**DESIGN AND ANALYSIS OF ALGORITHMS**

**MINI PROJECT**

**COMPOSITE BOYER MOORE ALGORITHM**

Course Code: IT252



**Submitted to:**

Ranjit Kolkar

Department of Information Technology

NITK Surathkal

Sachin – 15IT234

**Certificate:**

This is to certify that the B-Tech Mini project entitled “**DAA MINI PROJECT”** submitted by Sachin (Roll No.: 15IT234), as the record of the work carried out by then is accepted as the B-Tech Mini project submission in partial fulfillment of the requirements for mandatory learning course of IT252, Computer Graphics in the Department of Information Techonlogy.

Signature of the teacher

-----------------------

**Abstract**

The string matching problem has found wide application in computer science,molecular biology, genetic engineering,semantics and many other fields. In this paper, we give analysis to several classical algorithms, KMP, BM and their improvements. Then, by compositing the main method of the BM algorithm, we propose a new algorithm — the Composite BM algorithm (CBM). Differing from the BM algorithm that only uses current matching information, the CBM algorithm tries to take full advantage of the historical matching information. In this way, CBM accelerates the forward speed of the pattern during the matching, and hence the whole string matching process is sped up effectively. Finally, for binary matching, we made random test to BM and CBM,the result shown the efficiency of CBM is higher than of BM.

**Contents:**

Cover Page . . . . . . . . . . . . . . . . . . . . . . . . . *1*

Contents . . . . . . . . . . . . . . . . . . . . . . . . . *2*

Certificate . . . . . . . . . . . . . . . . . . . . . . . . . *3*

Synopsis . . . . . . . . . . . . . . . . . . . . . . . . . *4*

Introduction . . . . . . . . . . . . . . . . . . . . . . . . . *5*

Objective . . . . . . . . . . . . . . . . . . . . . . . . . 6

Methodology . . . . . . . . . . . . . . . . . . . . . . . . . 6

System Specifications . . . . . . . . . . . . . . . . . . . . . . . . . 7

Work done (screenshots) . . . . . . . . . . . . . . . . . . . . . . . . . 8-9

Future Works . . . . . . . . . . . . . . . . . . . . . . . . . 10

Result and discussion . . . . . . . . . . . . . . . . . . . . . . . . . . 10

Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . 11

References . . . . . . . . . . . . . . . . . . . . . . . . . 12

Appendix (code) . . . . . . . . . . . . . . . . . . . . . . . . . . 13-24

*(This document consists of* *pages in total, including both the cover pages.)*

**Introduction**

String matching is the most basic problem in the area of pattern matching. Not only is it widely used in the application of data compression, text editing, information retrieval etc, but it also plays an important role in the area of network security. Therefore, string matching algorithm is a basic component in many operating systems. Furthermore, string matching techniques also have some relations with other string operation problems. Including its application in the field of computer science, string matching problem also finds important applications in the field of molecular biology, genetic engineering, semantics and etc. Designing efficient and effective string matching algorithm will not only be very useful for pattern matching but also promote relevant application’s development.

**Literature survey:-**

**Outcome of literature survey:-**

The verybasic and conventional string matching strategy is Brute Force Algorithm which considers all possible cases and taking shifts only one place to right even match or mismatch condition occurs anywhere. This algorithm also known as Naives approach.

Avoidingnumerous comparisons in brute force algorithm,In 1970 Morris and Pratt algorithm was proposed which has linear behavior. This algorithm is based on pre-processing of pattern and compares character from left to right and if mismatch occurs, it skips some character based on pre-processing phase.

In 1977 Knuth Morris Pratt introduced an algorithm having a choice of improvements in Morris and Pratt algorithm. KMP has same time complexity as Morris and Pratt algorithm but searching performance found to be much

better than Morris and Pratt algorithm.

In 1977 Boyer and Moore also proposed algorithm which compares character from right to left. There are so many multiple pattern string matching algorithms has already been proposed in past decades such

as: In 1975 Aho-Corasick algorithm was presented by Alfred V. Aho and Margaret J. Corasick, which constructs automata for patterns in pre-processing phase. Commentz Walter proposed an algorithm which was based on Aho-Corasick and Boyer Moore algorithm, Rabin Karp algorithm is also used to search multiple patterns.

**Problem statement:-**

String matching is a problem of finding occurrence(s) of a pattern string within another string or body of text. The problem is also known as exact string matching, string searching, and text searching. It can be formalized as follows:

Given a text txt[0..n-1] and a pattern pat[0..m-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that n > m.

Examples:

Input: txt[] = "THIS IS A TEST TEXT"

pat[] = "TEST"

Output: Pattern found at index 10

Input: txt[] = "AABAACAADAABAABA"

pat[] = "AABA"

Output: Pattern found at index 0

Pattern found at index 9

Pattern found at index 12

**Objective:-**

* String matching algorithm is used everywhere in text editors as “search” and “substitute” commands.
* Our basic objective is to find such an algorithm which can considerably reduce the searching time of string in a big file, without doing a lot of computations.
* This algorithm can also be used in search engines to compare the the text taken as input with the search history or the keywords closest to the text taken.

**Methodology and Work done:-**

**1)Brute force algorithm:-**

In this algorithm we slide the pattern over text one by one and check for a match. If a match is found, then slides by 1 again to check for subsequent matches. This method is also known as **“Naive pattern searching”**.

**Code:-**

#include<stdio.h>

#include<string.h>

void search(char \*pat, char \*txt)

{

int count=0;

int M = strlen(pat);

int N = strlen(txt);

/\* A loop to slide pat[] one by one \*/

for (int i = 0; i <= N - M; i++)

{

int j;

count++;

/\* For current index i, check for pattern match \*/

for (j = 0; j < M; j++)

{

if (txt[i+j] != pat[j])

{

break;

}

count++;

}

if (j == M) // if pat[0...M-1] = txt[i, i+1, ...i+M-1]

printf("Pattern found at index %d \n", i);

}

printf("Total no. of times the loop has run is %d",count);

}

/\* Driver program to test above function \*/

int main()

{

char txt[1000];

char pat[50];

printf("Enter the text : ");

gets(txt);

printf("Enter the pattern : ");

gets(pat);

search(pat, txt);

return 0;

}

**2) Boyer Moore Algorithm:-**

The Boyer Moore algorithm does preprocessing for the same reason. It preporcesses the pattern and creates different arrays for both heuristics. At every step, it slides the pattern by max of the slides suggested by the two heuristics. So it uses best of the two heuristics at every step. Unlike the previous pattern searching algorithms, Boyer Moore algorithm starts matching from the last character of the pattern.

The character of the text which doesn’t match with the current character of pattern is called the Bad Character. Whenever a character doesn’t match, we slide the pattern in such a way that aligns the bad character with the last occurrence of it in pattern. We preprocess the pattern and store the last occurrence of every possible character in an array of size equal to alphabet size. If the character is not present at all, then it may result in a shift by m (length of pattern). Therefore, the bad character heuristic takes **O(n/m)** time in the best case.

The Bad Character Heuristic may take **O(mn)** time in worst case. The worst case occurs when all characters of the text and pattern are same. For example, txt[] = “AAAAAAAAAAAAAAAAAA” and pat[] = “AAAAA”.

**code:-**

# include <limits.h>

# include <string.h>

# include <stdio.h>

# define NO\_OF\_CHARS 256

// A utility function to get maximum of two integers

int max (int a, int b) { return (a > b)? a: b; }

// The preprocessing function for Boyer Moore's bad character heuristic

void badCharHeuristic( char \*str, int size, int badchar[NO\_OF\_CHARS])

{

int i;

// Initialize all occurrences as -1

for (i = 0; i < NO\_OF\_CHARS; i++)

badchar[i] = -1;

// Fill the actual value of last occurrence of a character

for (i = 0; i < size; i++)

badchar[(int) str[i]] = i;

}

/\* A pattern searching function that uses Bad Character Heuristic of

Boyer Moore Algorithm \*/

void search( char \*txt, char \*pat)

{

int count=0;

int m = strlen(pat);

int n = strlen(txt);

int badchar[NO\_OF\_CHARS];

/\* Fill the bad character array by calling the preprocessing

function badCharHeuristic() for given pattern \*/

badCharHeuristic(pat, m, badchar);

int s = 0; // s is shift of the pattern with respect to text

while(s <= (n - m))

{

int j = m-1;

count++;

/\* Keep reducing index j of pattern while characters of

pattern and text are matching at this shift s \*/

while(j >= 0 && pat[j] == txt[s+j])

j--;

count++;

/\* If the pattern is present at current shift, then index j

will become -1 after the above loop \*/

if (j < 0)

{

printf("\npattern occurs at shift = %d", s);

/\* Shift the pattern so that the next character in text

aligns with the last occurrence of it in pattern.

The condition s+m < n is necessary for the case when

pattern occurs at the end of text \*/

s += (s+m < n)? m-badchar[txt[s+m]] : 1;

}

else

/\* Shift the pattern so that the bad character in text

aligns with the last occurrence of it in pattern. The

max function is used to make sure that we get a positive

shift. We may get a negative shift if the last occurrence

of bad character in pattern is on the right side of the

current character. \*/

s += max(1, j - badchar[txt[s+j]]);

}

printf("\nThe loop runs %d no. of times\n",count);

}

/\* Driver program to test above funtion \*/

int main()

{

char txt[1000];

char pat[50];

printf("Enter the text : ");

gets(txt);

printf("Enter the pattern : ");

gets(pat);

search(txt, pat);

return 0;

}

**3)Composite Boyer Moore Algorithm:-**

According to the discussion in section 3, and referring the composite technology in Computational Mathematics, we propose a new string matching algorithm — Composite BM (CBM) algorithm in this paper. The key issue of the CBM algorithm is how to utilize the history comparison information achieved at previous iteration. So we construct a two-dimensional table Jump[m][m]. Jump[i][j] denotes the shift distance of pattern P, when the mismatch at previous iteration appears at p[i], and the mismatch at current iteration

appears at p[j]. This table is only related to pattern P. Once Jump[m][m] is constructed, it can be utilized for searching P in different texts. The comparison principle of algorithm CBM is shown in Figure 1. Suppose P is at place P0 at previous iteration, and the mismatch appears at index i of P0; and suppose P is at place P1 at current iteration, the mismatch appears at index j of P1; then P2, P’s new position, must meet following conditions: its substring at B matches with P1’s substring at B; its character at b does not match P1’s character at j; its substring at A matches P0’s substring at A; and its character at a does not matches with P1’s character at i. Above four matching conditions make a large shift distance Jump[i][j] for pattern P.

**CODE**

# include <limits.h>

# include <string.h>

# include <stdio.h>

# define NO\_OF\_CHARS 256

int cbmtable(int i, int j , int m ,int j\_jump,int k\_jump, char \*p){

int IsMatch =1;

int k,jump = k\_jump;

for(; jump < m ; jump++){

IsMatch = 1;

for( k = m-1; (k>j)&& (k>=jump); k-- ){

if(p[k]!= p[k-jump]){

IsMatch = 0;

break;

}

}

if(!IsMatch) continue;

if((j>= jump)&&(p[j]==p[j-jump]))continue;

IsMatch = 1;

int l,delta = jump + j\_jump;

for(l = m-1; (l>i)&& (l>=delta);l--){

if(p[l]!=p[l-delta]){

IsMatch = 0;

break;

}

}

if(!IsMatch) continue;

if((i>= delta)&&(p[i]==p[i-delta])) continue;

return jump;

}

return jump;

}

int max (int a, int b) { return (a > b)? a: b; }

void badCharHeuristic( char \*str, int size, int badchar[NO\_OF\_CHARS])

{

int i;

for (i = 0; i < NO\_OF\_CHARS; i++)

badchar[i] = -1;

for (i = 0; i < size; i++)

badchar[(int) str[i]] = i;

}

void search( char \*txt, char \*pat)

{

int m = strlen(pat);

int n = strlen(txt);

// int i,j,cbmt[m][m];

int j\_jump,k\_jump;

int badchar[NO\_OF\_CHARS];

int count=0;

badCharHeuristic(pat, m, badchar);

int s = 0;

while(s <= (n - m))

{count++;

int j = m-1;

while(j >= 0 && pat[j] == txt[s+j])

j--;

count++;

if (j < 0)

{

printf("\npattern occurs at shift = %d\n", s);

j\_jump= (s+m < n)? m-badchar[txt[s+m]] : 1;

s +=j\_jump;

}

else{

j\_jump = max(1, j - badchar[txt[s+j]]);

s+=j\_jump;

}

if(s>n-m)

break;

int k= m-1;

while(k>= 0 && pat[k] == txt[s+k])

k--;

count++;

if (k < 0)

{

printf("\npattern occurs at shift = %d\n", s);

k\_jump= (s+m < n)? m-badchar[txt[s+m]] : 1;

s+=k\_jump;

}

else{

k\_jump= max(1, k - badchar[txt[s+k]]);

s+=k\_jump;

}

int l= m-1;

while(l >= 0 && pat[l] == txt[s+l])

l--;

if(l< 0){

printf("\npattern occurs at shift = %d\n", s);

}

if(s>n-m)

break;

if((s<= n-m)&&(k>=0 )&&(j>=0)){

int jump = cbmtable(j,k ,m,j\_jump,k\_jump,pat);

count++;

s+=jump;

}

}

}

int main()

{

int i, n, p;

char word[26];

time\_t t;

n = 5000;

p = 10;

char str[n];

char pat[p];

srand((unsigned) time(&t));

for( i = 0 ; i < n ; i++ ) {

str[i] = 97 + (rand() % 26);

}

for(i = 0 ; i < p ; i++){

pat[i] = 97 + (rand() % 26);

}

int num = 1 + rand() % 100;

int index,pt;

for(i = 1 ; i <num ; i++){

index = rand()%((i)\*(n/num));

for(pt=0;(pt<p)&&(index+pt<n);pt++){

str[pt+index] = pat[pt];

}

}

/\*

for( i = 0 ; i < n ; i++ ) {

printf("%c",str[i]);

}

printf("\n\n");

for(i = 0 ; i < p ; i++){

printf("%c",pat[i]);

}

\*/

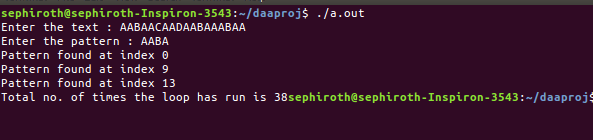
search(str, pat);

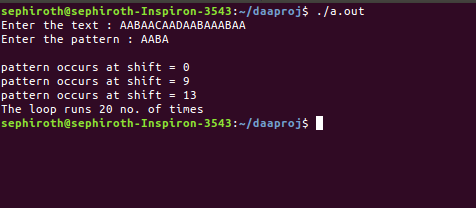
return 0;

}

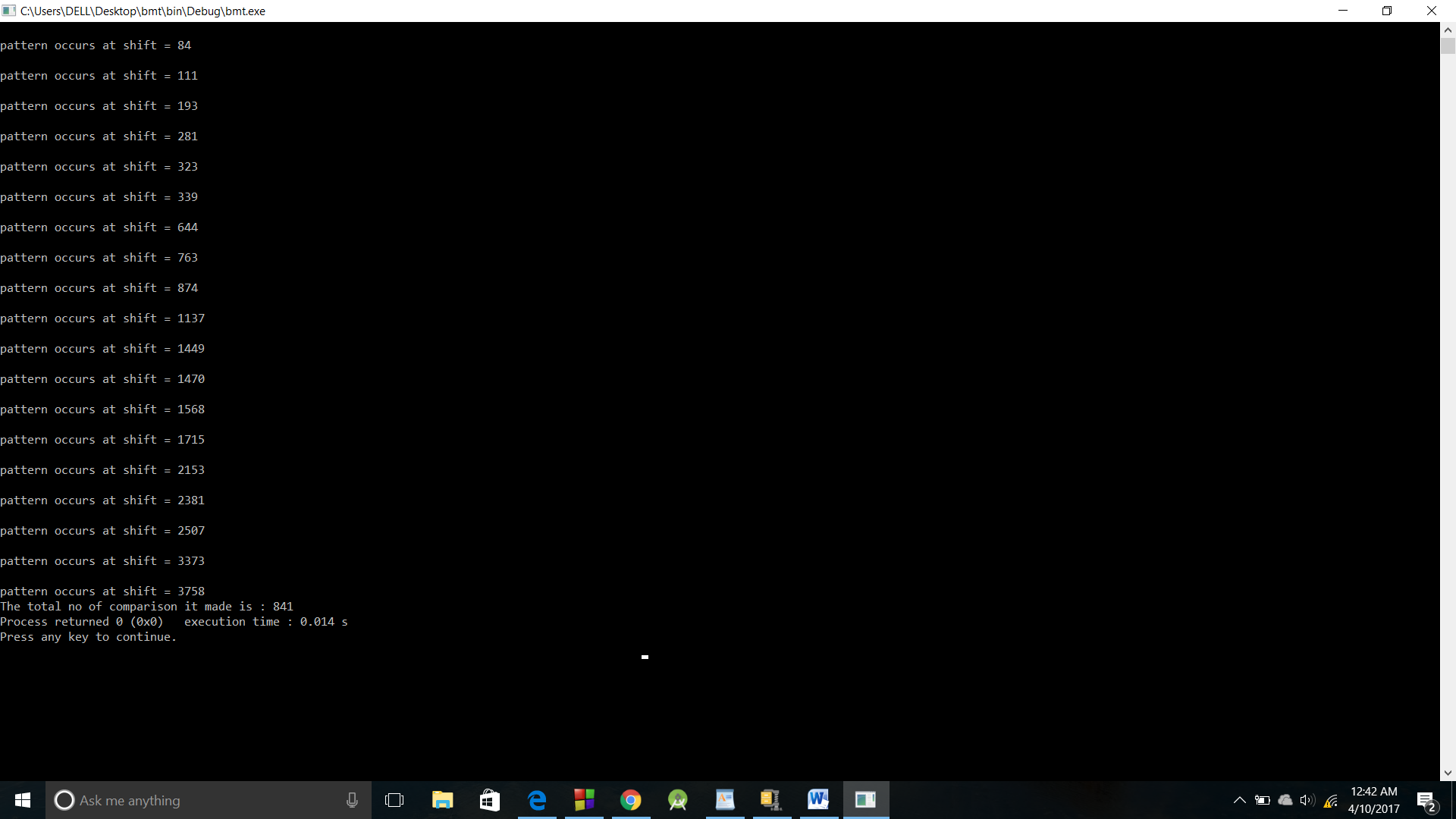
**Outputs (Screen Shots):-**

**Brute force:-**



**Boyer Moore Algorithm:-**

**CBM OUTPUT:-**



**Result and Analysis:-**

The efficiency of BM algorithm depends on the size

of the alphabet, especially for binary matching, BM

does not do quite as well. In view of this situation,

we had experimented and compared BM and CBM.

In the experiment, we set text T to a random binary

string of length 5000, set pattern P to random binary

string of length 11 to 20, and counted the numbers of

character comparisons of the matching.

**Conclusion:-**

The efficiency of BM algorithm depends on the size of the alphabet, especially for binary matching, BM does not do quite as well [1]. In view of this situation, we had experimented and compared BM and CBM. In the experiment, we set text T to a random binary string of length 5000, set pattern P to random binary string of length 11 to 20, and counted the numbers of character comparisons of the matching.

the numbers of character comparisons of CBM is about 84% of BM.

Generally, in the case of small alphabet and long pattern, values in Jump[m][m] that is close to the right column are usually larger than the corresponding values in Jump[m], and the matching efficiency are improved. Binary searching in Computer Science and DNA sequence tests in genetic engineering are such kind of applications.

**REFERENCES**

[1] Basse S., and A. V. Gelder, *Computer Algorithms,* Addison Wesley, 2000.

[2] R. S. Boyer, and J. S. Moore, “A Fast String Searching Algorithm”, Communications of the ACM, 1977, 20(10):762-772.

[3] R. N. Horspool, “Practical Fast Searching in Strings”, Software Practice and Experience, 1980, 10(6):501-506.

[4] D. E. Knuth, J. H. Morris, and V. R. Pratt, “Fast Pattern Matching in Strings”, SIAM Journal on Computing, 1977, 6(2):323-350.

[5] Y. Lu, S. Lu, and J. L. Ram, “Fast search in DNA sequence

databases using punctuation and indexing”, Proceedings of the

2nd IASTED international conference on Advances in computer science and technology, p.351-356, January 23-25, 2006, Puerto Vallarta, Mexico.

[6] D. M. Sunday, “A very fast substring search algorithm”,

Communications of the ACM, v.33 n.8, p.132-142, Aug. 1990.

[7] Renchao Jin, Enmin. Song, “A pre-processing algorithm for improving efficiency of the Boyer-Moore algorithm”, J. Huazhong Univ. of Sci. & Tech. (Nature Science Edition)(in Chinese), 2005, (33):265-267.

[8] Yi Qian and Yibin Hou, “A Fast String Matching Algorithm”, MINI-MICRO SYSTEMS ((in Chinese)), 2004, 25(3):410-413.