**LAB CYCLE 4**

**Submitted by**

**SIJI JOSE**

**MCA 135**

**PROGRAM 1: DFS**

#include <stdio.h>

#include <stdlib.h>

struct node {

int vertex;

struct node\* next;

};

struct node\* createNode(int v);

struct Graph {

int numVertices;

int\* visited;

// We need int\*\* to store a two dimensional array.

// Similary, we need struct node\*\* to store an array of Linked lists

struct node\*\* adjLists;

};

// DFS algo

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

struct node\* adjList = graph->adjLists[vertex];

struct node\* temp = adjList;

graph->visited[vertex] = 1;

printf("Visited %d \n", vertex);

while (temp != NULL) {

int connectedVertex = temp->vertex;

if (graph->visited[connectedVertex] == 0) {

DFS(graph, connectedVertex);

}

temp = temp->next;

}}

// Create a node

struct node\* createNode(int v) {

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

newNode->vertex = v;

newNode->next = NULL;

return newNode;

}

// Create graph

struct Graph\* createGraph(int vertices) {

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

graph->numVertices = vertices;

graph->adjLists = malloc(vertices \* sizeof(struct node\*));

graph->visited = malloc(vertices \* sizeof(int));

int i;

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

graph->adjLists[i] = NULL;

graph->visited[i] = 0;

}

return graph;

}

// Add edge

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

// Add edge from src to dest

struct node\* newNode = createNode(dest);

newNode->next = graph->adjLists[src];

graph->adjLists[src] = newNode;

// Add edge from dest to src

newNode = createNode(src);

newNode->next = graph->adjLists[dest];

graph->adjLists[dest] = newNode;

}

// Print the graph

void printGraph(struct Graph\* graph) {

int v;

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

struct node\* temp = graph->adjLists[v];

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

while (temp) {

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

temp = temp->next;

}

printf("\n");

}}

int main() {

struct Graph\* graph = createGraph(4);

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 2);

addEdge(graph, 2, 3);

printGraph(graph);

DFS(graph, 2);

return 0;

}

**PROGRAM 2 : BFS**

#include <stdio.h>

#include <stdlib.h>

#define SIZE 40

struct queue {

int items[SIZE];

int front;

int rear;

};

struct queue\* createQueue();

void enqueue(struct queue\* q, int);

int dequeue(struct queue\* q);

void display(struct queue\* q);

int isEmpty(struct queue\* q);

void printQueue(struct queue\* q);

struct node {

int vertex;

struct node\* next;

};

struct node\* createNode(int);

struct Graph {

int numVertices;

struct node\*\* adjLists;

int\* visited;

};

// BFS algorithm

void bfs(struct Graph\* graph, int startVertex) {

struct queue\* q = createQueue();

graph->visited[startVertex] = 1;

enqueue(q, startVertex);

while (!isEmpty(q)) {

printQueue(q);

int currentVertex = dequeue(q);

printf("Visited %d\n", currentVertex);

struct node\* temp = graph->adjLists[currentVertex];

while (temp) {

int adjVertex = temp->vertex;

if (graph->visited[adjVertex] == 0) {

graph->visited[adjVertex] = 1;

enqueue(q, adjVertex);

}

temp = temp->next;

}

}

}

// Creating a node

struct node\* createNode(int v) {

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

newNode->vertex = v;

newNode->next = NULL;

return newNode;

}

// Creating a graph

struct Graph\* createGraph(int vertices) {

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

graph->numVertices = vertices;

graph->adjLists = malloc(vertices \* sizeof(struct node\*));

graph->visited = malloc(vertices \* sizeof(int));

int i;

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

graph->adjLists[i] = NULL;

graph->visited[i] = 0;

}

return graph;

}

// Add edge

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

// Add edge from src to dest

struct node\* newNode = createNode(dest);

newNode->next = graph->adjLists[src];

graph->adjLists[src] = newNode;

// Add edge from dest to src

newNode = createNode(src);

newNode->next = graph->adjLists[dest];

graph->adjLists[dest] = newNode;

}

// Create a queue

struct queue\* createQueue() {

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

q->front = -1;

q->rear = -1;

return q;

}

// Check if the queue is empty

int isEmpty(struct queue\* q) {

if (q->rear == -1)

return 1;

else

return 0;

}

// Adding elements into queue

void enqueue(struct queue\* q, int value) {

if (q->rear == SIZE - 1)

printf("\nQueue is Full!!");

else {

if (q->front == -1)

q->front = 0;

q->rear++;

q->items[q->rear] = value;

}

}

// Removing elements from queue

int dequeue(struct queue\* q) {

int item;

if (isEmpty(q)) {

printf("Queue is empty");

item = -1;

} else {

item = q->items[q->front];

q->front++;

if (q->front > q->rear) {

printf("Resetting queue ");

q->front = q->rear = -1;

}

}

return item;

}

// Print the queue

void printQueue(struct queue\* q) {

int i = q->front;

if (isEmpty(q)) {

printf("Queue is empty");

} else {

printf("\nQueue contains \n");

for (i = q->front; i < q->rear + 1; i++) {

printf("%d ", q->items[i]);

}

}

}

int main() {

struct Graph\* graph = createGraph(6);

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 2);

addEdge(graph, 1, 4);

addEdge(graph, 1, 3);

addEdge(graph, 2, 4);

addEdge(graph, 3, 4);

bfs(graph, 0);

return 0;

}

**PROGRAM 3 : TOPOLOGICAL SORTING**

#include<stdio.h>

#include<stdlib.h>

#define MAX 100

int n; /\*Number of vertices in the graph\*/

int adj[MAX][MAX]; /\*Adjacency Matrix\*/

void create\_graph();

int queue[MAX], front = -1,rear = -1;

void insert\_queue(int v);

int delete\_queue();

int isEmpty\_queue();

int indegree(int v);

int main()

{

int i,v,count,topo\_order[MAX],indeg[MAX];

create\_graph();

/\*Find the indegree of each vertex\*/

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

{

indeg[i] = indegree(i);

if( indeg[i] == 0 )

insert\_queue(i);

}

count = 0;

while( !isEmpty\_queue( ) && count < n )

{

v = delete\_queue();

topo\_order[++count] = v; /\*Add vertex v to topo\_order array\*/

/\*Delete all edges going fron vertex v \*/

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

{

if(adj[v][i] == 1)

{

adj[v][i] = 0;

indeg[i] = indeg[i]-1;

if(indeg[i] == 0)

insert\_queue(i);

}

}

}

if( count < n )

{

printf("\nNo topological ordering possible, graph contains cycle\n");

exit(1);

}

printf("\nVertices in topological order are :\n");

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

printf( "%d ",topo\_order[i] );

printf("\n");

return 0;

}/\*End of main()\*/

void insert\_queue(int vertex)

{

if (rear == MAX-1)

printf("\nQueue Overflow\n");

else

{

if (front == -1) /\*If queue is initially empty \*/

front = 0;

rear = rear+1;

queue[rear] = vertex ;

}

}/\*End of insert\_queue()\*/

int isEmpty\_queue()

{

if(front == -1 || front > rear )

return 1;

else

return 0;

}/\*End of isEmpty\_queue()\*/

int delete\_queue()

{

int del\_item;

if (front == -1 || front > rear)

{

printf("\nQueue Underflow\n");

exit(1);

}

else

{

del\_item = queue[front];

front = front+1;

return del\_item;

}

}/\*End of delete\_queue() \*/

int indegree(int v)

{

int i,in\_deg = 0;

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

if(adj[i][v] == 1)

in\_deg++;

return in\_deg;

}/\*End of indegree() \*/

void create\_graph()

{

int i,max\_edges,origin,destin;

printf("\nEnter number of vertices : ");

scanf("%d",&n);

max\_edges = n\*(n-1);

for(i=1; i<=max\_edges; i++)

{

printf("\nEnter edge %d(-1 -1 to quit): ",i);

scanf("%d %d",&origin,&destin);

if((origin == -1) && (destin == -1))

break;

if( origin >= n || destin >= n || origin<0 || destin<0)

{

printf("\nInvalid edge!\n");

i--;

}

else

adj[origin][destin] = 1;

}}

**PROGRAM 14 : STRONGLY CONNECTED COMPONENT**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_DEGREE 5

#define MAX\_NUM\_VERTICES 20

struct vertices\_s {

int visited;

int deg;

int adj[MAX\_DEGREE]; /\* < 0 if incoming edge \*/

} vertices[] = {

{0, 3, {2, -3, 4}},

{0, 2, {-1, 3}},

{0, 3, {1, -2, 7}},

{0, 3, {-1, -5, 6}},

{0, 2, {4, -7}},

{0, 3, {-4, 7, -8}},

{0, 4, {-3, 5, -6, -12}},

{0, 3, {6, -9, 11}},

{0, 2, {8, -10}},

{0, 3, {9, -11, -12}},

{0, 3, {-8, 10, 12}},

{0, 3, {7, 10, -11}}

};

int num\_vertices = sizeof(vertices) / sizeof(vertices[0]);

struct stack\_s {

int top;

int items[MAX\_NUM\_VERTICES];

} stack = {-1, {}};

void stack\_push(int v) {

stack.top++;

if (stack.top < MAX\_NUM\_VERTICES)

stack.items[stack.top] = v;

else {

printf("Stack is full!\n");

exit(1);

}}

int stack\_pop() {

return stack.top < 0 ? -1 : stack.items[stack.top--];

}

void dfs(int v, int transpose) {

int i, c, n;

vertices[v].visited = 1;

for (i = 0, c = vertices[v].deg; i < c; ++i) {

n = vertices[v].adj[i] \* transpose;

if (n > 0)

/\* n - 1 because vertex indexing begins at 0 \*/

if (!vertices[n - 1].visited)

dfs(n - 1, transpose);

}

if (transpose < 0)

stack\_push(v);

else

printf("%d ", v + 1);

}

void reset\_visited() {

int i;

for (i = 0; i < num\_vertices; ++i)

vertices[i].visited = 0;

}

void order\_pass() {

int i;

for (i = 0; i < num\_vertices; ++i)

if (!vertices[i].visited)

dfs(i, -1);

}

void scc\_pass() {

int i = 0, v;

while((v = stack\_pop()) != -1) {

if (!vertices[v].visited) {

printf("scc %d: ", ++i);

dfs(v, 1);

printf("\n");

} }}

int main(void) {

order\_pass();

reset\_visited();

scc\_pass();

return 0;

}