**NAME : ROHAN NYATI**

**ROOL NO : R177219148**

**SAP ID : 500075940**

**BATCH : 5 (AI & ML)**

**Lab Records**

**ACTIVITY SELECTION PROBLEM**

**CODE**

#include<stdio.h>

void activities(int s[], int f[], int n)

{

int i, j;

printf ("Selected Activities are:\n");

i = 1;

printf("A%d ", i);

for (j = 1; j < n; j++)

{

if (s[j] >= f[i])

{

printf ("A%d ", j+1);

i = j;

}

}

}

int main()

{

int s[] = {1, 3, 0, 5, 3, 5, 6, 8, 8, 2, 12};

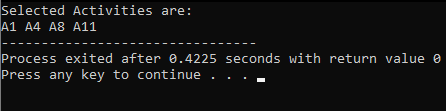
int f[] = {4, 5, 6, 7, 9, 9, 10, 11, 12, 14, 16};

int n = sizeof(s)/sizeof(s[0]);

activities(s, f, n);

}

**OUTPUT**

****

**MAX SUB ARRAY**

**CODE**

#include<stdio.h>

#define NEGINF -1000000000

#define MAX(a,b) a>b?a:b

int a[9]={-2,1,-3,4,-1,2,1,-5,4};

int max\_across(int low,int mid,int high){

int i;

int leftsum,rightsum,sum,maxsum;

leftsum=NEGINF;

sum=0;

for(i=mid;i>=low;i--){

sum=sum+a[i];

if(sum>leftsum)

leftsum=sum;

}

rightsum=NEGINF;

for(i=mid+1;i<=high;i++){

sum=sum+a[i];

if(sum>rightsum)

rightsum=sum;

}

maxsum=leftsum+rightsum;

return maxsum;

}

int max\_sum(int low,int high){

int left\_sum,right\_sum,across\_sum,final\_sum;

int mid=low+(high-low)/2;

if(low==high) return a[low];

left\_sum=max\_sum(low,mid);

right\_sum=max\_sum(mid+1,high);

across\_sum=max\_across(low,mid,high);

final\_sum=MAX(MAX(left\_sum,right\_sum),across\_sum);

return final\_sum;

}

int main(){

int sum;

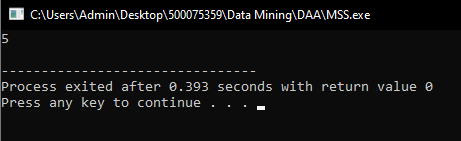
sum=max\_sum(0,9);

printf("%d\n",sum);

return 0;

}

**OUTPUT**



**Kruskal's algorithm**

**CODE**

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

int i,j,k,a,b,u,v,n,ne=1;

int min,mincost=0,cost[9][9],parent[9];

int uni(int,int);

int uni(int i,int j)

{

if(i!=j)

{

parent[j]=i;

return 1;

}

return 0;

}

void main()

{

printf("\n\tImplementation of Kruskal's algorithm\n");

printf("\nEnter the no. of vertices:");

scanf("%d",&n);

printf("\nEnter the cost adjacency matrix:\n");

for(i=1;i<=n;i++)

{

for(j=1;j<=n;j++)

{

scanf("%d",&cost[i][j]);

if(cost[i][j]==0)

cost[i][j]=999;

}

}

printf("The edges of Minimum Cost Spanning Tree are\n");

while(ne < n)

{

for(i=1,min=999;i<=n;i++)

{

for(j=1;j <= n;j++)

{

if(cost[i][j] < min)

{

min=cost[i][j];

a=u=i;

b=v=j;

}

}

}

if(uni(u,v))

{

printf("%d edge (%d,%d) =%d\n",ne++,a,b,min);

mincost +=min;

}

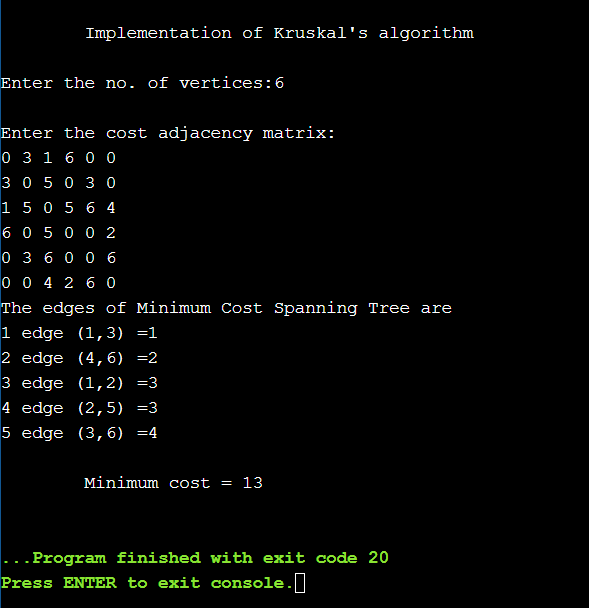
cost[a][b]=cost[b][a]=999;

}

printf("\n\tMinimum cost = %d\n",mincost);

}

**OUTPUT**

****

**PRIM’S ALGORITHM**

**CODE**

#include<stdio.h>

#include<conio.h>

int a,b,u,v,n,i,j,ne=1;

int visited[10]={0},min,mincost=0,cost[10][10];

void main()

{

printf("\nEnter the number of nodes:");

scanf("%d",&n);

printf("\nEnter the adjacency matrix:\n");

for(i=1;i<=n;i++)

for(j=1;j<=n;j++)

{

scanf("%d",&cost[i][j]);

if(cost[i][j]==0)

cost[i][j]=999;

}

visited[1]=1;

printf("\n");

while(ne < n)

{

for(i=1,min=999;i<=n;i++)

for(j=1;j<=n;j++)

if(cost[i][j]< min)

if(visited[i]!=0)

{

min=cost[i][j];

a=u=i;

b=v=j;

}

if(visited[u]==0 || visited[v]==0)

{

printf("\n Edge %d:(%d %d) cost:%d",ne++,a,b,min);

mincost+=min;

visited[b]=1;

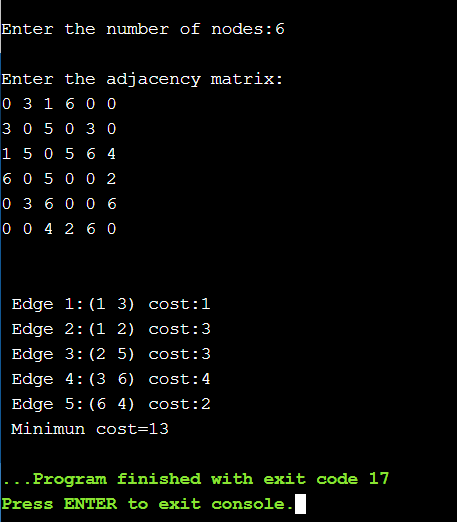
}

cost[a][b]=cost[b][a]=999;

}

printf("\n Minimun cost=%d",mincost);

}

**OUTPUT**

**HUFFMAN CODING**

**CODE**

// Huffman Coding in C

#include <stdio.h>

#include <stdlib.h>

#define MAX\_TREE\_HT 50

struct MinHNode {

char item;

unsigned freq;

struct MinHNode \*left, \*right;

};

struct MinHeap {

unsigned size;

unsigned capacity;

struct MinHNode \*\*array;

};

// Create nodes

struct MinHNode \*newNode(char item, unsigned freq) {

struct MinHNode \*temp = (struct MinHNode \*)malloc(sizeof(struct MinHNode));

temp->left = temp->right = NULL;

temp->item = item;

temp->freq = freq;

return temp;

}

// Create min heap

struct MinHeap \*createMinH(unsigned capacity) {

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

minHeap->size = 0;

minHeap->capacity = capacity;

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

return minHeap;

}

// Function to swap

void swapMinHNode(struct MinHNode \*\*a, struct MinHNode \*\*b) {

struct MinHNode \*t = \*a;

\*a = \*b;

\*b = t;

}

// Heapify

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]->freq < minHeap->array[smallest]->freq)

smallest = left;

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

smallest = right;

if (smallest != idx) {

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

minHeapify(minHeap, smallest);

}

}

// Check if size if 1

int checkSizeOne(struct MinHeap \*minHeap) {

return (minHeap->size == 1);

}

// Extract min

struct MinHNode \*extractMin(struct MinHeap \*minHeap) {

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

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

--minHeap->size;

minHeapify(minHeap, 0);

return temp;

}

// Insertion function

void insertMinHeap(struct MinHeap \*minHeap, struct MinHNode \*minHeapNode) {

++minHeap->size;

int i = minHeap->size - 1;

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

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 MinHNode \*root) {

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

}

struct MinHeap \*createAndBuildMinHeap(char item[], int freq[], int size) {

struct MinHeap \*minHeap = createMinH(size);

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

minHeap->array[i] = newNode(item[i], freq[i]);

minHeap->size = size;

buildMinHeap(minHeap);

return minHeap;

}

struct MinHNode \*buildHuffmanTree(char item[], int freq[], int size) {

struct MinHNode \*left, \*right, \*top;

struct MinHeap \*minHeap = createAndBuildMinHeap(item, freq, size);

while (!checkSizeOne(minHeap)) {

left = extractMin(minHeap);

right = extractMin(minHeap);

top = newNode('$', left->freq + right->freq);

top->left = left;

top->right = right;

insertMinHeap(minHeap, top);

}

return extractMin(minHeap);

}

void printHCodes(struct MinHNode \*root, int arr[], int top) {

if (root->left) {

arr[top] = 0;

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

}

if (root->right) {

arr[top] = 1;

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

}

if (isLeaf(root)) {

printf(" %c | ", root->item);

printArray(arr, top);

}

}

// Wrapper function

void HuffmanCodes(char item[], int freq[], int size) {

struct MinHNode \*root = buildHuffmanTree(item, freq, size);

int arr[MAX\_TREE\_HT], top = 0;

printHCodes(root, arr, top);

}

// Print the array

void printArray(int arr[], int n) {

int i;

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

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

printf("\n");

}

int main() {

char arr[] = {'A', 'B', 'C', 'D'};

int freq[] = {5, 1, 6, 3};

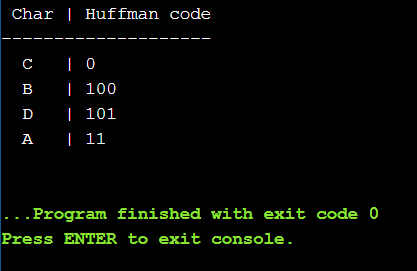
int size = sizeof(arr) / sizeof(arr[0]);

printf(" Char | Huffman code ");

printf("\n--------------------\n");

HuffmanCodes(arr, freq, size);

}

**OUTPUT**

**DJIKSTRA’S ALGORITHM**

**CODE**

#include<iostream>

using namespace std;

#define INFINITY 9999

#define max 10

void dijkstra(int G[max][max],int n,int startnode);

int main() {

int G[max][max]={{0,8,0,1,4,0},{0,0,0,0,0,5},{0,8,0,4,0,0},{0,2,9,0,0,0},{0,0,0,1,0,0},{0,0,0,0,0,0}};

int n=6; //number of vertices

int u=0; //source vertice

dijkstra(G,n,u);

return 0;

}

void dijkstra(int G[max][max],int n,int startnode) {

int cost[max][max],distance[max],pred[max];

int visited[max],count,mindistance,nextnode,i,j;

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

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

if(G[i][j]==0) //no edge in btw vertices

cost[i][j]=INFINITY; //cost will be infinity

else

cost[i][j]=G[i][j];

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

distance[i]=cost[startnode][i];

pred[i]=startnode;

visited[i]=0; //once visited,then it is initialized zero

}

distance[startnode]=0;

visited[startnode]=1; //if vertex is visited[0],is marked 1

count=1;

while(count<n-1)

{

mindistance=INFINITY;

for(i=0;i<n;i++) //next node gives the node at min distance

if(distance[i]<mindistance&&!visited[i]) {

mindistance=distance[i];

nextnode=i;

}

visited[nextnode]=1; //we check if there is any other better path through nextnode

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

if(!visited[i])

if(mindistance+cost[nextnode][i]<distance[i]) {

distance[i]=mindistance+cost[nextnode][i];

pred[i]=nextnode;

}

count++;

}

for(i=0;i<n;i++) //printing path and distance of each node

if(i!=startnode) {

cout<<"\nDistance of node"<<i<<"="<<distance[i];

cout<<"\nPath="<<i;

j=i;

do {

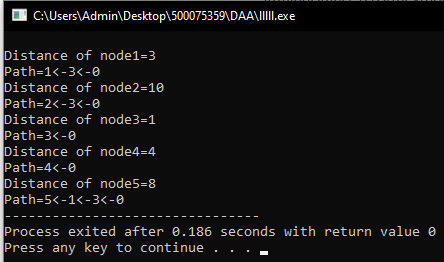
j=pred[j];

cout<<"<-"<<j;

}while(j!=startnode);

}

}

**OUTPUT  
**

**GRAPH IMPLEMENTATION**

**CODE**

#include <stdio.h>

#include <stdlib.h>

struct AdjListNode

{

int dest;

struct AdjListNode\* next;

};

struct AdjList

{

struct AdjListNode \*head;

};

struct Graph

{

int V;

struct AdjList\* array;

};

struct AdjListNode\* newAdjListNode(int dest)

{

struct AdjListNode\* newNode =

(struct AdjListNode\*) malloc(sizeof(struct AdjListNode));

newNode->dest = dest;

newNode->next = NULL;

return newNode;

}

struct Graph\* createGraph(int V)

{

struct Graph\* graph =

(struct Graph\*) malloc(sizeof(struct Graph));

graph->V = V;

graph->array =

(struct AdjList\*) malloc(V \* sizeof(struct AdjList));

int i;

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

graph->array[i].head = NULL;

return graph;

}

void addEdge(struct Graph\* graph, int src, int dest)

{

struct AdjListNode\* newNode = newAdjListNode(dest);

newNode->next = graph->array[src].head;

graph->array[src].head = newNode;

newNode = newAdjListNode(src);

newNode->next = graph->array[dest].head;

graph->array[dest].head = newNode;

}

void printGraph(struct Graph\* graph)

{

int v;

for (v = 0; v < graph->V; ++v)

{

struct AdjListNode\* pCrawl = graph->array[v].head;

printf("\n Adjacency list of vertex %d\n head ", v);

while (pCrawl)

{

printf("-> %d", pCrawl->dest);

pCrawl = pCrawl->next;

}

printf("\n");

}

}

int main()

{

int V = 5;

struct Graph\* graph = createGraph(V);

addEdge(graph, 0, 1);

addEdge(graph, 0, 4);

addEdge(graph, 1, 2);

addEdge(graph, 1, 3);

addEdge(graph, 1, 4);

addEdge(graph, 2, 3);

addEdge(graph, 3, 4);

printGraph(graph);

return 0;

}

**OUTPUT**

