UNIT 3 STRING MATCHING TESHNIQUES

Structure **Page No**

3.0 INTRODUCTION

String matching is an important problem in computer science in which a sub-string (also called a pattern) is searched in a larger string or a text (e.g., a sentence, a paragraph of a book) and returns the index of a starting character of a substring. When we search for a word in a text file/data base /browser, string matching algorithm is used to find the result. Some of its interesting applications are found in designing of text editors, plagiarism checking software, Spell Checkers, Search Engines, Bioinformatics, Digital Forensics and Information Retrieval Systems , searching for a pattern in DNA sequences, etc.,. In this unit we present three string matching algorithms: Brute Force or the Naïve algorithm, the Rabin-Karp algorithm and the Knuth-Morris-Pratt (KMP)algorithm. The Naïve algorithm is the simplest algorithm of all. The other two algorithms require some kinds of preprocessing.

3.1 OBJECTIVES

After going through this unit , you should be able to:

- Identify applications of string matching algorithms
- Write pseudo-codes of string matching algorithms
- Explain the working of string matching algorithms
- Calculate the running time efficiency of string matching algorithms

3.2 THE NAÏVE OR BRUTE FORCE ALGORITHM

The purpose of the string matching algorithm is to search for a location of a smaller text pattern in a paragraph or a book or any other sources.

The Naïve search algorithm compares the pattern string to the text, one character at a time .This process continues until there is mismatch of characters between the pattern

string and the text. Depending upon the requirement, the algorithm can be designed to find a single occurrence or multiple occurrences of a pattern string in the text. In the latter case, the entire text is required to be searched.

Pseudo-Code of Naïve String Matching Algorithm

do

{

if (pattern string character == text character)

compare next character of the pattern string to next character of text character;

else

shift the pattern string to the next character of the text;

while(entire pattern string matched or end of the text)

}

The following examples illustrates the concepts:

(i)Example 1(shift =0)

Text = a b c d e f g I

Pattern String = a b c

(ii) Example 2(shift =4)

Text = a b c d e f g h I

Pattern =e f g

(iii) Example 3(maximum valid shift in the text=(length of the text-length of pattern)=6)

Text = a b c d e f g h i

Pattern= g hi

Time Complexity of Naïve string matching algorithm

The algorithm finds matching of the pattern in a text using all valid shifts. Let us rewrite the algorithm:

1. n = length of a text T
2. m = length of a pattern string P
3. n-m = maximum valid shifts of a pattern in a text (refer to ex.3)
4. s – shift index
5. for (s = 0 , s<= n-m, s++)
6. if P[1..m] == T[s +1..s + m]
7. Display " Occurrence of a pattern string with shift" s

Best case- It happens if the pattern matches in the first m positions of the text. Total number of comparisons = m(size of the pattern string)
Therefore Best case time Complexity= O(*m*)
In the worst case scenario, the total number of comparisons: m(*m-n+1*)
Therefore worst case time complexity= O(*mn*)

☞ Check Your Progress-1

Q1 What is a string matching problem?

.....

.....

.....

.....

Q2 What are different types of string matching algorithms?

.....

.....

.....

.....

Q3What are the applications of string matching algorithms?

.....

.....

.....

.....

3.3 THE RABIN KARP ALGORITHM

The central idea in Rabin Karp algorithm is computation of hash function to speed up the pattern matching. The algorithm calculates hash values for (i) pattern string of m-characters (ii) m-character substring of a text . If the hash values of (i) and (ii) are equal, the algorithm will perform a brute force comparison between the pattern string and m-character text substring. Advantage of this approach is that there is only one comparison per text substring and brute force method is required only when hash values are equal.

If hash values do not match, the algorithm will pick up the next m-character text substring for calculation of its hash values.

Pseudo-code of Robin –Karp Algorithm

Input

m- length of a pattern substring

P_ hash – Hash value of a pattern string

T_ hash – Hash value of a first m character of a text substring

do

if(P_ hash == T_ hash)

brute force comparison of the pattern string and the first m-character text substring ;

else

T_ hash = hash value of the next m-character of the text substring after one character shift;

while(match of the pattern string **or** end of the text)

But how to design a hash function? In simple terms, a hash function maps a big n string to a small value. A good hash function should have properties such as: efficiently computable and should uniformly distribute the keys and do not generate spurious hits. The hash function considered here has been suggested in the Rabin Karp algorithm. It is also called **rolling hash function**

To compute the hash value , let us number the alphabet as a=1, b=2, c= 3, d=4,e=5,f= 6,g=7,h=8,i=9 and j=10 to simplify computation instead of assigning the ASCII value to each alphabet. Also consider m-character sequence as m-digit number having base 10, where 10 is the number of alphabets used in our pattern string . Now let us convert m-character text which can be written in form of a polynomial expression:

hash(m-character sequence)= P[1]* 10^{m-1} +P[2] *10^{m-2}+P[3]* $10^{m-3}$where P[1], P[2],P[3] are first, second and third equivalent digital values of 3-character pattern. If the first character is 'a' then P[1] =1. Furthermore, given hash(m- character string), we can compute the hash value of the next m-character substring skipping one character by subtracting the leftmost digit and adding new rightmost digit in constant time. Using this approach, we simply adjust the existing value but never explicitly compute the new value. This is demonstrated through an example below:

Now let us take one example to understand :

Text T = "baecddabcdef"

Pattern P= "ecd".

Step 1: Calculate the hash value of a pattern string P

Hash(P) =5*10^2 + 3*10^1 + 4* 10^0 = 534

Step 2: Compute hash value of a text substring of the size equal to P

Hash(T=bae) = $2 *10^2+ 1* 10^1+ 5* 10^0= 215$

Step 3 : Compare the hash values of P and T(a text substring). Since both are not equal, pick up the next text substring sliding over one character, compute its hash value and compare it with P. Using rolling hash function, the hash value of the next text substring can be calculated easily from the previous expression by subtracting the first digit , multiplying the second and the third digits by 10 and finally adding the new third digit as :

- Hash (aec) = $[2 *10^2+ 1* 10^1+ 5* 10^0]- 2 *10^2] *10+3*10^0$
  If you write it directly, you get the same expression:

  $1* 10^2+ 5* 10^1+ 3* 10^0= 153$

Step 3(contd..)Again there is mismatch, so we will take the third text substring "ecd", compute its hash value and compare it P

Hash (ecd)= $[1* 10^2+ 5* 10^1+ 3* 10^0]- 1* 10^2] * 10] + 4*10^0$

$= 53* 10 +4 = 534$

The hash value of both are equal. Then we compare each character of a pattern with each character of a text substring. All characters are equal. Therefore the pattern string was found.

Rabin-Karp Complexity

Best Case –O(n) where n is a length of a text. If sufficiently large base number or a large prime number is used for computing hash value , there will be no spurious hits and the hashed values would be distinct for both a pattern string and a text substring. In such a case , the searching would take O(n) time

Worst Case = O (*mn*) where m is a length of a pattern string and n is a length of a text string.. This may happen if there are spurious hits because of use a small base number/prime number in hash calculation of a pattern and a text string

☞ Check Your Progress-2

Q1 How does the Rabin Karp algorithm work?

Q2 What is the worst case time complexity of Rabin Karp algorithm?

.....

.....

.....

.....

3.4 KNUTH MORRIS PRATT ALGORITHM

This is linear time string matching algorithm. The complexity is **O(*m+n)*** where m and n are the length of a pattern string and a text string respectively. This happens because the KPS algorithm avoids frequent backtracking in the text string as it is done in the naïve algorithm. The key idea in KMP algorithm is to build a LPS(largest prefix as suffix) array to determine from which point in the pattern string to restart comparing for pattern matching in a text in case there is a mismatch of a character **without moving the text pointer backward**. In such a case first we go back one character backward from the position where mismatch occurred, read its value in the LPS array which defines the length of a prefix also as a suffix if any ,i.e., we check whether there is any occurrence of the largest prefix as a suffix in the pattern to decide how many characters in the pattern need to be skipped to start searching for the string matching hat the next stage. If there is a mismatch at the i^{th} character of a pattern string, we move to $(i-1)^{th}$ character) in the pattern string and find out the LPS array value of this character. Suppose the LPS array value of $(i-1)^{th}$ character is 2 . This number defines the length of the largest prefix which is also a suffix in a pattern string. It also indicates the first two characters in the pattern string need to be skipped for the next comparison of pattern string with a text. If the length of a pattern string is m then only (m-2) characters will be compared with a text(not from the beginning of the text but from the position where there was a mismatch).

In the following example we first do the pattern matching exercise without building the LPS array to get the idea quickly. But efficient implementation of KPS would be done through LPS array only.

Example :

Text :abcfa b c x a b f a b a b c x a b c z

Pattern = abcxa b c z

Examining the text and the pattern string we notice that there is mismatch of characters at the fourth position. In the text string it is ‘f’ where as in the pattern string it is ’x’. To decide from which position in the pattern string the search should restart , we go back and examine the substring in the pattern just before the position where there was a mismatch, i.e., we examine “a b c” substring of pattern (so far we have not built up the LPS array). Since all the characters are unique, there is no prefix also as a suffix in this substring, therefore we can not skip any character in the pattern string, the comparison will start from the beginning character of the pattern, i.e., ‘a’ with ‘f’(at this position , there was a mismatch). Again there is a mismatch, so we will move to the next character in the text, i.e., ‘a’ and start comparing with the first

character in the pattern string ,i.e., 'a'. There is a match of a character, we will compare the next character of a pattern with the next character of a text and continue till there is a mismatch. Please notice that there is a mismatch at the 7th character in the pattern, i.e., 'c' with 'f' in the text. Now we examine the substring of a pattern just before the position of the mismatch, i.e., "a b c x a b " and decide how many characters to be skipped in the pattern. We try to find out the length of prefix also as a suffix in this substring. It is "ab" which is suffix as well as prefix and the length is 2. Therefore we will skip two characters in the pattern from the beginning, which is now, 'c' and start comparing with the same position of character in the text where there was a mismatch, i.e., 'f'. It makes a sense because " ab " is existing in the text just before 'f'. It need not be compared again. Now there is a mismatch, so we go back and examine the substring in pattern to find out if there is any prefix which is also as a suffix. The substring is "ab". There is no prefix and suffix information in the substring, because both are unique characters. Therefore the comparison starts with the first character of the pattern ('a') and the next character in the text('a'). The next character is 'b' in the pattern as well as in the text. Finally we find that the pattern is found in the text .

It is time now to build a LPS array for a pattern P.

P = d e f g d e f

I j

d	e	f	g	D	e	f	d
0	1	2	3	4	5	6	7
0	0	0	0	1	2	3	1

Figure : LPS Array (last row)

In the above table , the last row builds information about prefix and suffix of the pattern. The first and the second row indicate the pattern and its index values. We consider two pointers i and j. Initially i is pointing to the first character of the pattern and j is pointing to the second character of the pattern. The first entry in LPS array(last row of the table) is zero. For the second entry, we compare i with j which are not equal. i is pointing to d and j is pointing to e. If i and j are not equal, the related entry in the LPS array will be zero. Then j will be incremented by one. Again i and j are not equal. i is pointing to a and j is pointing to 'f'. The LPS entry for this index will be zero. Similarly the next entry in the LPS array is zero again. What will happen when j is pointing to d and i is also pointing to d. In this case the LPS entry of the character pointed to by j will be equal to the current index value of i and + 1. Therefore the entry will be 1 at this column. The next two entries will be 2 and 3. Each time there is a match, i will get incremented. Now i is pointing to g and j is pointing to d. There is mismatch. In this case i will be decremented by 1. i is pointing to 'f' now. Its LPS entry value is zero. Accordingly i will shift to the beginning of the array. Both i and j are pointing to the same character. The LPS entry for the last character will be the index value of i, which is zero plus 1,i.e., 1

The question is now how to use LPS array for pattern matching? Please refer to the last row of the array. There is entry 1 at the 4th column. We will ignore the remaining entries in the LPS array after this entry. It indicates that the length of a prefix which is also a suffix is one. Therefore skip one character from the beginning in the pattern for the next comparison. Let us take one example:

Text = d e f g d x . . . .

Pattern = d e f g d e

There is a mismatch between 'x' and 'e' characters. After mismatch we go back to the previous character, i.e., 'd'. The entry for d in the array is 1 , which says that the next comparison will start from 'e' in the pattern and 'x' in the text.

Time Complexity of KMP Algorithm

The worst case time complexity of KMP is O(m+n) where O(m) is time taken to build LPS array and O(n) is the time taken to search for entire text.

☞ Check your Progress-3

Q1 What is the basic principle in KMP algorithm?

.....

.....

.....

.....

Q2. How do you build LPS array in KMP algorithm?

.....

.....

.....

.....

3.5 SUMMARY

String matching is a problem in which we look for a pattern string into a larger text and return the location where the pattern has occurred in the text. In this unit we examined three string matching algorithms: Naïve algorithm, Rabin Karp Algorithm and Knuth, Morris and Pratt algorithm. The naïve algorithm is the simplest algorithm of all. The Rabin Karp algorithm is based on computing the hash values of an m-character pattern string and an m-character text substring. If the hash values are equal then only we make comparison of the pattern string and a text substring character by character, otherwise we compute the hash value of the next text substring and compare with the pattern string unless there is an end of the text or the pattern was found at some step. The KMP is linear time algorithm. The algorithm forbids moving a text pointer backward.

3.6 SOLUTION / ANSWERS

☞ Check Your Progress-1

Q1 What is a string matching problem?

Ans- The string matching is a problem of finding occurrence(s) of a **pattern string** within a larger text. If there is a match, the algorithm returns the location of the text where the pattern has occurred.

Q2 What are different types of string matching algorithms?

1. The Naive String Matching Algorithm
2. The Rabin-Karp-Algorithm
3. Finite Automata (not discussed in this unit)
4. The Knuth-Morris-Pratt Algorithm
5. The Boyer-Moore Algorithm(not discussed in this unit)

Q3 What are the applications of string matching algorithms?

Ans. String matching algorithms have found applications in various real world problems .A few applications are spell checkers, information retrieval systems, spam filters, intrusion detection system, search engines and plagiarism detection

☞ Check Your Progress-2

Q1 How does the Rabin Karp algorithm work?

AnsThe Rabin-Karp string matching algorithm performs calculation of a hash value for the pattern string , as well as for each m-character substring(size of a pattern string) of text to be compared. If the hash values are equal, the algorithm will compare the pattern string and the m-character text substring . In this way, there is only one comparison per text substring, and character by character pattern matching is only needed when the hash values match. In case the hash values do not match, the algorithm will determine the hash value for the next m-character substring.

Q2 What is the worst case time complexity of Rabin Karp algorithm?

Ans The worst **case complexity** is O(mn) . The worst-**case complexity** occurs when spurious hits occur very frequently because of use a small base number/prime number in hash calculation of a pattern and a text string

☞ Check your Progress-3

Q1 What is the basic principle in KMP algorithm?

The Knuth–Morris–Pratt string-searching **algorithm** (or **KMP algorithm**) searches for occurrences of a pattern string within a text by employing the observation that when a mismatch occurs, the pattern itself contain sufficient information related to prefix and suffix which determine where the next match could begin.

Q2 How do you build LPS array in KMP algorithm?

Steps for Creating LPS array

Step 1 - Define a one dimensional array with the size equal to the length of the Pattern string P

Step 2 - Define two variables i& j. Let i point to the first character and j points to the second character of P

Step 3. LPS[0] = 0

Step 4 - Compare the characters at P[i] and P[j].

Step 5 - If both are matched then set **LPS**[j] = i+1 and increment both i& j values by one.