**OPERATING SYSTEM ASSIGNMENT - 5**

SUBMITTED BY,

SOORYA HARSHA P

7176 22 51 035

I MCA

1. **THE DINING PHILOSOPHERS PROBLEM USING SEMAPHORE**

do { wait(chopstick[i]); wait(chopstick[(i+1)%5]);

EATING THE RICE

Signal(chopstick[i]); signal(chopstick[(i+1)%5]);

THINKING

}while(1);

**USING MONITOR:**

monitor DiningPhilosophers

{

status state[5]; condition self[5];

void Pick(int i)

{

state[i] = HUNGRY;

if (state[i] != EATING)

self[i].wait;

}

void Put(int i)

{

state[i] = THINKING;

test((i + 1) % 5);

test((i + 4) % 5);

}

void test(int i)

{

if (state[(i + 1) % 5] != EATING && state[(i + 4) % 5] != EATING && state[i] ==

HUNGRY)

{

state[i] = EATING;

philosophers during Put()

self[i].signal();

}

}

initialization\_code()

{

for (int i = 0; i < 5; i++)

state[i] = THINKING;

}

}

1. **THE BARBER PROBLEM USING SEMAPHORE:**

Semaphore Customers = 0;

Semaphore Barber = 0; Mutex Seats = 1; int FreeSeats = N;

Barber { while(true) {

down(Customers);

down(Seats);

FreeSeats++;

up(Barber);

up(Seats);

}

}

Customer {

while(true) {

down(Seats);

if(FreeSeats > 0) {

FreeSeats--;

up(Customers);

up(Seats);

down(Barber);

} else {

up(Seats);

}

}

}

**3.) READERS AND WRITERS USING SEMAPHORE:**

while (true) { wait(rw mutex);

...

/\* writing is performed \*/

...

signal(rw mutex); }while (true) { wait(mutex); read count++; if (read count == 1) wait(rw mutex); signal(mutex);

...

/\* reading is performed \*/

wait(mutex); read count--; if (read count == 0) signal(rw mutex); signal(mutex);

}

**USING MONITOR:**

monitor ReadersWriters condition OKtoWrite, OKtoRead;

int ReaderCount = 0; Boolean busy = false;

procedure StartRead()

{

if (busy)

OKtoRead.wait;

ReaderCount++;

OKtoRead.signal();

}

procedure EndRead()

{

ReaderCount-- ;

if ( ReaderCount == 0 )

OKtoWrite.signal();

}

procedure StartWrite()

{

if ( busy || ReaderCount != 0 )

OKtoWrite.wait();

busy = true;

}

procedure EndWrite()

{

busy = false;

If (OKtoRead.Queue) OKtoRead.signal(); else

OKtoWrite.signal();

}

Reader()

{

while (TRUE)

{

ReadersWriters.StartRead();

readDatabase();

ReadersWriters.EndRead();

}

}

Writer()

{

while (TRUE)

{

make\_data(&info);

ReaderWriters.StartWrite();

writeDatabase();

ReadersWriters.EndWrite();

}

**CODING :**

#include<stdio.h>

#include<stdlib.h>

struct node

{

int vertex;

struct node \*next;

};

int n,i,j;

void create(struct node \*list[])

{

int k;struct node \*newnode;

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

{

printf("Enter no. of vertices adjacent to %d: ",i);

scanf("%d",&k);

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

{ struct node \*last;

newnode=(struct node\*)malloc(sizeof(struct node));

printf("Enter the %d vertex adjacent to %d: ",j,i);

scanf("%d",&newnode->vertex);

newnode->next=NULL;

if(list[i]==NULL)

{

list[i]=newnode;

}

else

{

last->next=newnode;

}

last=newnode;

}

}

}

void print(struct node \*list[])

{

struct node \*temp;

int i;

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

{

printf("\n");

temp=list[i];

printf("Vertices adjacent to %d :",i);

while(temp!=0)

{

printf("%d ,",temp->vertex);

temp=temp->next;

}

}

}

void main()

{

int a[20][20];

printf("Enter the no. of nodes: ");

scanf("%d",&n);

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

{

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

{

printf("Enter 1 if %d and %d is adjacent else enter 0: ",i,j);

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

}

printf("\n");

}

printf("ADJACENCY MATRIX: \n");

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

{

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

{

printf("%d ",a[i][j]);

}

printf("\n");

}

printf("\n");

struct node \*list[n];

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

{

list[i]=NULL;

}

create(list);

printf("\nADJACENCY LIST: ");

print(list);

}

**OUTPUT :**

Enter the no. of nodes: 4

Enter 1 if 1 and 1 is adjacent else enter 0: 0

Enter 1 if 1 and 2 is adjacent else enter 0: 1

Enter 1 if 1 and 3 is adjacent else enter 0: 0

Enter 1 if 1 and 4 is adjacent else enter 0: 0

Enter 1 if 2 and 1 is adjacent else enter 0: 1

Enter 1 if 2 and 2 is adjacent else enter 0: 0

Enter 1 if 2 and 3 is adjacent else enter 0: 1

Enter 1 if 2 and 4 is adjacent else enter 0: 0

Enter 1 if 3 and 1 is adjacent else enter 0: 0

Enter 1 if 3 and 2 is adjacent else enter 0: 1

Enter 1 if 3 and 3 is adjacent else enter 0: 0

Enter 1 if 3 and 4 is adjacent else enter 0: 1

Enter 1 if 4 and 1 is adjacent else enter 0: 1

Enter 1 if 4 and 2 is adjacent else enter 0: 0

Enter 1 if 4 and 3 is adjacent else enter 0: 0

Enter 1 if 4 and 4 is adjacent else enter 0: 0

ADJACENCY MATRIX:

0 1 0 0

1 0 1 0

0 1 0 1

1 0 0 0

Enter no. of vertices adjacent to 1: 3

Enter the 1 vertex adjacent to 1: 2

Enter the 2 vertex adjacent to 1: 3

Enter the 3 vertex adjacent to 1: 4

Enter no. of vertices adjacent to 2: 1

Enter the 1 vertex adjacent to 2: 1

Enter no. of vertices adjacent to 3: 2

Enter the 1 vertex adjacent to 3: 4

Enter the 2 vertex adjacent to 3: 1

Enter no. of vertices adjacent to 4: 1

Enter the 1 vertex adjacent to 4: 1

ADJACENCY LIST:

Vertices adjacent to 1 :2 ,3 ,4 ,

Vertices adjacent to 2 :1 ,

Vertices adjacent to 3 :4 ,1 ,

Vertices adjacent to 4 :1 ,

**CODING :**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_VERTICES 100

struct Graph {

int V;

int adj\_mat[MAX\_VERTICES][MAX\_VERTICES];

};

struct Queue {

int arr[MAX\_VERTICES];

int front, rear;

};

int i,j;

struct Graph\* createGraph(int V) {

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

graph->V = V;

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

for ( j = 0; j < V; j++) {

graph->adj\_mat[i][j] = 0;

}

}

return graph;

}

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

graph->adj\_mat[src][dest] = 1;

graph->adj\_mat[dest][src] = 1;

}

bool isQueueEmpty(struct Queue\* q) {

return (q->front > q->rear);

}

void enqueue(struct Queue\* q, int item) {

q->arr[++q->rear] = item;

}

int dequeue(struct Queue\* q) {

return q->arr[q->front++];

}

void BFS(struct Graph\* graph, int start) {

bool visited[graph->V];

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

visited[i] = false;

}

struct Queue\* q = (struct Queue\*) malloc(sizeof(struct Queue));

q->front = 0;

q->rear = -1;

visited[start] = true;

enqueue(q, start);

while (!isQueueEmpty(q)) {

int curr = dequeue(q);

printf("%d ", curr);

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

if (graph->adj\_mat[curr][i] && !visited[i]) {

visited[i] = true;

enqueue(q, i);

}

}

}

printf("\n");

}

void DFSUtil(struct Graph\* graph, int v, bool visited[]) {

visited[v] = true;

printf("%d ", v);

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

if (graph->adj\_mat[v][i] && !visited[i]) {

DFSUtil(graph, i, visited);

}

}

}

void DFS(struct Graph\* graph, int start) {

bool visited[graph->V];

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

visited[i] = false;

}

DFSUtil(graph, start, visited);

printf("\n");

}

int main() {

int V = 4;

int value;

struct Graph\* graph = createGraph(V);

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 2);

addEdge(graph, 2, 0);

addEdge(graph, 2, 3);

printf("Enter the value you want to search: ");

scanf("%d", &value);

printf("BFS starting from vertex %d: ", value);

BFS(graph, value);

printf("DFS starting from vertex %d: ", value);

DFS(graph, value);

return 0;

}

**OUTPUT :**

Enter the value you want to search: 3

BFS starting from vertex 3: 3 2 0 1

DFS starting from vertex 3: 3 2 0 1

**CODING :**

#include <stdio.h>

#include <stdbool.h>

#include <string.h>

#define V 6

void printPath(int path[], int len) {

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

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

}

printf("\n");

}

void findAllPaths(int adjMatrix[V][V], int start, int end, bool visited[], int path[], int len) {

visited[start] = true;

path[len] = start;

len++;

if (start == end) {

printPath(path, len);

} else {

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

if (adjMatrix[start][i] && !visited[i]) {

findAllPaths(adjMatrix, i, end, visited, path, len);

}

}

}

// backtrack

visited[start] = false;

len--;

}

void findAllPathsBetweenNodes(int adjMatrix[V][V], int start, int end) {

bool visited[V];

memset(visited, false, sizeof(visited));

int path[V];

findAllPaths(adjMatrix, start, end, visited, path, 0);

}

int main() {

int adjMatrix[V][V] = {

{0, 1, 1, 0, 0, 0},

{0, 0, 0, 1, 1, 0},

{0, 0, 0, 0, 0, 1},

{0, 0, 0, 0, 0, 1},

{0, 0, 0, 0, 0, 1},

{0, 0, 0, 0, 0, 0}

};

int start = 0;

int end = 5;

printf("All paths from node %d to node %d: \n", start, end);

findAllPathsBetweenNodes(adjMatrix, start, end);

return 0;

}

**OUTPUT :**

0 1 3 5

**CODING :**

#include <stdlib.h>

struct AjlistNode

{

int id;

struct AjlistNode \*next;

};

struct Vertices

{

int data;

struct AjlistNode \*next;

struct AjlistNode \*last;

};

struct Graph

{

int size;

struct Vertices \*node;

};

void setData(struct Graph \*g)

{

if (g->node != NULL)

{

int index = 0;

for (int index = 0; index < g->size; index++)

{

g->node[index].data = index;

g->node[index].next = NULL;

g->node[index].next = NULL;

}

}

else

{

printf("Vertic Node is Empty");

}

}

struct Graph \*newGraph(int size)

{

if (size < 1)

{

printf("\n Invalid graph size ");

exit(0);

}

struct Graph \*g = (struct Graph \*)

malloc(sizeof(struct Graph));

g->size = size;

g->node = (struct Vertices \*)

malloc(sizeof(struct Vertices) \*size);

setData(g);

return g;

}

void connect(struct Graph \*g, int start, int last)

{

struct AjlistNode \*edge = (struct AjlistNode \*)

malloc(sizeof(struct AjlistNode));

if (edge != NULL)

{

edge->id = last;

edge->next = NULL;

if (g->node[start].next == NULL)

{

g->node[start].next = edge;

}

else

{

g->node[start].last->next = edge;

}

g->node[start].last = edge;

}

else

{

printf("\n Memory overflow to create edge");

}

}

void addEdge(struct Graph \*g, int start, int last)

{

if (start < g->size && last < g->size)

{

connect(g, start, last);

}

else

{

printf("Invalid Node Vertices %d %d", start, last);

}

}

void printGraph(struct Graph \*g)

{

if (g != NULL)

{

struct AjlistNode \*temp = NULL;

for (int index = 0; index < g->size; index++)

{

printf("\n Adjacency list of vertex %d :", index);

temp = g->node[index].next;

while (temp != NULL)

{

printf(" %d", g->node[temp->id].data);

temp = temp->next;

}

}

}

else

{

printf("Empty Graph");

}

}

int detectCycle(struct Graph \*g,

int start, int point, int visit[])

{

if (visit[start] == 1 && point == start)

{

return 1;

}

else if (visit[point] == 1)

{

return 0;

}

visit[point] = 1;

struct AjlistNode \*temp = g->node[point].next;

int status = 0;

while (temp != NULL && !status)

{

status = detectCycle(g, start, temp->id, visit);

temp = temp->next;

}

return status;

}

void checkCycle(struct Graph \*g)

{

if (g->size <= 0)

{

printf("Empty Graph\n");

return;

}

printGraph(g);

int visit[g->size];

int result = 0;

int location = 0;

while (location < g->size && result == 0)

{

for (int index = 0; index < g->size; index++)

{

visit[index] = 0;

}

result = detectCycle(g, location, location, visit);

location++;

}

if (result == 1)

{

printf("\n Result : Detect Cycle\n");

}

else

{

printf("\n Result : No Cycle\n");

}

}

int main()

{

struct Graph \*g = newGraph(6);

addEdge(g, 0, 1);

addEdge(g, 0, 3);

addEdge(g, 1, 2);

addEdge(g, 2, 5);

addEdge(g, 3, 4);

addEdge(g, 4, 2);

// Test A

checkCycle(g);

addEdge(g, 5, 1);

// Test B

checkCycle(g);

return 0;

}

**OUTPUT :**

Adjacency list of vertex 0 : 1 3

Adjacency list of vertex 1 : 2

Adjacency list of vertex 2 : 5

Adjacency list of vertex 3 : 4

Adjacency list of vertex 4 : 2

Adjacency list of vertex 5 :

Result : No Cycle

Adjacency list of vertex 0 : 1 3

Adjacency list of vertex 1 : 2

Adjacency list of vertex 2 : 5

Adjacency list of vertex 3 : 4

Adjacency list of vertex 4 : 2

Adjacency list of vertex 5 : 1

Result : Detect Cycle

**CODING :**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_EDGES 100

#define MAX\_VERTICES 100

#define INF 0x3f3f3f3f

int parent[MAX\_VERTICES];

struct Edge {

int src, dest, weight;

};

struct Graph {

int V, E;

struct Edge\* edge;

};

struct Graph\* createGraph(int V, int E) {

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

graph->V = V;

graph->E = E;

graph->edge = (struct Edge\*) malloc(graph->E \* sizeof(struct Edge));

return graph;

}

int find(int x) {

if (parent[x] == x)

return x;

return find(parent[x]);

}

int cmp(const void\* a, const void\* b) {

struct Edge\* a1 = (struct Edge\*) a;

struct Edge\* b1 = (struct Edge\*) b;

return a1->weight > b1->weight;

}

void KruskalMST(struct Graph\* graph) {

int V = graph->V;

int E = graph->E;

struct Edge result[V];

int e = 0, i = 0;

for (int v = 0; v < V; v++)

parent[v] = v;

qsort(graph->edge, E, sizeof(graph->edge[0]), cmp);

while (e < V - 1 && i < E) {

struct Edge next\_edge = graph->edge[i++];

int x = find(next\_edge.src);

int y = find(next\_edge.dest);

if (x != y) {

result[e++] = next\_edge;

parent[x] = y;

}

}

int total\_weight = 0;

printf("Following are the edges in the constructed MST\n");

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

printf("%d -- %d == %d\n", result[i].src, result[i].dest, result[i].weight);

total\_weight += result[i].weight;

}

printf("Total weight of MST is %d\n", total\_weight);

}

void primMST(struct Graph\* graph) {

int V = graph->V;

int E = graph->E;

int dist[V];

int visited[V];

int min\_index;

int total\_weight = 0;

for (int v = 0; v < V; v++) {

dist[v] = INF;

visited[v] = 0;

}

dist[0] = 0;

parent[0] = -1;

for (int count = 0; count < V - 1; count++) {

min\_index = 0;

int min = INF;

for (int v = 0; v < V; v++) {

if (visited[v] == 0 && dist[v] < min) {

min = dist[v];

min\_index = v;

}

}

visited[min\_index] = 1;

total\_weight += dist[min\_index];

for (int v = 0; v < V; v++) {

for (int e = 0; e < E; e++) {

if (graph->edge[e].src == min\_index) {

if (visited[graph->edge[e].dest] == 0 && graph->edge[e].weight < dist[graph->edge[e].dest]) {

dist[graph->edge[e].dest] = graph->edge[e].weight;

parent[graph->edge[e].dest] = min\_index;

}

}

}

}

}

printf("Following are the edges in the constructed MST\n");

for (int i = 1; i < V; i++) {

printf("%d - %d == %d\n", parent[i], i, dist[i]);

}

printf("Total weight of MST is %d\n", total\_weight);

}

int main() {

int V = 4, E = 5;

struct Graph\* graph = createGraph(V, E);

graph->edge[0].src = 0;

graph->edge[0].dest = 1;

graph->edge[0].weight = 10;

graph->edge[1].src = 0;

graph->edge[1].dest = 2;

graph->edge[1].weight = 6;

graph->edge[2].src = 0;

graph->edge[2].dest = 3;

graph->edge[2].weight = 5;

graph->edge[3].src = 1;

graph->edge[3].dest = 3;

graph->edge[3].weight = 15;

graph->edge[4].src = 2;

graph->edge[4].dest = 3;

graph->edge[4].weight = 4;

printf("Kruskal's MST:\n");

KruskalMST(graph);

printf("Prim's MST:\n");

primMST(graph);

return 0;

}

**OUTPUT :**

Kruskal's MST:

Following are the edges in the constructed MST

2 -- 3 == 4

0 -- 3 == 5

0 -- 1 == 10

Total weight of MST is 19

Prim's MST:

Following are the edges in the constructed MST

0 - 1 == 10

0 - 2 == 6

0 - 3 == 5

Total weight of MST is 11