Analysis Design and Algorithms

PROJECT REPORT

Title: *Comparison of String Matching Algorithms*

Submitted By: **Rahul Kumar T** **1PI13CS114**

Sharath K P **1PI13CS141**

Guide:

Prof. N S Kumar

Visiting Faculty

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

PES INSTITUTE OF TECHNOLOGY

100 FT. RING ROAD, BANASHANKARI 3RD STAGE, HOSAKEREHALLI , BENGALURU – 560085

Abstract:

String matching is the problem of finding all occurrences of a character pattern in a text. This project provides an overview of different string matching algorithms and comparative study of these algorithms. In this project, we have evaluated several algorithms, such as Brute Force algorithm, Rabin-Karp algorithm, Boyer Moore algorithm, Knuth-Morris-Pratt algorithm, Horspool algorithm. We analyzed the core ideas of these single pattern string matching algorithms. We compared the matching efficiencies of these algorithms by searching speed, pre - processing time, matching time and the key ideas used in these algorithms. It is observed that performance of string matching algorithm is based on selection of algorithms and also on the type of text and pattern matching.

APPLICATIONS

1. Searching for indexes in website(eg: Google search)
2. Compiler (eg: syntax checking)
3. Text editor(eg: keyword searching)
4. Dictionary
5. Password matching
6. The main advantage of string matching is in the field of biotechnology such as DNA pattern matching(g,c,t,u).

Algorithms

1.Naive string matching algorithm:

It is also known as *Brute Force algorithm***.** It has no preprocessing phase, needs constant extra space. It always shifts the window by exactly one position to the right. It finds all valid shifts using a loop that checks the condition P[1....m]=T[s+1.....s+m] for each of the n-m+1 possible values of s .

ALGORITHM *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

```
//Implements brute-force string matching
//Input: An array T[0..n − 1] of n characters representing a text and
//       an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//        matching substring or −1 if the search is unsuccessful
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return −1
```

2.Rabin Karp String Matching Algorithm:

Rabin Karp algorithm is used to find a numeric pattern P from a given text T. It mainly uses hashing method. It firstly computes the hash value of pattern and the specified window of the text using a hashing function and then divides the pattern with a predefined prime number q to calculate the remainder of pattern P. Then it takes the first m characters from text T at first shift s to compute remainder of m characters from text T. If the remainder of the pattern P and remainder of the text T are equal, only then we compare the text with pattern otherwise there is no need for comparison. We will repeat the process for next set of characters from text for all possible shifts which are from s=0 to n-m. ***In practice, this algorithm is slow due to the multiplications and the modulus operations.***

```
function RabinKarp(string s[1..n], string pattern[1..m])
hpattern := hash(pattern[1..m]);  hs := hash(s[1..m])
for i from 1 to n-m+1
  if hs = hpattern
    if s[i..i+m-1] = pattern[1..m]
      return i
  hs := hash(s[i+1..i+m])
return not found
```

3. Horspool String Matching Algorithm:

The Horspool algorithm scans the characters of the pattern from right to left beginning with the rightmost one and performs the comparisons from right to left. In case of a mismatch (or a complete match of the whole pattern) it uses pre-computed shift table to shift the window to the right.

ALGORITHM *HorspoolMatching*$(P[0..m-1], T[0..n-1])$

```
//Implements Horspool's algorithm for string matching
//Input: Pattern P[0..m − 1] and text T[0..n − 1]
//Output: The index of the left end of the first matching substring
//         or −1 if there are no matches
ShiftTable(P[0..m − 1])      //generate Table of shifts
i ← m − 1                    //position of the pattern's right end
while i ≤ n − 1 do
    k ← 0                    //number of matched characters
    while k ≤ m − 1 and P[m − 1 − k] = T[i − k] do
        k ← k + 1
    if k = m
        return i − m + 1
    else i ← i + Table[T[i]]
return −1
```

4. Boyer-Moore String Matching Algorithm:

The BM algorithm scans the characters of the pattern from right to left beginning with the rightmost one and performs the comparisons from right to left. In case of a mismatch (or a

complete match of the whole pattern) it uses two pre-computed functions to shift the window to the right. These two shift functions are called the *good-suffix shift* (also called matching shift) and the *bad-character shift* (also called the occurrence shift).

The Boyer-Moore algorithm

Step 1 For a given pattern and the alphabet used in both the pattern and the text, construct the bad-symbol shift table as described earlier.

Step 2 Using the pattern, construct the good-suffix shift table as described earlier.

Step 3 Align the pattern against the beginning of the text.

Step 4 Repeat the following step until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and the text until either all m character pairs are matched (then stop) or a mismatching pair is encountered after $k \geq 0$ character pairs are matched successfully. In the latter case, retrieve the entry $t_1(c)$ from the c's column of the bad-symbol table where c is the text's mismatched character. If $k > 0$, also retrieve the corresponding d_2 entry from the good-suffix table. Shift the pattern to the right by the number of positions computed by the formula

$$d = \begin{cases} d_1 & \text{if } k = 0, \\ \max\{d_1, d_2\} & \text{if } k > 0, \end{cases}$$

where $d_1 = \max\{t_1(c) - k, 1\}$.

5. Knuth-Morris-Pratt String Matching Algorithm:

It compares the pattern with the text from left to right. It decreases the time of searching compared to the Brute Force algorithm. The basic idea behind this algorithm is that each time a mismatch is detected, the "false start" consists of characters that we have already examined. We can take advantage of this information instead of repeating comparisons with the known characters. KMP is the first algorithm for which the constant factor in the linear term, in the worst case, does not depend on the length of the pattern.

```
algorithm kmp_search:
    input:
        an array of characters, S (the text to be searched)
        an array of characters, W (the word sought)
    output:
        an integer (the zero-based position in S at which W is found)

    define variables:
        an integer, m ← 0 (the beginning of the current match in S)
        an integer, i ← 0 (the position of the current character in W)
        an array of integers, T (the table, computed elsewhere)

    while m + i < length(S) do
        if W[i] = S[m + i] then
            if i = length(W) - 1 then
                return m
            let i ← i + 1
        else
            if T[i] > -1 then
                let m ← m + i - T[i], i ← T[i]
            else
                let i ← 0, m ← m + 1

    (if we reach here, we have searched all of S unsuccessfully)
    return the length of S
```

COMPARATIVE ANALYSIS:

This project categorizes the algorithms heuristics-based and hashing-based.

A heuristics-based algorithm allows skipping some characters to accelerate the search according to certain heuristics. Some algorithms require a verification algorithm following a possible match to verify if a true match occurs.

A hashing based algorithm compares the hash values of characters in the text segment by segment with those of the characters in the patterns. If both hash values are equal, a possible match may occur. The characters in the text and those in the patterns are then compared to verify if a true match occurs.

Algorithms	Time Complexity	Search Type	Key Ideas	Approach
Brute Force	O((n-m+1)m)	Prefix	Searching with all alphabets	Linear Searching
Rabin Karp	O(m),O(M+1)	Prefix	Compare the text and patterns from their hashing functions	Hashing Based
Boyer-Moore	O(m+n),O(m)	Suffix	Bad-character and good-suffix heuristics to determine the sift distance	Heuristics Based
KMP	O(m),O(m+n)	Prefix	Construct a suffix prefix table from the pattern	Heuristics Based
Hoorspool	O(nm),O(n)	Suffix	Bad-character heuristics to determine the sift distance	Heuristics Based

METRICS

Following results are obtained for various sizes of text and pattern strings. Its shows that for large input sizes, KMP and Boyer-Moore algorithms runs faster than Brute Force algorithm, Robin Karp algorithm and Horsepool algorithm. End results are tabulated as fallows.

No. Of Character in text	**No. Of Character in Pattern**	**KMP**	**Horsepool**	**Brute Force**	**Boyer Moore**	**Robin Karp**
		Time(µs)	*Time(µs)*	*Time(µs)*	*Time(µs)*	*Time(µs)*
		Comparison	*Comparison*	*Comparison*	*Comparison*	*Comparison*
10	03	184	17	2	5	2
		2	2	10	6	4
50	07	146	18	3	6	5
		5	13	53	16	9
100	10	152	20	5	6	7
		7	25	107	21	13
1000	15	177	22	30	12	61
		61	133	1099	136	77
5000	50	329	39	118	23	307
		307	471	5026	376	455
10000	100	428	62	256	29	530
		530	700	10464	472	891

CONCLUSION:

This project reviews and profiles some typical string matching algorithms to observe their performance under various conditions and gives an insight into choosing the efficient algorithms. By analyzing these string matching algorithms, it can be concluded that Boyer-Moore, KMP string matching algorithms are efficient. Practice shows that BM Algorithm is fast in the case of larger alphabet. KMP decreases the time of searching compared to the Brute Force algorithm.