ISSN:2229-6093

Ravendra Singh et al, Int. J. Comp. Tech. Appl., Vol 2 (6),1877-1883

A FAST STRING MATCHING ALGORITHM

[1]H N Verma, [2]Ravendra Singh

[1]Department of CSE, Sachdeva Institute of Technology, Mathura, India, hnverma@rediffmail.com

[2]M.Tech(CSE-0104cs09mt16) RKDF IST Bhopal, India, ravendra85@gmail.com

ABSTRACT

The pattern matching is a well known and important task of the pattern discovery process in today's world for finding the nucleotide or amino acid sequence patterns in protein sequence databases. Although pattern matching is commonly used in computer science, its applications cover a wide range, including in editors, information retrieval. In this paper we propose a new pattern matching algorithm that has an improved performance compare to the well known algorithms in the literature so far. Our proposed algorithm has been evolved after the comparatively study of the well known algorithms like Boyer Moore , Horspool and Raita. When we are talking about the overall performance of the proposed algorithm it has been improved using the shift provided by the Horspool search bad-character and by defining a fixed order of comparison. The proposed algorithm has been compared with other well known algorithm.

1. INTRODUCTION

Pattern matching problem has attract a lot of interest throughout the history of computer science, pattern matching has been used in various computer application for several decades these algorithm are applied in most of the Operating Systems, Editors, Search Engines on the internet, retrieval of information and searching Protein or DNA sequence pattern in genome and protein sequence databases.

Theoretical studies of different algorithm suggest various possible means by which those algorithms are likely to perform but in some cases they fail to predict the actual performance, here we demonstrate that better methods can be devised from theoretical analysis by extensive experimentation and modification of the existing algorithm.

Pattern matching can be defined as the text is an array T[1…n] of length n and that the pattern is an array p[1…m] of length m<=n. We further assume that the element of P and T are character drawn from a finite alphabet ∑. For example, we may have ∑ = {0, 1}, ∑= {a, b ,……z} etc. The character arrays P and T are often called strings of characters. All pattern matching algorithm scan the text with the help of a window which is equal to the length of the pattern.

The first process is to align the left ends of the pattern with the text window and then compared the corresponding characters of the window and the pattern. This process is known as an attempt.

After a whole match or a mismatch of the pattern, the text window is shifted in the forward direction until the window is positioned at the (n-m+1) position of the text. This approach is similar to the brute-force algorithm. In brute-force algorithm, window is shifted to the right by one character after an attempt.

For a large character of text the brute-force algorithm is not efficient to perform this task. We have several well known algorithms in the literature so far and these have their own advantages and limitation based on the method they use to calculate the shift value(the number of characters the window should move forward). The algorithms vary in the order in which character comparisons are made and the distance by which the window is shifted on the text after each attempt.

The efficiency of the algorithm lies in two phases:
1. The preprocessing phases
2. The searching phase

The preprocessing phase collects the full information about the pattern and uses this information in searching phase. In searching phase, pattern is compared with the sliding window from right to left or left to right until a match or mismatched occur. The efficiency of pattern-matching algorithm is calculated on the basis of the search phase. The efficiency of the search is achieved by changing the order in which the characters are compared at each

attempt and moving the window on text, hence calculating the shift value.

2. DESCRIPTION OF WELL KNOWN ALGORITHM

Naïve string matching algorithm: The naïve algorithm finds the entire valid shifts by comparing each character of the pattern to text character, if the whole pattern is match to the text than it has a valid shift otherwise in case of mismatch it shift the pattern by one character and again compare the each character of the pattern to the text in this manner it finds entire valid shift of the pattern in the text.

Horspool Algorithm (Horspool): The Horspool[2] algorithm that has the shift value of the right most character of the sliding window. In pre processing phase the shift value are computed for all characters. In an attempt, we compare the pattern from the right to the left until the pattern match or mismatch occurs. The right most character in the window is used as the index to obtain the shift value. In case of mismatch i.e. character does not occur in the pattern, the window is shifted by the length of the pattern. Otherwise the window is shifted according to the right most character in the pattern.

Raita Algorithm: In Raita[3] algorithm, first it compare the last character of the pattern with the corresponding(i.e. nth character of the text) text character of the window, if they match, than it compare with the first character of the pattern with the left most text character of the window, if it again match then it compare the middle character of the pattern with the corresponding text character of the sliding window. And finally, it they match than again compare the characters starting from the second character to second last character of the pattern. In this case middle character of the pattern compare again while comparing from second character to second last character of the pattern.

Boyer-Moore algorithm (BM): The Boyer Moore[11] algorithm uses two different tables, one is Boyer-Moore Bad Character (bmBc) and another is Boyer-Moore Good Suffix (bmGs), a bad character table stores the shift value that are obtained on the occurrence of the character in the pattern. Another good suffix table contains the matching shift values for each character of the pattern.

3. PROPOSED ALGORITHM

This proposed algorithm is evolved after the comparatively study of the well know algorithms like Horspool and Raita. This new algorithm compares the right most character of the pattern to the text window, if it is matched then the left most character of the pattern and the sliding window is compared, if it also matched then the middle character of the pattern is compared to the corresponding position of the text window, if it is matched than it again compare the characters staring from the second last character (m-1) of the pattern to second character of the pattern to the text window[7]. In case of match or mismatch the skip of the window is achieved by the Horspool Bad character (Hbc) shift value for the character that is placed next to the window. This new algorithm first compare the last character of the pattern and second the first character of the pattern with corresponding characters of the text window, because by doing this the probability of the finding the pattern in the text window is increased over the comparison of the one by one character of the pattern in the text window[12]. This new algorithm postpones the comparison of the neighboring characters. Hence the probability of an exact match increases between the pattern and the text window[9].

3.1. PREPROCESSING PHASE

In preprocessing phase, the Horspool Bad character (Hbc) function is generated for all the characters in the alphabet set. A Horspool bad character table is create as the value of Horspool Bad character (Hbc) for a particular alphabet is defined as the right most position of that character in the pattern-1. If the character does not occur in the pattern , then the value is equal to m (length of the pattern). The skip values for each character are stored in the Horspool Bad Character (Hbc) table and these skip values are used in the searching phase. In case of mismatch, the skip value of the right most character that is stored in the Horspool Bad Character (Hbc) table is used to shift the pattern in an attempt. This process is repeated until the match or mismatch of the pattern occur in the text. When the alphabet set is large then the character occurring probability in the pattern is less and this provide a maximum skip of the window. This proposed algorithm has Horspool Bad Character (Hbc) over the Boyer-Moore Bad Character (BmBc) for the following reasons:

1. When the alphabet size is large and the pattern length is too small, then Boyer-Moore's bad-character technique is not affected.

2. Boyer-Moore algorithm has two different tables to skip the window, one is Boyer-Moore's Bad Character and another is Boyer-Moore's good suffix to calculate the skip of the window. While Horspool's Bad Character is always have the values to be ≥1 and so Horspool's algorithm works independently.

3.2 SEARCHING PHASE

The proposed algorithm search the pattern in the window, first it match the right most character of the pattern to the text window, in case of a match the left most character of the pattern to the text window is compared, if these match, then the middle character of the pattern to the text window is compared, otherwise in case of mismatch the pattern is shifted by the shift value calculated by the Horspool Bad Character (Hbc) value of the right most character of the pattern.

In case of matched middle character of the pattern, it compare the second last character of the pattern (m-1) to second character of the pattern to the text window. Middle character of the pattern is compared twice to the corresponding character of text window. This procedure is repeated until the shift does not reach to n-m+1.

3.3. C LANGUAGE IMPLEMENTATION

```
//This function computes the Horspool Bad Character, Hbc, table and store it in an array L.
//Input is Pattern and the number of elements in Σ as ALPHABET.
//m is the length of Pattern.

preprocessing( )
{
        int i;

        for(i=0; i<ALPHABETS; i++)
        L[i] = m;

        for(i=1; i<=m-1; i++)
        L[Pattern[i]]  = m-i;
}
```

```
//The function for matching.
//Input is Pattern and Text string.
//flag is a variable to check whether pattern occurs in the text or not at all.
//The valid shift values are from 0 to n-m.
/*First the last character of Pattern and Text Window is compared, in case of a match, the first character of Pattern and the Text Window is compared, in case of these two matches, the mid(if m is even, then lower median) character of the Pattern and the Text character will be compared. If it also matches then we compare second-last to second character of Pattern with corresponding text window. If all characters matches then we get a message that Pattern has been occurred. In case of mismatch at anytime we compute the shift value as shift + L[Text[shift+m]].*/

matching( )
{
        int flag=0;
        int shift = 0;

while(shift <= n-m)
{
if(Pattern[m]== Text[shift+m])
{
if(Pattern[1]==Text[shift+1])
{
if(Pattern[m/2] == Text[shift + m/2])
{
i = m-1;
k = 0;
while(i>=2&&Pattern[i]==Text[shift+i])
{
        k = k + 1;
        i = i - 1;
}
if(k==m-2)
{
printf("\n\n\t Pattern occurs with shift %d", shift);
flag=1;
}
}
}
}
shift = shift + L[Text[shift+m]];
}

        if(flag == 0)
        printf("\n\n\t Pattern does not found");
        else
        printf("\n\n\t No more pattern exists");
}
```

4. WORKING EXAMPLE

To test the proposed algorithm, the following sequence has been considered for the test run

Text
T=BBARBERBABCDEEAACDEEBBRBSRB
AREERBERBERBARBER

Pattern P = BARBER

Length of pattern *m*=6 and,

Length of text *n*=44

4.1. PREPROCESSING PHASE

The Horspool Bad Character table is obtained with a size ∑[10] that store the character and its corresponding skip value. The set of alphabets ALPHABET (∑) we have taken, includes all capital and small English letters, all digits and some special characters and punctuation marks.

∑	A	B	C	D	E	F	.. .	R	S	...
Hbc	4	2	6	6	1	6	.. .	3	6	...

4.2. SEARCHING PHASE

In first attempt:
The shift is 0.

BBARB**E**RBABCDEEAACDEEBBRBSRBAR
EERBERBERBARBER
1
BARBE**R**

Here the comparison between the last character of the pattern and the window is done, and a mismatch occur so shift value is needed to shift the pattern, the shift value is calculated from the Horspool Bad Character table Hbc[E]. Hence the window is moved by the Horspool Bad Character shift value of Hbc[E] =1.The new shift value is shift + 1 i.e. 0 + 1 = 1.

In second attempt:
The shift is 1.

B**BARBER**BABCDEEAACDEEBBRBSRBAR
EERBERBERBARBER
2 7 35 4 1
BARBER

First, last R of text window is compared with last R of pattern, then the first character B of text window is compared with B of pattern, then middle R of pattern is compared and matched with 4th character of text. Then finally pattern's second last to second characters are compared and matched with 6th character to 3rd character of text window respectively.

Pattern occurs with shift 1.
New shift value is calculated as shift + Hbc[R] i.e. 1 + 3 = 4

Total number of attempt is 2;
Total number of character comparison: 7.

In third attempt:
The shift is 4.

BBARBERBA**B**CDEEAACDEEBBRBSRBAR
EERBERBERBARBER
1
BARBE**R**

The pattern's last character is compared with tenth character of Text. The mismatched character is B, The new shift value is calculated as shift + Hbc[B] i.e.4 + 2 = 6.

In fourth attempt:
The shift is 6.

BBARBERBABC**D**EEAACDEEBBRBSRBAR
EERBERBERBARBER
1
BARBE**R**

Now pattern's last character is compared with Text's 12th character. A mismatch occurs. The mismatched character is D. The new shift value is shift + Hbc[D] i.e. 6 + 6 = 12.

In fifth attempt:
The shift value is 12.

BBARBERBABCDEEAAC**D**EEBBRBSRBAR
EERBERBERBARBER
1
BARBE**R**

In this attempt, the comparison of the last character of the pattern and the text window is carried out and a mismatch occur, the window is shifted based on the value shift + Hbc[D].The new shift value is 12 + 6 = 18.

In sixth attempt:
The shift value is 18.

BBARBERBABCDEEAACDEEBBR**B**SRBAR
EERBERBERBARBER
1
BARBE**R**

Here Pattern's last character is compared with Text's twenty-fourth character and a mismatch occurs. So the window is shifted based on the shift value shift + Hbc[B] i.e. 18 + 2 = 20.

In seventh attempt:
The shift value is 20.

BBARBERBABCDEEAACDEE**B**B**R**B**S**R**B**AR
EERBERBERBARBER
2 3 4 1
BA**R**B**E**R

In this attempt, last character of the pattern is matched with the text window, then the first character of the pattern is also matched with the text window, then the middle character* of the pattern is matched with the text window. Then, the second last character of the pattern is compared and a mismatch occurs with the text window character **S**, so the window is shifted by shift + Hbc[R] i.e. 20 + 3 = 23.

In eighth attempt:
The shift value is 23.

BBARBERBABCDEEAACDEEBBR**B**S**R**B**A**R
EERBERBERBARBER
2 3 4 1
BA**R**B**E**R

Here again, a match occurs with the right most character of the pattern and the text window, then the first character of the pattern is also matched with the text window, and the middle character of the pattern is matched with the text window. Next comparison is in between window's A and pattern's E. We get a mismatch and the mismatched character is A. So the window is shifted based on the shift value shift + Hbc[R] i.e. 23 + 3 = 26.

*If length of the Pattern is even then out of two, the lower median is selected[4].

In ninth attempt:
The shift value is 26.

BBARBERBABCDEEAACDEEBBRBSR**B**A**R**
E**ER**BERBERBARBER
2 3
54 1 **B**A**R**
B**ER**

Again, a match occurs with the right most character of the pattern and the text window, the first character of the pattern is also matched with the text window, and the middle character of the pattern is matched with the text window. Then we compare second-last character of pattern with 31st character of text. It is a match. Then we compare third-last character of pattern with 30th character of text. It is a mismatch and mismatched character is E. The new shift value is shift + Hbc[R] i.e. 26 + 3 = 29.

In tenth attempt:
The shift value is 29.

BBARBERBABCDEEAACDEEBBRBSRBAR
E**E**RBE**R**BERBARBER
2 1
B**A**RBE**R**

In this attempt, last character of the pattern is matched with the last character of text window, but the first character of the pattern is mismatched with the first character of text window, so window is shifted by shift + Hbc[R] i.e. 29 + 3 = 32.

In eleventh attempt:
The shift value is 32.

BBARBERBABCDEEAACDEEBBRBSRBAR
EER**B**E**R**B**ER**BARBER
2 7 3 54 1
BA**R**B**ER**

Here again, a match occurs with the right most character of the pattern and the text window, the first character of the pattern is also matched with the text window, and the middle character of the pattern is matched with the text window. Then we compare pattern's 5th, 4th and 3rd characters with text's 37th, 36th and 35th characters respectively and these all are matched. Next comparison is in between pattern's 2nd character and text's 34th character, which is a mismatch and the mismatched character is E. So the window is shifted based on the shift value shift + Hbc[R] i.e. 32 + 3 = 35.

In twelfth attempt:
The shift value is 35.

BBARBERBABCDEEAACDEEBBRBSRBAR
EERBER**BER**B**A**R**B**ER
2 3 4 1
BA**R**B**E**R

Again, a match occurs with the right most character of the pattern and the right most character of the text window, the first character of the pattern is also matched with the first character of the text window, and the middle character of the pattern is matched with the middle character of the text window. Then we compare second-last character of the pattern with the 40th character of the text and get a mismatch. The mismatched character is A. So the window is shifted based on the shift value shift + Hbc[R] i.e. 35 + 3 = 38.

In thirteenth attempt:
The shift value is 38.

BBARBERBABCDEEAACDEEBBRBSRBAR
EERBERBER**BARBER**
2 7 3 54 1
BARBER

Here again, a match occurs with the right most character of the pattern and the right most character of the text window, the first character of the pattern is also matched with the first character of the text window, and the middle character of the pattern is matched with the middle character of the text window. Then second-last to second character of pattern is compared and matched with the second-last to second character of text window, thus the whole pattern is matched with the text window.

Pattern occurs with shift =38.

Now the window is shifted based on the shift value shift + Hbc[R] i.e. 38 + 3 = 41, which is more than *n-m*. We get a message-

No more pattern exists.

5. ANALYSIS

First for loop of preprocessing function iterates Θ(|Σ|) times and the second for loop iterates Θ (m) times. Since, generally |Σ| > m, the complexity of preprocessing is Θ(|Σ|).

In matching function the outer while loop iterates O(n-m+1) times. Experiments show that the exact number of iterations is far less than n-m+1. The inner while loop iterates O(m) times. Thus the total complexity of the matching phase is O(m(n-m+1)).

6. CONCLUSION

We have proposed a new algorithm for string matching, by defining a new order of comparison between given Pattern and the text Window[1]. We have explained our algorithm by an example. The example strings are taken such that in which cases are covered. Our algorithm is checked and tested on many random strings. The complexity analysis is also given. Although the given algorithm is not much better than other existing algorithms in complexity, but practically its running time is faster than other existing algorithms. Hence the proposed algorithm is efficient and faster as it can be observed from the given chart and the graph.

Pattern Length	BM	Raita	Horspool	VS (our algorithm)
2	238	207	214	119.435620
4	189	174	175	95.5909075
6	175	165	166	82.4739625
8	166	159	157	75.5494650
10	163	156	154	73.9011125
12	157	153	151	71.0164925
14	154	151	150	70.6044025
16	153	151	149	65.2472575
18	152	151	147	64.8351675
20	151	148	147	62.7747250

Table1: Comparison of our algorithm(VS) with other well known algorithms[6]

The graph on next page clearly shows the difference of our algorithm with other known algorithms. Our algorithm is applicable in many areas like Biology[5], Computer science, Medical etc. It also works fine with multiple pattern matching[8].

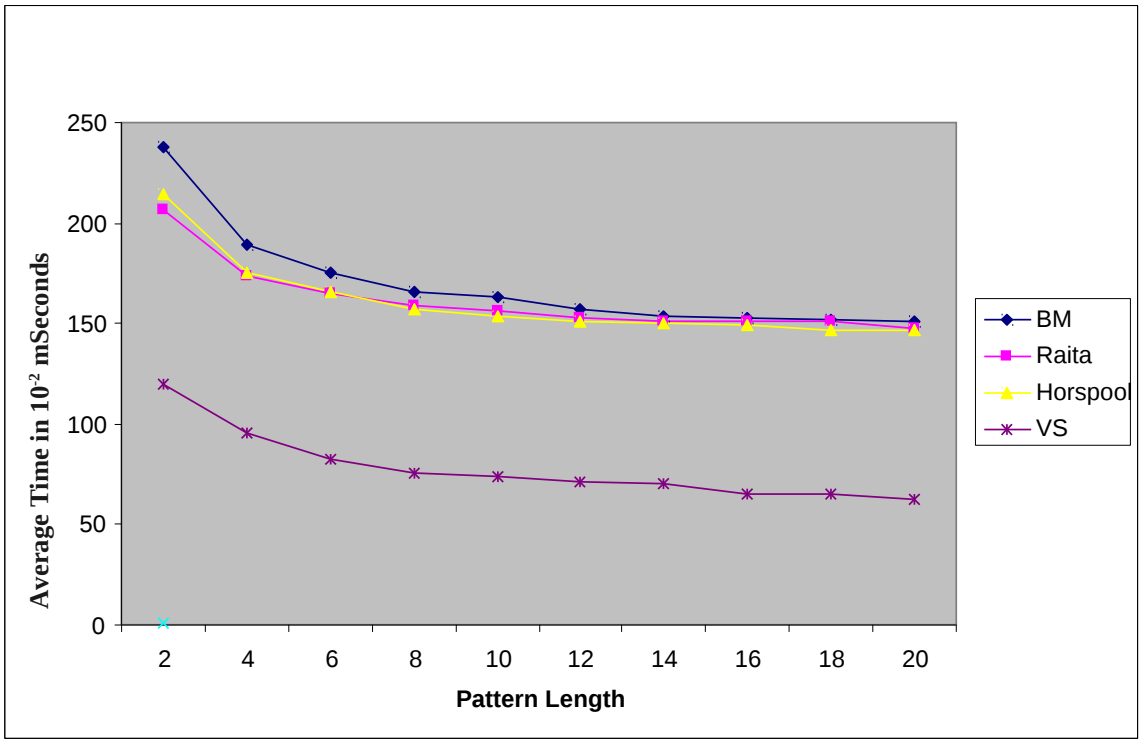

Figure 1: Comparison Chart with other well known algorithms

7. REFERENCES

1. R. Manivannan and S. K. Srivasta, "Semi automatic method for string matching", Information Technology Journal, 2011, 10(1), 195-200.
2. Horspool R. N., "Practical fast searching in strings". Software-Practice Experience 1980, 10(6), 501-506.
3. Raita T., "Tuning the Boyer-Moore-Horspool string-searching algorithm". Software-Practice Experience 1992, 22(10), 879-884.
4. Tim Bell, Matt Powell, Amar Mukherjee and Don Adjeroh, "Searching BWT compressed text with the Boyer-Moore algorithm and binary search". University of Central Florida, USA, 2001, pp. 1-10.
5. Rahul Thathoo, Ashish Virmani, S. Sai Lakshmi, N. Balakrishnan and K. Sekar1, "TVSBS: A fast exact pattern matching algorithm for biological sequences". India, 2006, pp. 47-53.
6. S. S. Sheik, Sumit K. Aggarwal, Anindya Poddar, N. Balakrishnan and K. Sekar, "A fast pattern matching algorithm", J. Chem. Inf. Comput. Sci. 2004, 44, 1251-1256.
7. Olivier Danvy and Henning Korsholm Rohde, "Obtaining the Boyer-Moore string-matching algorithm by partial evaluation". Department of Computer Science University of Aarhus, 2005, pp. 1-9.
8. Castelo AT, Martins W, Gao GR, "TROLL--tandem repeat occurrence locator". *Bioinformatics* 2002, 18(4):634-636.
9. Yoginder S Dandass, Shane C Burgess, Mark Lawrence and Susan M Bridges, "Accelerating string set matching in FPGA" Hardware for Bioinformatics Research, *BMC Bioinformatics* 2008, 9:197.
10. Mansi R. H., J. Q. Alnihoud, 2010, "An efficient ASCII-based algorithm for single pattern matching", Information Technology Journal, 2010, 9: 453-459.
11. Boyer R. S., Moore J. S., "A fast string searching algorithm". Commun. ACM 1977, 20, 762-772.
12. Domenico Cantone and Simone Faro, "Forward-fast-search: Another fast variant of the Boyer-Moore string matching algorithm". Dipartimento di Matematica e informatica, Universita di Catania, Italy, 2003, pp. 10-24.