International Journal of Latest Trends in Engineering and Technology
Special Issue SACAIM 2016, pp. 221-225
e-ISSN:2278-621X

COMPARATIVE ANALYSIS ON EFFICIENCY OF SINGLE STRING PATTERN MATCHING ALGORITHMS

Enola D'Souza[1], B Shalini Pai[2] and Ms. Suchetha Vijayakumar[3]

Abstract-Data is stored in different forms but, text remains the main form of exchanging information. The manipulation of text involves several problems among which pattern matching is one of them.Pattern-matching is routinely used in various computer applications, like editors, retrieval of information etc. Pattern-matching algorithm matches the pattern exactly or approximately within the text. This paper presents the Comparative Analysis of various Pattern String matching algorithms. The highly efficient algorithms like The Brute Force Algorithm, The Karp-Rabin Algorithm, and The Boyer Moore Algorithm are used for exact or approximate patternmatching on diverse systems. After performing a detailed study on the above mentioned algorithms, we have analysed the efficiency of algorithms based on their execution time and different types of inputs provided.

Keywords- Execution Time, Text, Pattern,Window.

I. INTRODUCTION

Pattern-matching is the act of checking sequence of tokens for the presence of some pattern. It is a process which takes Pattern as inputof length 'P' and Text of length 'T', where 'P' is smaller than 'T'.Pattern matching techniques has two categories:

- Single pattern matching technique
- Multiple pattern matching technique

In single pattern matching it is required to find all occurrences of the pattern in the given input text. And if morethan one pattern is matched against the given input text simultaneously, then it is known as, multiple patternmatching.

In pattern-matching problem, it is convenient to consider that the text is examined through a window. The window delimits a factor of the text and has usually the length of the pattern. It slides along the text from left to right. During the search it is periodically shifted according to rules that are specific to each algorithm. When the window is at a certain position on the text, the algorithm checks whether the pattern occurs there or not. If there is a whole match, the position is reported.

The main objective behind the pattern-matching algorithms is to reduce the total number ofcharacter comparisons between the pattern and the text to increase the overall efficiency. The efficiency of algorithms is evaluated by their running times and the type of inputs they provided. The pattern matching algorithms are widely used in network security environments, Information Retrieval, Text Editors etc.

[1] *Aloysius Institute of Management and Information Technology Mangalore, Karnataka, India.*
[2] *Aloysius Institute of Management and Information Technology Mangalore, Karnataka, India.*
[3] *AIMIT, Beeri, Mangalore, Karnataka, India.*

II. TEXT PATTERN MATCHING ALGORITHM

A. The Brute Force Algorithm

The simplest approach for string matching problem is - The Brute ForceAlgorithm which is also known as Naive Algorithm. It follows linear search approach. As shown in the Figure1the algorithm simply tries to match the first letter of the Textand the first letter of the Pattern and checks whether these two letters are equal. If it is, then check second letters of the text and pattern. If it is not equal, then move first letter of the pattern to the second letter of the text. Then check these two letters. When we find a match, return its starting location.
Example:

Let the Text (T) be,
THIS IS A SIMPLE EXAMPLE
and the Pattern (P) be,
SIMPLE

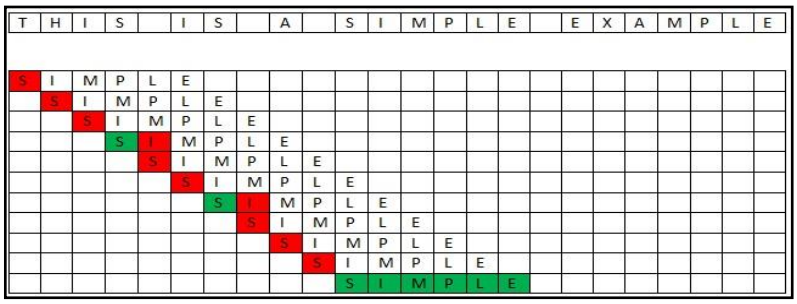

Figure1. Example for Brute-Force Algorithm

Implementation:

```
import java.util.*;
class PatternMatching
{
     int bruteForceMatcher(String text,String pattern)
     {
          int n = text.length();     //n is the length of text
          int m = pattern.length(); // m is length of pattern
          int j;
          for(int i=0; i <= (n-m); i++)
          {
               j = 0;
               while ((j < m) && (text.charAt(i+j) == pattern.charAt(j)) )
               j++;
               if (j == m)
                    return i;    // match at i
          }
          return -1;  // no match
     }// end of bruteForceMatcher()
}
public class BruteForce
{
     public static void main(String args[])
     {
          PatternMatching p= new PatternMatching();
          Scanner sc= new Scanner(System.in);
          System.out.println("Enter the Text");
          String text = sc.nextLine();
          System.out.println("Enter the Pattern");
          String pattern = sc.nextLine();
          System.out.println("Text:" + text);
          System.out.println("Pattern:" +pattern);
          int position = p.bruteForceMatcher(text,pattern);
          if (position == -1)
               System.out.println("Pattern not found");
          else
               System.out.println("Pattern starts at position " + (position));
     }
}
```

B. The Karp-Rabin Algorithm

This algorithm exploits a hash function to speed up the search. It calculates a hash value for the pattern to be searched and each subsequence of text to be compared.Then both the hash values are compared. If the hash values are not equal the algorithm will estimate the hash value for next character sequence. If the hash values are equal then the algorithm will do the brute force comparison with the pattern and the character sequence for which the hash value matched as shown in Figure2.One popular and affective rolling hash function reads every substring as a number in some ways, the base being usually large prime number.

Example:

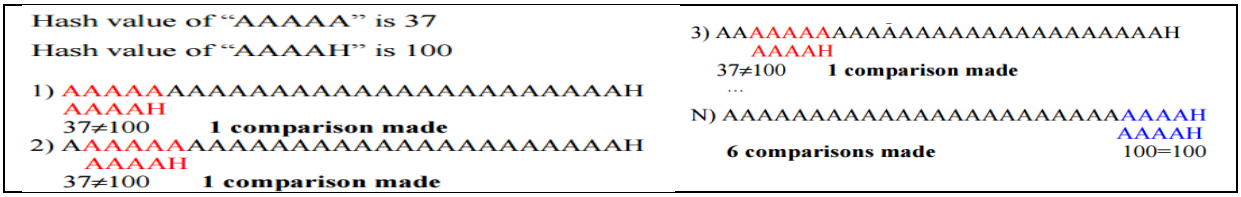

Figure2. Example for Karp-Rabin Algorithm

Implementation:

```
import java.util.*;
class RabinKarp {

    private int prime = 101;

    public int patternSearch(char[] text, char[] pattern){
        int m = pattern.length;
        int n = text.length;
        long patternHash = createHash(pattern, m - 1);
        long textHash = createHash(text, m - 1);
        for (int i = 1; i <= n - m + 1; i++) {
            if(patternHash == textHash && checkEqual(text, i - 1, i + m - 2, pattern, 0, m - 1)) {
                return i - 1;
            }
            if(i < n - m + 1) {
                textHash = recalculateHash(text, i - 1, i + m - 1, textHash, m);
            }
        }
        return -1;
    }

    private long recalculateHash(char[] str,int oldIndex, int newIndex,long oldHash, int patternLen) {
        long newHash = oldHash - str[oldIndex];
        newHash = newHash/prime;
        newHash += str[newIndex]*Math.pow(prime, patternLen - 1);
        return newHash;
    }

    private long createHash(char[] str, int end){
        long hash = 0;
        for (int i = 0 ; i <= end; i++) {
            hash += str[i]*Math.pow(prime,i);
        }
        return hash;
    }

    private boolean checkEqual(char str1[],int start1,int end1, char str2[],int start2,int end2){
        if(end1 - start1 != end2 - start2) {
            return false;
        }
        while(start1 <= end1 && start2 <= end2){
            if(str1[start1] != str2[start2]){
                return false;
            }
            start1++;
            start2++;
        }
        return true;
    }

    public static void main(String args[]){
        RabinKarp rks = new RabinKarp();
        System.out.println("The postion of the given Input for 'SimpleExample' as the Text:");
        System.out.println(rks.patternSearch("SimpleExample".toCharArray(), "pleExam".toCharArray()));
        System.out.println(rks.patternSearch("SimpleExample".toCharArray(), "Example".toCharArray()));
        System.out.println(rks.patternSearch("SimpleExample".toCharArray(), "mpel".toCharArray()));
        System.out.println(rks.patternSearch("SimpleExample".toCharArray(), "impl".toCharArray()));
        System.out.println(rks.patternSearch("SimpleExample".toCharArray(), "Sim".toCharArray()));
    }
}
```

C. The Boyer Moore Algorithm

This algorithm scans the characters of the Pattern from right to left beginning with the rightmost character and performs the comparisons from right to left. In case of a match or a mismatch it uses two pre-computed functions to shift the window to the right. These twoshift functions are called the good-suffix shift or matching shift and the bad-character shift or occurrence shift as shown in Figure3.

Example:

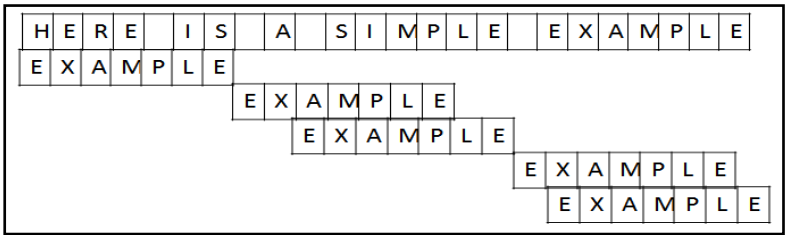

Figure3. Example for Boyer-Moore Algorithm

Implementation:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
public class BoyerMoore
{
    public void findPattern(String t, String p)
    {
        char[] text = t.toCharArray();
        char[] pattern = p.toCharArray();
        int pos = indexOf(text, pattern);
        if (pos == -1)
            System.out.println("\nNo Match\n");
        else
            System.out.println("Pattern found at position : "+ pos);
    }
    /** Function to calculate index of pattern substring **/
    public int indexOf(char[] text, char[] pattern)
    {
        if (pattern.length == 0)
            return 0;
        int charTable[] = makeCharTable(pattern);
        int offsetTable[] = makeOffsetTable(pattern);
        for (int i = pattern.length - 1, j; i < text.length;)
        {
            for (j = pattern.length - 1; pattern[j] == text[i]; --i, --j)
                if (j == 0)
                return i;
              i += Math.max(offsetTable[pattern.length - 1 - j], charTable[text[i]]);
        }
        return -1;
      }

    /** Makes the jump table based on the mismatched character information **/
    private int[] makeCharTable(char[] pattern)
    {
      final int ALPHABET_SIZE = 256;
      int[] table = new int[ALPHABET_SIZE];
      for (int i = 0; i < table.length; ++i)
            table[i] = pattern.length;
      for (int i = 0; i < pattern.length - 1; ++i)
            table[pattern[i]] = pattern.length - 1 - i;
      return table;
    }
```

```
    /** Makes the jump table based on the scan offset which mismatch occurs. **/
    private static int[] makeOffsetTable(char[] pattern)
    {
      int[] table = new int[pattern.length];
      int lastPrefixPosition = pattern.length;
      for (int i = pattern.length - 1; i >= 0; --i)
      {
          if (isPrefix(pattern, i + 1))
                lastPrefixPosition = i + 1;
            table[pattern.length - 1 - i] = lastPrefixPosition - i + pattern.length - 1;
      }
      for (int i = 0; i < pattern.length - 1; ++i)
      {
            int slen = suffixLength(pattern, i);
            table[slen] = pattern.length - 1 - i + slen;
      }
      return table;
  }

    /** function to check if needle[p:end] a prefix of pattern **/
    private static boolean isPrefix(char[] pattern, int p)
    {
        for (int i = p, j = 0; i < pattern.length; ++i, ++j)
            if (pattern[i] != pattern[j])
                return false;
        return true;
    }
    /** function to returns the maximum length of the substring ends at p and is a suffix **/
    private static int suffixLength(char[] pattern, int p)
    {
        int len = 0;
        for (int i = p, j = pattern.length - 1; i >= 0 && pattern[i] == pattern[j]; --i, --j)
            len += 1;
        return len;
    }
    /** Main Function **/
    public static void main(String[] args) throws IOException
    {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Boyer Moore Algorithm Test\n");
        System.out.println("\nEnter Text\n");
        String text = br.readLine();
        System.out.println("\nEnter Pattern\n");
        String pattern = br.readLine();
        BoyerMoore bm = new BoyerMoore();
        bm.findPattern(text, pattern);
    }
}
```

III. COMPARATIVE ANALYSIS ON STRING PATTERN MATCHING ALGORITHM

In this paper, we analysed selected Single pattern string matching algorithms on the basis of Execution time and search type. Each algorithm has certain advantages and disadvantages.

A. Algorithm Techniques: Every algorithm uses some special techniques to find pattern matching. Following table shows the different techniques used by different algorithms.

Algorithm	Techniques
Brute Force Algorithm	Each character of the pattern is compared to a substring of the text which is the length of the pattern, until there is a mismatch or a match.
Rabin–Karp string search algorithm	Hashing
Boyer–Moore string search algorithm	Use both good suffix shift and bad character shift

B. Observations on Algorithms Used For Matching:

1) Brute-Force String Search Algorithm: The "naive" approach is easy to understand and implement but it can be too slow in some cases. If the length of the text is 'n' and the length of the pattern is 'm', in the worst case it may take (n * m) iterations to complete the task.The main advantage of Brute Force Algorithm is that wide applicability, simplicity, reasonable algorithms forsome important problems (e.g., matrix multiplication, sorting, searching, string matching).Weakness of the brute-force algorithms are unacceptably slow. It takes much time as it search linearly.

2) Rabin Karp String Search Algorithm: It is a string searching algorithm that uses hashing to find any one of a set of pattern strings in a text. This algorithm works well in many practical cases, but can exhibit relatively long running times on certain examples, such as searching for a pattern string of 10,000 "A"s followed by a single "B" in a search string of 10 million "A"s.

3) Boyer–Moore String Search Algorithm: It is a particularly efficient string searching algorithm. The algorithm pre-processes the pattern that is being searched for, but not the text. Generally the algorithm gets faster as the pattern being searched for becomes longer.TheBoyer-Moore Algorithm achieves sub-linear running time by skipping characters in theinput text according to the bad character and good suffix heuristics.

C. Analysis

Table1.Table of findings for execution time(in seconds) based on various inputs for different algorithms.

Algorithm	**Length of I/P / No. of executions**	**Nature of I/P**									**Findings**
		AU			**AL**			**M**			
		1	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	
Brute Force (BFA)	Short	6	4	3	5	4	3	5	3	2	BFA is good for short strings and search is faster for lower case than uppercase letters.
	Long	8	5	3	8	7	5	9	7	6	
Karp Rabin (KRA)	Short	6	5	4	4	3	2	4	3	2	KRA results are similar to BFA but and search is faster for mixed letters compared to BFA.
	Long	8	6	5	8	7	6	9	6	5	
Boyer Moore (BMA)	Short	7	7	5	6	6	6	4	3	3	BMA is good for long strings than the short.
	Long	6	5	4	5	4	4	5	4	3	

AU- All Uppercase	**AL- All Lowercase**	**M- Mixed**

By looking at the performance of various algorithms in Table 1, we can conclude that the best algorithm in majority of the cases is Boyer-Moore. When the pattern is long, its advantage becomes significant. The reason to this can be easily explained by the fact that it could skip more characters.The Brute-Force Algorithm obtains similar results, but the performance is very slow when the pattern is very large. The Brute-Force could be a good choice if the length of pattern is very short. Rabin-Karp obtains very good results in these tests. Its results are a lot better than the naive solution and it is definitely the best choice in the situations where Boyce-Moore is not adapted.

IV. CONCLUSION

We have presented the most famousPattern String Matching Algorithms. There are various scenarios where we can use a particular type of algorithm. To answer the question: Which algorithm is the best? We implemented the three algorithms using programming language (Java), and after conducting different tests and comparisons with these implementations, the answer is that Boyer-Moore is the Best Algorithm. Not in all cases,but in the practical cases, Boyer-Moore algorithm is extremely fast on large text that is why we can consider BM algorithm as the best one. Rabin-Karp is also an effective algorithm, it is even better than BM when the pattern and the text are very small. The naive algorithm is the worst solution, with the slowest execution time.

REFERENCES

[1] https://www.cs.purdue.edu/homes/ayg/CS251/slides/chap11.pdf
[2] http://www.geeksforgeeks.org/pattern-searching-set-7-boyer-moore-algorithm-bad-character-heuristic/
[3] http://www.stoimen.com/blog/2012/03/27/computer-algorithms-brute-force-string-matching/
[4] GeorgyGimel'farb, "String matching Algorithms", COMPSCI 369 Computational Science
[5] AkhtarRasool Amrita Tiwari et al, / (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 3 (2) , 2012,3394– 3397
[6] Algorithms for String matching,Marc GOU,July 30, 2014
[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff ord Stein. Introduction to algorithms, Third edition. The MIT Press, 2009.