21)

#include <stdio.h>

int N, M;

void createAdjMatrix(int Adj[][N + 1],

int arr[][2])

{

for (int i = 0; i < N + 1; i++) {

for (int j = 0; j < N + 1; j++) {

Adj[i][j] = 0;

}

}

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

int x = arr[i][0];

int y = arr[i][1];

Adj[x][y] = 1;

Adj[y][x] = 1;

}

}

void printAdjMatrix(int Adj[][N + 1])

{

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

for (int j = 1; j < N + 1; j++) {

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

}

printf("\n");

}

}

void printAdjList(int Adj[][N + 1])

{

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

printf("Adjacency List for vertex %d: ", i);

for (int j = 1; j < N + 1; j++) {

if (Adj[i][j] == 1) {

printf("%d ", j);

}

}

printf("\n");

}

}

int main()

{

N = 5;

int arr[][2]

= { { 1, 2 }, { 2, 3 },

{ 4, 5 }, { 1, 5 } };

M = sizeof(arr) / sizeof(arr[0]);

int Adj[N + 1][N + 1];

createAdjMatrix(Adj, arr);

printf("Adjacency Matrix:\n");

printAdjMatrix(Adj);

printf("\nAdjacency List:\n");

printAdjList(Adj);

return 0;

}

22) #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;

};

struct Graph\* createGraph(int V) {

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

graph->V = V;

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

for (int 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 (int 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 (int 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 (int 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 (int 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;

}

23) #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);

}

}

}

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;

}

24)

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

const int V=4;

bool isCyclicUtil(int v, bool visited[], bool \*recStack, int adjMatrix[V][V]) {

if(visited[v] == false) {

visited[v] = true;

recStack[v] = true;

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

if (adjMatrix[v][i] && isCyclicUtil(i, visited, recStack, adjMatrix))

return true;

else if (recStack[i])

return true;

}

}

recStack[v] = false;

return false;

}

bool isCyclic(int V, int adjMatrix[V][V]) {

bool \*visited = calloc(V, sizeof(bool));

bool \*recStack = calloc(V, sizeof(bool));

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

if (isCyclicUtil(i, visited, recStack, adjMatrix))

return true;

return false;

}

int main() {

int adjMatrix[4][4] = {

{0, 1, 0, 1},

{1, 0, 1, 0},

{0, 1, 0, 1},

{1, 0, 1, 0}

};

if(isCyclic(V, adjMatrix))

printf("Graph contains a cycle\n");

else

printf("Graph doesn't contain a cycle\n");

return 0;

}

25)

#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;

}