International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 4 Issue 11, November-2015

Partial String Matching Algorithm

Shibdas Bhattacharya
Dept. of Computer Science & Technology
Technique Polytechnic Institute
Hooghly, West Bengal, India

Aratrika Saha
Dept. of Computer Science & Engineering [3rd year]
Kalyani Government Engineering College
Nadia , West Bengal, India

***Abstract*— The purpose of this research is to find an effective string matching algorithm whose complexity is reduced by a good amount compared to the common string matching algorithms that already exsits . We tried doing this by partial matching , i.e. we are initially matching the first and last character of the given substring with the original string and then if that matches then only we are proceeding to check the intermediate characters .**

Keywords—String, String-Matching, Partial Matching, Complexity .

I. INTRODUCTION

We are first going to discuss about the basic concepts of string and string matching algorithms .

A. Definition of String Matching

In computer programming a string is traditionally a sequence of characters, either as a literal constant or some kind of variable . The latter may allow its elements to be mutated and the length changed , or it may be fixed(after creation) . A string is generally understood as a data type and is often implemented as an array of bytes(or words) that stores a sequence of elements, typically characters, using some character encoding .

B. Definition of String Matching Algoritms

(1) Text-editing programs frequently need to find all occurences of a pattern in the text . Typically , the text is a document being edited , and the pattern searched for is a particular word supplied by the user. Efficient algorithms for this problem-called "String Matching" . It can greatly aid the responsiveness of the text-editing programs . Among their many other applications, string-matching algorithms search for particular patterns in DNA sequences . Internet search engines also use them to find Web pages relevant to queries .

C. Formal illustration -

(1) We formalize the string matching problem as follows.

We assume that 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 elemnts of P and t are drawn from afinite alphabet $\sum$. We say that pattern P ooccurs with shift s in text T(or, equivalently , that pattern P occurs beginning at position s+1 in text T) if 0 <= s <= n-m and T[s+1…..s+m] = P[1…m](that is , if T[s+j] = P[j] , for 1<=j<=m). If P occurs with shift s in T , then we call s a valid shift ; otherwise we call s an invalid shift . The string-matching problem is the problem of finding all valid shifts with which a given pattern P occurs in a given text T.

These are the basic concepts of string matching algorithm .Some of the common string matching algorithms that already exsits are : Naïve String matching algorithm , Rabin –Karp string search algorithm ,Finite State automaton based search , Boyer-Moore string search algorithm , Bitmap algorithm .

In our algorithm we have achieved a time complexity which is less than the time complexity of all the above mentioned algorithm .

It is : Preprocessing = $\Theta(n)$, Matching = O(mk)

Where m = Length of the pattern , n = Length of the searchable text , K = number of. Time there is a hit.

The algorithm basically first takes note of all the position in the searchable string where the first character of the pattern matches . Then after this step it starts matching the last character of the pattern with the character that is present at an index number which is the summation of pattern length and index number of the character that matched with the first character of the pattern . If this holds true then it further goes back and match rest of the pattern characters else it moves on for another processing . Thus we can see here that we are not matching each character of the text with the pattern , hereby we can say that we are implementing partial checking .

We will explain in details in the Algorithm section .

II. SURVEY

Before we go in detailed study of our algorithm we would like to give a brief overview of the already existing algorithms along with their complexity analysis and try to explain why our algorithm performs better than them .

The algorithms with their respective complexity algorithm is as follows :

ALGORITHM	PREPROCESSING TIME	MATCHING TIME
Naïve String , matching algorithm	[1],0(no preprocessing)	[1] , O((n-m+1)m)
Rabin –Karp string search algorithm	[2],Θ(m)	[2], O(n+m)
Finite State automaton based search	[3]Θ(mk)	[3]O(n)
Boyer-Moore string search algorithm	Θ(m+k)	O(n)
Bitmap algorithm	Θ(m+k)	O(mn)

Table 1 : Comparative Study of String Matching Algorithms

Where m = Length of the pattern , n = Length of the searchable text , k = particular constant .

As already mentioned above we can see that the complexity of our algorithm is much less than all of the above mentioned algorithms .

NOTE

Also we will see in the next section that if the given pattern is not present in the text at all then it will produce result only after precomputation and wont require the matching part of the algorithm . Thus the complexity then reduces to only O(n) .

III. AlGORITHM

Following assumptions are being made in this algorithm . Let there be an –

Text[n] – Searchable text

Pattern[m] – The subtext to be matched

m = Length of the pattern

n = Length of the searchable text ,

Hit[n] = Array which stores the index number of the characters in the searchable text which matches with the first character of the pattern (i.e. Pattern[1]) .

STEP 1 (Precomputation) :

```
int t = 0 ;
For(i=0 ; i<n ; i++)
{
    If(Pattern[0] == Text[i] && pattern[m-1] == Text[i+m-1] )
    {
While(t < n)
        {
            Hit[t] = i ;
            t++ ;
        }
    }
 }
if(t==0)
Pattern does not exist in the Searchable string
STEP 2 (Matching) :
For(i=0 ; i<t ; t++)
{
    For(u = m-2 ; u > 0 ; u--)
        {
            if(Text[Hit[t] + u] == Patterrn[u])
            match ;
        }
}
```

We can see that in the pre computation part we have to do maximum ‘n’number of comparisons , thus it gives a complexity of O(n) . Also in the precomputation part only we are checking with last character of the pattern , thereby determining that only which part in the searchable string/Text we need to revisit during the matching time .This herby reduces the complexity of the matching algorithm as we only revisit the pre-computed parts in the given searchable string. So it reduces to O(km) , where k is some constant .

Also in our algorithm we can keep track of the number of the comparisons that has to be made in the matching part of the algorithm. It will always be equal to the number of hit that occurs.

IV. ILLUSTRATION

Example 1 :

Searchable string/Text –

a	b	b	d	a	c	a	a	b	c
i=0	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8	i=9

Pattern –

a	b	c
i=0	i=1	i=2

Searchable string length (n) = 10

Pattern length (m) = 3

Hit[n] = [7]

From the above mentioned algorithm we can clearly say that after precomputation the Hit[n] have only one value since the 1st and the last of the pattern matched with only a single instance in the Searchable String [i.e. when Text[7] == Pattern[1] && Text[7+3-1 = 9] == Pattern[3-1 = 2]] . Now the control goes to the matching algorithm and it checks only once with [Text[8] == pattern[1]] , which is true in this case , so it generates the result “Match”

Here since In preprocessing total comparisons is 10 (i.e. O(n))

In matching total comparisons is 1 (i.e. O(1*m) = O(m))

Example 2 :

Searchable string/Text –

a	b	b	d	a	c	a	a	b	c
i=0	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8	i=9

Pattern –

x	y	z
i=0	i=1	i=2

Searchable string length (n) = 10

Pattern length (m) = 3

Hit[n] = 0(no hit)

In this example we can see that none of the characters present in the pattern match with any character of the given searchable string. Thus there will be no Hit . From there only we can say that it Does not match . So time complexity for this particular example is O(n) , cause it does not require the matching part of the algorithm , it can generate result from only the precomputation part .

Example 3 :

Searchable string/Text –

a	a	a	a	a	a	a	a	a	a
i=0	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8	i=9

Pattern –

a	b	c
i=0	i=1	i=2

Searchable string length (n) = 10

Pattern length (m) = 3

Hit[n] = 0(no hit)

Since here there will be no hit as Pattern[2] will not match with any of Text[n] so we can generate the result in the precomputation part only that the string is not present .

Hence time complexity for this problem is O(n) only .

ACKNOWLEDGMENT

We would like to thank Technique Polytechnic Institute , Hooghly , West-Bengal ,India for all their support without which this paper would not have been possible .

REFERENCES

[1] Thomas H. Cormen , Charles E. Leiseron , Ronald L.Rivest , Clifford Stein, “Introduction to Algorithms” , Thrird edition , PHI publication

[2] International journal of advanced research in Computer Scienceand Software engineering , Rabin-Karp Algorithm with hashing a string matchingtool,Vol4,Issue3,Ms Sunita,MsRitu Malik,Ms Mamta Gulia

[3] http://www.iitg.ac.in/rinkulu/algorithms/slides/str-dfa.pdf