Experiment 8 – Write a Case Study on String Matching Algorithms
SE COMP C 34 YASH SINHA

Learning Objective: Student should be able to develop a case study on several String Matching Algorithms.

Tools: C/C++/Java/Python under Windows or Linux environment.

Executive Summary:
In computer science, string-searching algorithms, sometimes called string-matching algorithms, are an important class of string algorithms that try to find a place where one or several strings (also called patterns) are found within a larger string or text. A basic example of string searching is when the pattern and the searched text are arrays of elements of an alphabet (finite set) Σ. Σ may be a human language alphabet, for example, the letters A through Z and other applications may use a binary alphabet (Σ = {0,1}) or a DNA alphabet (Σ = {A,C,G,T}) in bioinformatics.

Background: Given a string ‘S’, the problem of string matching deals with finding whether a pattern ‘p’ occurs in ‘S’ and if ‘p’ does occur then returning position in ‘S’ where ‘p’ occurs.
O(mn) approach: One of the most obvious approaches towards the string matching problem would be to compare the first element of the pattern to be searched ‘p’, with the first element of the string ‘S’ in which to locate ‘p’. If the first element of ‘p’ matches the first element of ‘S’, compare the second element of ‘p’ with the second element of ‘S’. If a match is found, proceed likewise until the entire ‘p’ is found. If a mismatch is found at any position, shift ‘p’ one position to the right and repeat comparison beginning from first element nz

Proposed Solutions:

1. **Naive Search Algorithm:**
   The naïve approach tests all the possible placement of Pattern P [1.......m] relative to text T [1......n]. We try shift s = 0, 1.......n-m, successively and for each shift s. Compare T [s+1.......s+m] to P [1......m].
   The naïve algorithm 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 value of s.

```
1. n ← length [T]
2. m ← length [P]
3. for s ← 0 to n –m
4. do if P [1.....m] = T [s + 1....s + m]
5. then print "Pattern occurs with shift" s
```

2. **Rabin Karp Algorithm:**
   The Rabin-Karp string matching algorithm calculates a hash value for the pattern, as well as for each M-character subsequences of text to be compared. If the hash values are unequal, the algorithm will determine the hash value for next M-

character sequence. If the hash values are equal, the algorithm will analyze the pattern and the M-character sequence. In this way, there is only one comparison per text subsequence, and character matching is only required when the hash values match.

RABIN-KARP-MATCHER (T, P, d, q)

```
1. n ← length [T]
2. m ← length [P]
3. h ← $d^{m-1}$ mod q
4. p ← 0
5. $t_0$ ← 0
6. for i ← 1 to m
7. do p ← (dp + P[i]) mod q
8. $t_0$ ← ($dt_0$+T [i]) mod q
9. for s ← 0 to n–m
10. do if p = $t_s$
11. then if P [1.....m] = T [s+1.....s + m]
12. then "Pattern occurs with shift" s
13. If s < n-m
14. then $t_{s+1}$ ← (d ($t_s$-T [s+1]h)+T [s+m+1])mod q
```

3. **KMP (Knuth Morris Pratt Algorithm)**

Knuth-Morris and Pratt introduce a linear time algorithm for the string matching problem. A matching time of O (n) is achieved by avoiding comparison with an element of 'S' that have previously been involved in comparison with some element of the pattern 'p' to be matched. i.e., backtracking on the string 'S' never occurs

COMPUTE- PREFIX- FUNCTION (P)

```
1. m ←length [P]            //'p' pattern to be matched
2. Π [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 ← Π [k]
7. If P [k + 1] = P [q]
8. then k← k + 1
9. Π [q] ← k
10. Return Π
```

KMP-MATCHER (T, P)

```
1. n ← length [T]
2. m ← length [P]
3. Π← COMPUTE-PREFIX-FUNCTION (P)
4. q ← 0                    // numbers of characters matched
5. for i ← 1 to n           // scan S from left to right
6. do while q > 0 and P [q + 1] ≠ T [i]
7. do q ← Π [q]                   // next character does not match
8. If P [q + 1] = T [i]
9. then q ← q + 1                 // next character matches
10. If q = m                               // is all of p matched?
11. then print "Pattern occurs with shift" i - m
12. q ← Π [q]                              // look for the next match
```

4. **Boyer-Moore Algorithm**

The B-M algorithm takes a 'backward' approach: the pattern string (P) is aligned with the start of the text string (T), and then compares the characters of a pattern from right to left, beginning with the rightmost character.

BOYER-MOORE-MATCHER (T, P, Σ)

```
1. n ←length [T]
2. m ←length [P]
3. λ← COMPUTE-LAST-OCCURRENCE-FUNCTION (P, m, ∑ )
4. ɣ← COMPUTE-GOOD-SUFFIX-FUNCTION (P, m)
5. s ←0
6. While s ≤ n − m
7. do j ← m
8. While j > 0 and P [j] = T [s + j]
9. do j ←j-1
10. If j = 0
11. then print "Pattern occurs at shift" s
12. s ← s + ɣ[0]
13. else s ← s + max (ɣ [j], j -  λ[T[s+j]])
```

5. **String matching with finite automata**

The string-matching automaton is a very useful tool which is used in string matching algorithm. It examines every character in the text exactly once and reports all the valid shifts in O (n) time. The goal of string matching is to find the location of specific text pattern within the larger body of text (a sentence, a paragraph, a book, etc.)

FINITE- AUTOMATON-MATCHER (T,δ, m),

```
1. n ← length [T]
2. q ← 0
3. for i ← 1 to n
4. do q ← δ (q, T[i])
5. If q =m
6. then s←i-m
7. print "Pattern occurs with shift s" s
```

COMPUTE-TRANSITION-FUNCTION (P, Σ)

```
1. m ← length [P]
2. for q ← 0 to m
3. do for each character a ∈ Σ*
4. do k ← min (m+1, q+2)
     5. repeat k←k-1
6. Until
7. δ(q,a)←k
8. Return δ
```

Case Evaluation:

Application:

1. **Spelling Checker:** Trie is built based on a predefined set of patterns. Then, this trie is used for string matching. The text is taken as input, and if any such pattern occurs, it is shown by reaching the acceptance state.
2. **Spam filters:** Spam filters use string matching to discard the spam. For example, to categorize an email as spam or not, suspected spam keywords are searched in the content of the email by string matching algorithms. Hence, the content is classified as spam or not.
3. **Search engines or content search in large databases:** To categorize and organize data efficiently, string matching algorithms are used. Categorization is done based on the search keywords. Thus, string matching algorithms make it easier for one to find the information they are searching for.

Analysis:

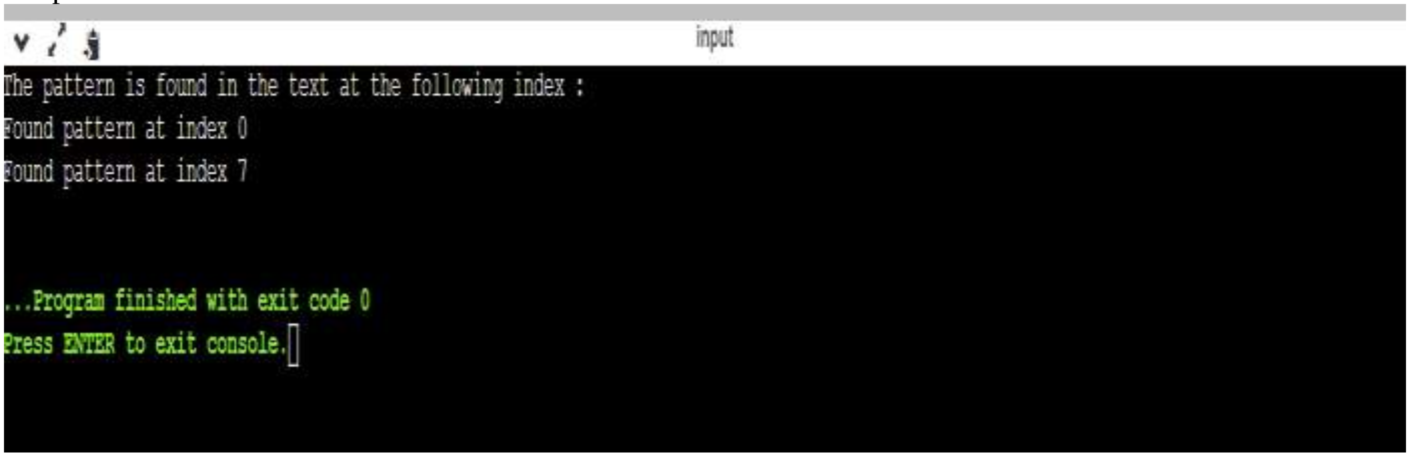
Algorithm	Preprocessing Time	Matching Time
Naive	O	(O (n - m + 1)m)
Rabin-Karp	O(m)	(O (n - m + 1)m)
Finite Automata	O(m|Σ|)	O (n)
Knuth-Morris-Pratt	O(m)	O (n)
Boyer-Moore	O(|Σ|)	(O ((n - m + 1) + |Σ|))

<u>**Implementation:**</u>

<u>**Code;**</u>

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>
void prefixSuffixArray(char* pat, int M, int* pps) {
  int length = 0;
  pps[0] = 0;
  int i = 1;
  while (i < M) {
    if (pat[i] == pat[length]) {
      length++;
      pps[i] = length;
      i++;
    } else {
      if (length != 0)
      length = pps[length - 1];
      else {
        pps[i] = 0;
        i++;
      }
    }
  }
}
void KMPAlgorithm(char* text, char* pattern) {
  int M = strlen(pattern);
  int N = strlen(text);
  int pps[M];
  prefixSuffixArray(pattern, M, pps);
  int i = 0;
  int j = 0;
  while (i < N) {
    if (pattern[j] == text[i]) {
      j++;
      i++;
    }
    if (j == M) {
      printf("Found pattern at index %d\n", i - j);
      j = pps[j - 1];
    }
    else if (i < N && pattern[j] != text[i]) {
      if (j != 0)
      j = pps[j - 1];
      else
      i = i + 1;
    }
  }
}
int main() {
  char text[] = "xyztrwqxyzfg";
  char pattern[] = "xyz";
  printf("The pattern is found in the text at the following index : \n");
  KMPAlgorithm(text, pattern);
  return 0;
}
```

Output:



```
The pattern is found in the text at the following index :
Found pattern at index 0
Found pattern at index 7


...Program finished with exit code 0
Press ENTER to exit console.
```

Learning Outcome: The student should have the ability to solve the problem based on the String matching algorithm.

Course Outcomes: Upon completion of the course students will be able to apply all the string matching algorithm to given data.

Conclusion:

1. In this we learnt about the different types of string matching algorithms.
2. We analyze the time complexity of all the string matching algorithms

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	
Marks Obtained				