A Novel String Matching Algorithm and Comparison with KMP Algorithm

Garima Pandey
Student (UG)
Comp. Sci. & Engg. Dept.
Women Institute of Technology,
UTU, Dehradun

Mamta Martolia
Assistant Professor
Comp. Sci. & Engg. Dept.
Women Institute of Technology,
UTU, Dehradun

Nitin Arora
Assistant Professor(SS)
School of Comp. Sci. & Engg.
UPES, Dehradun

ABSTRACT
In today's world, we need fast algorithm with minimum errors for solving the problems. Pattern matching method is a real time problem. There exist different types of data in web application problems, for example, text files, image files, audio files and video files searching. For searching different types of data search engine is required and every search algorithm are used by every search engine for handling different types of data. This paper provides a modified version of KMP algorithm for text matching. This algorithm is implemented in C language and has been checked with arbitrary input arrangement of length 10,100,1000,5000,10000. The results reflect that the performance of modified KMP algorithms is better than that of KMP algorithm.

Keywords
String Matching; KMP; Algorithms; Data Structures.

1. INTRODUCTION
A string searching algorithm works on alignment of the arrangement with the start of the text and retains on shifting the pattern advancing until a match or the finish of the text is touched [1]. All String matching algorithms are used for annoying to find one or some or all existences of a pattern string in a given text string. String matching algorithms can be used in many areas. Some of the application of string matching algorithms are, they can support to increase the awareness of a text-editor software, Other Claims in IT includes web based search systems, to filters the spam, in natural language processing, computation biology, to Feature detection in digital image processing, and many more. There is different algorithm to provide results that allow to crack the pattern matching problem. These are:

1.1 Naive String Matching Algorithm
Naïve String Matching algorithms are easy to discover, often easy to prove correct. Despite their inefficiency, naïve algorithms are often the stepping stone to more efficient, perhaps even asymptotically optimal algorithms.

1.2 KMP String Matching Algorithm
Knuth, Morris and Pratt invented the procedure that uses preprocessing of the pattern to obtain a better result.

1.3 Boyer-Moore String Matching Algorithm
Other algorithm that uses preprocessing of the pattern was invented by Boyer and Moore [2], it is thus well appropriate for applications in which the arrangement is plentiful smaller than that of text. The main feature of this procedure is to match on the right end of the arrangement rather than the left end.

1.4 Rabin Karp Algorithm
Hashing is used in this algorithm to discover any one of a set of arrangement strings in a given text string [5-6]

2. BACKGROUND
There are many methods available that permit to solve the sequence matching problem. Naive algorithm, the easiest one, which attempts to match the arrangement to each string of the same size in the text. From the 1970s, several others algorithms, additional refined and additional operational, have been invented [3-4].In 1975, Knuth, Pratt and Morris invented the first algorithm that preprocesses the pattern arrangement to obtain improved performance. In 1977, another algorithm that preprocesses the pattern arrangement was developed: Boyer-Moore Algorithm [2], its main feature is that it tries to establish the correspondence of the substring with the sequence in the converse direction. In 1987, Rabin and Karp suggest an algorithm that is centered on a totally dissimilar technique: Rabin-Karp Algorithm [5-6], which calculates a hash function for the pattern and then look for a match by using the same hash function for each possible substring of the same size length in the text.

3. STRING MATCHING ALGORITHMS
3.1 Naïve-String-Matching Algorithm
The naive method simply check all the likely arrangements of pattern $p[1 \ldots . m]$ relative to text $[1 \ldots . . n]$. Unambiguously, it tried shift $s = 0,1,2 \ldots . , n - m,$ sequentially and for each shift, S. Compare $T[S + 1 \ldots \ldots S + m]$ to $P[1 \ldots . . m]$. If P take place with shift S in T, then we say S a valid shift; else, we say S an invalid shift. Complexity of Naïve-String-Match is $O((n - m + 1)m)$WORST CASE COMPLEXITY)

$\boldsymbol{Naive\_String\_Matching(T, P)}$
1. $n = length[T]$
2. $m = length[P]$
3. $for\ S = 0\ to\ n - m$
4. $do\ if\ substring - AT(T, P, S)$
5. $then\ output(S)$

$\boldsymbol{Substring - AT(T, P, S)}$
1. $for\ i = 1\ to\ length\ [P]$
2. $do\ if\ T[S + 1]! = P[i]$
3. $then\ return\ false$
4. $return\ true$

3.2 KMP String Matching Algorithm
KMP string matching algorithm is similar to the naive string matching algorithm at high level. It reflects shifts in order

from 1 to n-m, and concludes if the arrangement matches at that shift. The modification is that the KMP string matching algorithm uses facts assembled from fractional matches of the pattern and text to avoid over shifts that are certain not to effect in a match.

KMP − MATCHER(T, P)
1 N −> length[T]
2 M −> length[P]
3 Pi −> COMPUTE − PREFIX − FUNCTION(P)
4 Q −> 0
5 for (i = 1 to n)
6 do while (q > 0 and P[q + 1] = T[i])
7 do q = pi[q]
8 if (P[q + 1] = T[i])
9 then q = q + 1
10 if (q == m)
11 then print "Pattern occurs with shift" i − m

COMPUTE − PREFIX − FUNCTION (P)
1 m −> length[P]
2 set Pi[1] = 0
3 k = 0
4 for (q = 2 to m)
5 do while (k > 0 and P[k + 1]! = P[q])
6 do k = pi[k]
7 if (P[k + 1] = P[q])
8 then k = k + 1
9 pi[q] = k
10 return pi

Time complexity: O(n + k)

Here O(n) and O(k) are the complexities of the two portions of the algorithm

3.3 Boyer-Moore Algorithm for String Matching

The procedure preprocesses the given string being examined for (the pattern), but not the string being examined in (the text). It gives better results in less time when the alphabet is reasonably sized and the pattern is reasonably lengthy.

The main characteristics of this procedure are to match on the end of the pattern rather than the starting, and to avoid along the text in jumps of numerous letters rather than examining all single letter available in the text.

3.3.1 Preprocessing Stage

For the given P, Compute L'(i) and l'(i) for each position I of P, and calculate R(x) for each letter x belongs to sigma.

3.3.2 Search Stage

The search stage is as follows:
1. K: = n;
2. While k <= m do
3. Begin
4. i: = n;
5. h: = k;
6.while i > 0 and P(i) = T(h)do
7. begin
8. i: = i − 1;
9. h: = h − 1;
10. end;
11. if i = 0 then
12. begin
13.report an occurrence of P in Tending at position k.
14. k: = k + n − l'(2);
15. end
16. else
17. shift P(increase k)by the maximum amount determined by the(extended)bad character rule and the good suffix rule.
18. End;

3.4 Rabin Karp String Matching Algorith

Rabin Karp String Matching
1 m = length[P]
2 h = d^m − 1 mod q
3 p = 0
4 to = 0
5 for i = 1 to m
6 do p(dp + P[i])mod q
7 do p(dto + P[i])mod q
8 for s = 0 to n − m
9 do if p == ts
10 then if P[1 ... m] = T[s + 1 ... ... s + m]
11 then "Pattern occur with shift" s
12 if s < n − m
13 then ts + 1(d(ts − T[s + 1]h) + T[s + m + 1])mod q

The time complexity of the Rabin Karp procedure is O(n + m) in best case and average case, but the worst-case time complexity of Rabin Karp procedure is O(nm)

4. NEW MODIFIED KMP ALGORITHM

KMPsearch(P, T)
1. m −> Pattern
2. n −> Text
3. int c[]
4. j = 0
5. COMPUTELPSARRAY(PAT, M, C)
6. i = 0
7. while(i < n)
8. if(pat[j] == txt[i])
9. j + +
10. i + +
11. if(j == m)
12. printf("found pattern at index %d", i − j)
13. j = c[j − 1]
14. else if(i < n&& pat[j]! = txt[i])
15. if(j! = 0)
16. j = c[j − 1];
17. else
18. i = i + 1;

COMPUTELPSARRAY (PAT, M, C)
1. m = p. length
2. *let c*[1 ... *m*] *be a new array*
3. $j = 0, i = j + 1, c[0] = 0$
4. *while*$(i < m - 1)$

```
{
while(p[i]! = p[j])
{
c[i] = 0;
i + +;
}
if(p[i] == p[j])
{
c[i] = j + 1;
```

```
 i + +;
 j + +;
 }
 }
5.    while(i == m − 1)
       {
       Int k;
       while(p[i]! = p[j])
       {
       k = c[i − 1];
       j = c[k];
       }
       if(p[i] == p[j])
       {
       j = j + 1;
       c[i] = j;
       }
       }
6. return c;
```

5. COMPARISON AND RESULTS

Simple KMP

```
Pattern: abcdabca
Computelpsarray(pat, m, c)
1. intlen = 0
C[0] = 0, i = 1;
while(i < m) //m is the length of pattern
i. e. while(1 < 8) − − − true
{
If(pat[i] == pat[len]
i. e. if(pat[1] == pat[0])— false
else {
if(len! = 0)— false
else
{lps[i] = 0; i. e. lps[1] = 0;
I + +; i. e. i = 2
}
}
Continuing this till m = 8
While(i < m) //m is the length of pattern
i. e. while(7 < 8) − − − true
{
If(pat[i] == pat[len]
i. e. if(pat[7] == pat[0])— true
{
len + +;
i. e. len = 1
lps[i] = len;
lps[7] = 1;
i + +;
i = 8
}
}
```

Modified KMP

```
Pattern: abcdabca
computelpsarray(pat, m, c)
1. j = 0, C[0] = 0, i = j + 1;
While(i < m − 1) //m is the length of pattern
i. e. while(1 < 7) − − − true
{while(pat[i]! = pat[j])
i. e. if(pat[1]! = pat[0])— true
c[i] = 0 and i + +;
i. e. i = 2, j = 0
}
Continuing this till m = 7
While(i == m − 1) //here m is length of the pattern
i. e. while(7 == 7) − − − true
{int k;
while(pat[i]! = pat[j])
i. e. if(pat[7]! = pat[3])— true
{
K = c[i − 1];// i. e. k = c[7 − 1], k = c[6], k = 3;
J = c[k]
J = c[3] = 0;
while(pat[7]! = pat[0])
if(pat[i] == pat[j])
i. e. if(pat[7] == pat[0])— true
{
J = j + 1;
i. e. j = 1;
c[i] = j;
i. e. c[7] = 1;
}
```

In the modified compute prefix function, we have used two while loops rather than using the nested if else that is decreasing the time complexity of the algorithm and making our algorithm more efficient.

6. CONCLUSION

In digital atmosphere searching the exact contented in least time is utmost essential. String matching algorithms play a vivacious role for this. Many persons are functioning on software and hardware levels to make arrangements searching quicker. By approximate best algorithms in various algorithms in various claims is determined. The recommended algorithms i.e. the modified compute prefix function give the compact complexity and also compact calculation time. The procedure allotted to various requests may not be the best optimum algorithm but better than the all-purpose algorithms. It has been well-known that many applications use Boyer Moore, KMP algorithm for their operational functionality and other uses the basics of these algorithms for their functionalities as the KMP algorithm has less time complexity and Boyer Moore algorithm has preprocessing time complexity less.

7. REFERENCES

[1] Koloud Al-Khamaiseh and ShadiALShagarin, "*A Survey of String Matching Algorithms*" in Int. Journal of Engineering Research and Applications, IJERA, ISSN: 2248-9622, Vol. 4, Issue 7 (Version 2), July 2014, pp.144-156

[2] Robert S. Boyer and J. Strother Moore. "*A fast string searching algorithm*". Communications of the ACM, Volume 20, Number 10, pages 762{772, October 1977.

[3] Syeda Shabnam Hasan, Fareal Ahmed and Rosina Surovi Khan, "*Approximate String Matching Algorithms: A Brief Survey and Comparison*" in International Journal of Computer Applications, Volume 120 – No.8, June 2015.

[4] Iftikhar Hussain, SaminaKausar, Liaqat Hussain and Muhammad Asif Khan, "*Improved Approach for Exact Pattern Matching*" in International Journal of Computer Science Issues, Vol. 10, Issue 3, No 1, May 2013.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*, Third edition. The MIT Press, 2009.

[6] Robert Sedgewick and Kevin Wayne. Algorithms, Fourth edition.Addison-Wesley,2011