the size of the text and pattern, provided if either the first or the last character was not present in the given input text. In this case, the performance of the DP algorithm has been improved by 100%. The algorithm also doesn't require any pre-processing phase, if the search is on the same given input text and with different patterns, provided the first and the last characters are same as in the case of first pattern. Because, the same table of occurrences can be used for the purpose. The time complexity of the preprocessing phase of the DP algorithm is *O(m),* which is less than the BM and Quick Search algorithms. The time complexity of the search phase of the DP algorithm is directly proportional to the total number of occurrences of the first and the last characters of the pattern in the given input text rather than the sizes of the pattern and/or given input text.

REFERENCES

[1] R. S. Boyer and J. S. Moore: *"A fast String Searching algorithm"*, Communications of the ACM, vol 20, no. 10, pp.762-772, 1977.
[2] C. Charras, and T. Lecroq. *"Exact string matching algorithms"*, http://www.igm.univ-mlv.fr/~lecroq/string/, 1997
[3] Snort, http://www.snort.org
[4] V. Paxson, Bro: *"A System for Detecting Network Intruders in Real-Time"*, Proc. Of the 7th USENIX Security Symposium, 1998.
[5] Stephen Gossin, et al: *"Pattern Matching in Snort"*, http://www.sporksoft.com/~njones/notes/CSE202/project.pdf, 2002.
[6] M. Crochemore, C. Hancart: *"Pattern Matching in Algorithms and Theory of Computation Handbook"*, CRC Press Inc. Bocaaton, FL. 1999.
[7] R. N. Horspool: *"Practical Fast Searching in string. Software – Practice & Experience"*, 10(6):501-506, 1980.
[8] M. Fisk and G. Varghese: *"An analysis of fast string matching applied to content-based forwarding and intrusion detection"*, Technical Report CS2001-0670 (updated version), 2002.
[9] S. Antonatos, K. G. Anagnostakis, and E. P. Markatos: *"Generating realistic workloads for network intrusion detection systems"*, In ACM workloads on software and performance, 2004.
[10] M. Fisk, and G. Varghese, *"Fast content-based packet handling for intrusion detection"*, UCSD Technical Report CS2001-0670, May 2001
[11] *"Fundamentals of Computer Algorithms"*, Ellis Horowitz, Sartaz Sahni, and Sanguthevar Rajasekaran, Galgotia Publications.
[12] Beate Commentz, Walter: A string matching algorithm fast on average. Proc. 6th International Colloquium on Automata, Languages and Programming, Vol 71 of Lecture Notes in Computer Science, pp 118-132, 1979.
[13] Knuth, D. and J. Morris, V. Pratt, *"Fast pattern matching in strings"*, SIAM J. Computing 6(1977) 323-350.

P P Bharath Bhushan, received MCA degree from S. V. University, Tirupati, India, in 2000. He has been received the M. Tech. degree in Computer Science and Engineering from Sam Higginbottom Institute of Agriculture, Technology and Science – Deemed University, India in 2004. His main area of interests in research is network security and data mining.

Pendlimarri Devaki, received MCA degree from S. V. University, Tirupati, India, in 2000. She is pursuing project of M. Tech. degree final semester in Computer Science and Engineering in Jawaharlal Nehru Technological University, Kakinada, India. Her main area of interests in research is intrusion detection, network security and data mining.