Experiment No. 6 **Internet Algorithms**

Date:

Aim: Write a C program to implement the following program using internet algorithms

strategy

a. KMP (Knuth-Morris-Pratt Algorithm)
b. BM (Boyer-Moore Algorithm)
c. Huffman Encoding
d. LCS (Longest Common Subsequence)

THEORY:

- Internet algorithms are specifically designed algorithms used in various aspects of the internet to solve specific problems efficiently.

- These algorithms address challenges related to data transmission, routing, network optimization, information retrieval, and security, among others.

- Internet algorithms often consider factors such as latency, bandwidth, scalability, fault tolerance, and resource constraints to ensure efficient and
reliable operations.

- Text processing algorithms in modern internet algorithms focus on analyzing and extracting information from textual data to enable various applications such as search engines, language translation, sentiment analysis, and natural language processing.

- Advantages: Enhanced user experience through personalization and efficient data processing.
- Disadvantages: Potential reinforcement of filter bubbles and biased outcomes.

a. KMP (Knuth-Morris-Pratt Algorithm)

Date:

The **Knuth-Morris-Pratt (KMP)** algorithm is a string matching algorithm that finds occurrences of a pattern within a text by utilizing a preprocessed table to skip unnecessary comparisons, resulting in efficient pattern search.

Problem Statement:

Write a c program to implement KMP on the following string :

i = 112221131224311221

j = 11221

Algorithm

Algorithm KMPMatch(T, P):
  Input: Strings T (text) with n characters and P (pattern) with m ch
  Output: Starting index of the first substring of T matching P, or a
    that P is not a substring of T
  $f \leftarrow$ KMPFailureFunction(P) {construct the failure functi
  $i \leftarrow 0$
  $j \leftarrow 0$
  while $i < n$ **do**
    if $P[j] = T[i]$ **then**
      if $j = m - 1$ **then**
        return $i - m + 1$ {a match!}
      $i \leftarrow i + 1$
      $j \leftarrow j + 1$
    else if $j > 0$ {no match, but we have advanced in P} **then**
      $j \leftarrow f(j - 1)$ {j indexes just after prefix of P that mu
    else
      $i \leftarrow i + 1$
  return "There is no substring of T matching P."

Algorithm KMPFailureFunction(P):
Input: String P (pattern) with m characters
Output: The failure function f for P, which maps j to the length
prefix of P that is a suffix of $P[1..j]$
$i \leftarrow 1$
$j \leftarrow 0$
$f(0) \leftarrow 0$
while $i < m$ **do**
if $P[j] = P[i]$ **then**
{we have matched $j+1$ characters}
$f(i) \leftarrow j+1$
$i \leftarrow i+1$
$j \leftarrow j+1$
else if $j > 0$ **then**
{j indexes just after a prefix of P that must match}
$j \leftarrow f(j-1)$
else
{we have no match here}
$f(i) \leftarrow 0$
$i \leftarrow i+1$

Time Complexity & Space Complexity

The time complexity of the Knuth-Morris-Pratt (KMP) algorithm is O(n + m), where n is the length of the text and m is the length of the pattern.

The space complexity of KMP is O(m), where m is the length of the pattern. This space is required to store the partial match table used for efficient pattern matching.

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void KMPFailureFunction(char *P, int m,
int *f) {
int i = 1, j = 0;
f[0] = 0;

while (i < m) {
if (P[i] == P[j]) {
f[i] = j + 1;
i++;
j++;
} else if (j > 0) {
j = f[j - 1];
} else {
f[i] = 0;
i++;
}
}
}
```

```
int KMPMatch(char *T, char *P) {
int n = strlen(T);
int m = strlen(P);

int *f = (int *)malloc(m * sizeof(int));
KMPFailureFunction(P, m, f);

int i = 0, j = 0;
while (i < n) {
if (T[i] == P[j]) {
if (j == m - 1) {
free(f);
return i - j;
}
i++;
j++;
} else if (j > 0) {
j = f[j - 1];
} else {
i++;
}
}
free(f);
return -1;
}

int main() {
char T[100];
char P[100];

printf("Enter string T: ");
scanf("%s", T);

printf("Enter string P: ");
scanf("%s", P);

int result = KMPMatch(T, P);
if (result != -1) {
printf("Pattern found at index %d\n",
result);
} else {
printf("Pattern not found in the text.\n");
}

return 0;
}
```

Output

Enter string T: 112221131224311221

Enter string P: 11221

Pattern found at index 13

b. BM (Boyer-Moore Algorithm)

Date:

The **Boyer-Moore** algorithm is a string searching algorithm that efficiently finds occurrences of a pattern within a text by utilizing a combination of bad character and good suffix heuristics to skip comparisons, resulting in faster pattern matching.

Problem Statement:

Write a c program to implement Boyer moore algorithm on the following string :

i = abbbcbbacbbcabca

j = cabca

Algorithm

Algorithm BMMatch(T, P):
Input: Strings T (text) with n characters and P (pattern) with m characters
Output: Starting index of the first substring of T matching P, or an indication that P is not a substring of T

```
compute function last
i ← m − 1
j ← m − 1
repeat
  if P[j] = T[i] then
    if j = 0 then
      return i          {a match!}
    else
      i ← i − 1
      j ← j − 1
  else
    i ← i + m − min(j, 1 + last(T[i]))          { jump step }
    j ← m − 1
until i > n − 1
return "There is no substring of T matching P."
```

Time Complexity and Space Complexity

The time complexity of the Boyer-Moore algorithm is typically O(n + m), where n is the length of the text and m is the length of the pattern.

The space complexity of Boyer-Moore is O(m), where m is the length of the pattern. This space is required to store information related to the bad character rule and the good suffix rule, such as the bad character shift table and the suffix table.

Code

```
#include <stdio.h>
#include <string.h>
#define MAX_TEXT_SIZE 1000
#define MAX_PATTERN_SIZE 100
int max(int a, int b) {
return (a > b) ? a : b; }
void computeBadCharTable(char* pattern, int patternSize, int badCharTable[]) {
int i;
for (i = 0; i < 256; i++) {
badCharTable[i] = -1;
}
for (i = 0; i < patternSize; i++) {
badCharTable[(int)pattern[i]] = i;
}}
void boyerMoore(char* text, char* pattern) {
int textSize = strlen(text);
int patternSize = strlen(pattern);
int badCharTable[256];
computeBadCharTable(pattern, patternSize, badCharTable);
int shift = 0;
while (shift <= (textSize - patternSize)) {
int j = patternSize - 1;
while (j >= 0 && pattern[j] == text[shift + j]) {
j--;
```

```
}

if (j < 0) {

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

shift += (shift + patternSize < textSize) ? patternSize - badCharTable[text[shift +
patternSize]] : 1;

} else {

int badCharShift = j - badCharTable[text[shift + j]];

shift += max(1, badCharShift);

}}}

int main() {

char text[MAX_TEXT_SIZE];

char pattern[MAX_PATTERN_SIZE];

printf("Enter the text: ");

fgets(text, sizeof(text), stdin);

text[strcspn(text, "\n")] = '\0';

printf("Enter the pattern to search: ");

fgets(pattern, sizeof(pattern), stdin);

pattern[strcspn(pattern, "\n")] = '\0';

boyerMoore(text, pattern);

return 0;

}
```

Output:

```
Enter the text: abbbcbbacbbcabca

Enter the pattern to search: cabca

Pattern found at index 11
```

C. Huffman Encoding

Date:

Huffman encoding is a text compression algorithm that uses variable-length encoding to reduce the size of the text data. It was developed by David A. Huffman. The Huffman encoding algorithm works by assigning shorter codes to frequently occurring characters or patterns in the text and longer codes to less frequent ones.

Problem Statement:

Write a c program to implement Huffman encoding algorithm on the following string :

X = “three missionaries and three cannibals”

Algorithm

Algorithm Huffman(X):
Input: String X of length n with d distinct characters
Output: Coding tree for X
Compute the frequency $f(c)$ of each character c of X.
Initialize a priority queue Q.
for each character c in X **do**
Create a single-node binary tree T storing c.
Insert T into Q with key $f(c)$.
while Q.size() > 1 **do**
$f_1 \leftarrow Q$.minKey()
$T_1 \leftarrow Q$.removeMin()
$f_2 \leftarrow Q$.minKey()
$T_2 \leftarrow Q$.removeMin()
Create a new binary tree T with left subtree T_1 and right subtree T_2.
Insert T into Q with key $f_1 + f_2$.
return tree Q.removeMin()

Time Complexity and Space Complexity

The time complexity of Huffman encoding, a lossless data compression algorithm, is typically, O(nlogn+m), where n is the number of symbols in the input data and m is the number of unique symbols.

The space complexity of Huffman encoding is O(n+mlogn), where n is the number of symbols in the input data and m is the number of unique symbols.

Code

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX_CHAR 256

struct MinHeapNode {

char data;

unsigned frequency;

struct MinHeapNode* left, * right;

};

struct MinHeap {

unsigned size;

unsigned capacity;

struct MinHeapNode** array;

};

struct MinHeapNode* createNode(char data, unsigned frequency){

struct MinHeapNode* node = (struct MinHeapNode*)malloc(sizeof(struct MinHeapNode));

node->left = node->right = NULL;

node->data = data;

node->frequency = frequency;

return node;

}

struct MinHeap* createMinHeap(unsigned capacity){

struct MinHeap* minHeap = (struct MinHeap*)malloc(sizeof(struct MinHeap));

minHeap->size = 0;

minHeap->capacity = capacity;

minHeap->array = (struct MinHeapNode**)malloc(minHeap->capacity * sizeof(struct MinHeapNode*));

return minHeap;

}

void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b){

struct MinHeapNode* t = *a;

*a = *b;

*b = t;

}

void minHeapify(struct MinHeap* minHeap, int idx){

int smallest = idx;

int left = 2 * idx + 1;

int right = 2 * idx + 2;

if (left < minHeap->size && minHeap->array[left]->frequency < minHeap->array[smallest]->frequency)

smallest = left;

if (right < minHeap->size && minHeap->array[right]->frequency < minHeap->array[smallest]->frequency)

smallest = right;

if (smallest != idx) {

swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);
```

```
minHeapify(minHeap, smallest);

}}

int isSizeOne(struct MinHeap* minHeap){

return (minHeap->size == 1);

}

struct MinHeapNode* extractMin(struct
MinHeap* minHeap){

struct MinHeapNode* temp = minHeap-
>array[0];

minHeap->array[0] = minHeap-
>array[minHeap->size - 1];

--minHeap->size;

minHeapify(minHeap, 0);

return temp;

}

void insertMinHeap(struct MinHeap*
minHeap, struct MinHeapNode*
minHeapNode){

++minHeap->size;

int i = minHeap->size - 1;

while (i && minHeapNode->frequency <
minHeap->array[(i - 1) / 2]->frequency) {

minHeap->array[i] = minHeap->array[(i -
1) / 2];

i = (i - 1) / 2;

}

minHeap->array[i] = minHeapNode;

}

void buildMinHeap(struct MinHeap*
minHeap){

int n = minHeap->size - 1;

int i;

for (i = (n - 1) / 2; i >= 0; --i)

minHeapify(minHeap, i);

}

int isLeaf(struct MinHeapNode* root){

return !(root->left) && !(root->right);

}

struct MinHeap*
createAndBuildMinHeap(char data[],
unsigned frequency[], int size)

{

struct MinHeap* minHeap =
createMinHeap(size);

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

minHeap->array[i] = createNode(data[i],
frequency[i]);

minHeap->size = size;

buildMinHeap(minHeap);

return minHeap;

}

struct MinHeapNode*
buildHuffmanTree(char data[], unsigned
frequency[], int size){

struct MinHeapNode* left, * right, * top;

struct MinHeap* minHeap =
createAndBuildMinHeap(data, frequency,
size);

while (!isSizeOne(minHeap)) {

left = extractMin(minHeap);

right = extractMin(minHeap);
```

```
top = createNode('$', left->frequency + right->frequency);

top->left = left;

top->right = right;

insertMinHeap(minHeap, top);

}

return extractMin(minHeap);

}


void printCodes(struct MinHeapNode* root, int arr[], int top)

{

if (root->left) {

arr[top] = 0;

printCodes(root->left, arr, top + 1);

}


if (root->right) {

arr[top] = 1;

printCodes(root->right, arr, top + 1);

}


if (isLeaf(root)) {

printf("%c: ", root->data);

for (int i = 0; i < top; ++i)

printf("%d", arr[i]);

printf("\n");

}
```

```
}


// Function to perform Huffman coding

void HuffmanCodes(char data[], unsigned frequency[], int size)

{

struct MinHeapNode* root = buildHuffmanTree(data, frequency, size);


int arr[MAX_CHAR], top = 0;


printCodes(root, arr, top);

}


int main()

{

char sentence[1000];

printf("Enter a sentence: ");

fgets(sentence, sizeof(sentence), stdin);

sentence[strcspn(sentence, "\n")] = '\0';

unsigned frequency[MAX_CHAR] = { 0 };

for (int i = 0; sentence[i] != '\0'; ++i)

frequency[(int)sentence[i]]++;


int size = 0;

for (int i = 0; i < MAX_CHAR; ++i) {

if (frequency[i] > 0)

size++;
```

```
}
char data[size];
unsigned freq[size];
int index = 0;
for (int i = 0; i < MAX_CHAR; ++i) {
if (frequency[i] > 0) {
data[index] = (char)i;
freq[index] = frequency[i];
index++;
}
}
HuffmanCodes(data, freq, size);
return 0;
}
```

Output:

Enter a sentence: three missionaries and three cannibals

: 000

i: 001

a: 010

d: 01100

m: 01101

b: 01110

o: 01111

n: 100

h: 1010

r: 1011

e: 110

c: 111000

l: 111001

t: 11101

s: 1111

d. LCS (Longest Common Subsequence)

Date:

The **Longest Common Subsequence (LCS)** algorithm finds the longest sequence of characters that appears in the same order in two given strings, allowing for gaps and mismatches. It uses dynamic programming to efficiently determine the length of the LCS and reconstruct the actual subsequence if needed.

Problem Statement:

Write a c program to implement LCS algorithm on the following strings :

X = CGATAATTGAGA

Y = GTTCCTAATA

Algorithm

Algorithm LCS(X,Y):
Input: Strings X and Y with n and m elements, respectively
Output: For $i = 0,\ldots,n-1$, $j = 0,\ldots,m-1$, the length $L[i,j]$ of a longest common subsequence of $X[0..i]$ and $Y[0..j]$

for $i \leftarrow -1$ to $n-1$ **do**
  $L[i,-1] \leftarrow 0$
for $j \leftarrow 0$ to $m-1$ **do**
  $L[-1,j] \leftarrow 0$
for $i \leftarrow 0$ to $n-1$ **do**
  for $j \leftarrow 0$ to $m-1$ **do**
    if $X[i] = Y[j]$ **then**
      $L[i,j] \leftarrow L[i-1,j-1]+1$
    else
      $L[i,j] \leftarrow \max\{L[i-1,j], L[i,j-1]\}$
return array L

Time complexity and Space complexity

The time complexity of the Longest Common Subsequence (LCS) algorithm is typically O(mn), where m and n are the lengths of the two input strings.

The space complexity of the LCS algorithm is O(mn) as well.

Code

```
#include <stdio.h>

#include <string.h>

#define MAX_LENGTH 100


int LCS(char *X, char *Y, char
*subsequence)

{

int n = strlen(X);

int m = strlen(Y);

int L[MAX_LENGTH][MAX_LENGTH];

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

for (int j = 0; j <= m; j++) {

if (i == 0 || j == 0) {

L[i][j] = 0;

} else if (X[i - 1] == Y[j - 1]) {

L[i][j] = L[i - 1][j - 1] + 1;

} else {

L[i][j] = (L[i - 1][j] > L[i][j - 1]) ? L[i -
1][j] : L[i][j - 1];

}

}

}

int index = L[n][m];

subsequence[index] = '\0';

int i = n, j = m;

while (i > 0 && j > 0) {

if (X[i - 1] == Y[j - 1]) {

subsequence[index - 1] = X[i - 1];

i--;

j--;

index--;

} else if (L[i - 1][j] >= L[i][j - 1]) {

i--;

} else {

j--;

}

}

return L[n][m];

}

int main() {

char X[MAX_LENGTH],
Y[MAX_LENGTH];

char subsequence[MAX_LENGTH];

printf("Enter string X: ");

scanf("%s", X);

printf("Enter string Y: ");

scanf("%s", Y);

int result = LCS(X, Y, subsequence);


printf("Length of the Longest Common
Subsequence: %d\n", result);

printf("Longest Common Subsequence:
%s\n", subsequence);

return 0;

}
```

Output

Enter string X: CGATAATTGAGA

Enter string Y: GTTCCTAATA

Length of the Longest Common Subsequence: 6

Longest Common Subsequence: CTAATA

CONCLUSION:

Internet Algorithms was studied. The programs for (a) KMP (Knuth-Morris-Pratt Algorithm) , (b) BM (Boyer-Moore Algorithm) (c) Huffman Encoding , and (d) LCS (Longest Common Subsequence) using internet algorithm were studied and implemented successfully.