

Roll – 2023UGCS066

Name – Alisha Khatoon

Assignment 11: Graphs

1. Write a program that performs a breadth-first traversal of a given graph. Ensure that your program can handle graphs with cycles, identifying and marking any cycles encountered during traversal.
2. Write a program that performs a depth-first traversal of a given graph. Additionally, implement functionality to detect and highlight back edges to verify if the graph contains cycles.

⇒

```
#include "queue.c"
#include <stdio.h>
#include <stdlib.h>
```

```
void bfs(int start, int **adj, int *vis, int *adjCount) {
    struct Queue q;
    q.FRONT = q.REAR = NULL;
    q.size = 0;
    Enqueue(&q, start);
    vis[start] = 1;
    while (!isEmpty(&q)) {
        int node = Dequeue(&q)->data;
        printf("%d ", node);
        for (int i = 0; i < adjCount[node]; i++) {
            int neigh = adj[node][i];
            if (vis[neigh] == 0) {
                vis[neigh] = 1;
                Enqueue(&q, neigh);
            }
        }
    }
}
```

```

    } } }
void dfs(int i, int **adj, int *vis, int *adjCount){
    vis[i] = 1;
    printf("%d ", i);
    for(int j = 0; j<adjCount[i]; j++){
        int node = adj[i][j];
        if(vis[node]==0){
            dfs(node, adj, vis, adjCount);
        }
    }
}
}
int main() {
    int n;
    printf("Enter number of nodes in the graph: ");
    scanf("%d", &n);
    int **adj = (int **)malloc(sizeof(int *) * n);
    int *adjCount = (int *)malloc(sizeof(int) * n);
    for (int i = 0; i < n; i++) {
        printf("Enter the number of adjacent nodes for node %d: ", i);
        scanf("%d", &adjCount[i]);
        adj[i] = (int *)malloc(sizeof(int) * adjCount[i]);
        if(adjCount[i] != 0) printf("Enter adjacent nodes: ");
        for (int j = 0; j < adjCount[i]; j++) {
            scanf("%d", &adj[i][j]);
        }
    }
    int *vis = (int *)malloc(sizeof(int) * n);
    for (int i = 0; i < n; i++) {
        vis[i] = 0;

        printf("The Breadth first traversal of the graph is: ");
        for (int i = 0; i < n; i++) {
            if (vis[i] == 0) {
                bfs(i, adj, vis, adjCount);
            }
        }
    }
}

```

```

    }
}
for (int i = 0; i < n; i++) {
    vis[i] = 0;
}
printf("\nThe depth first traversal of the graph is: ");
for (int i = 0; i < n; i++) {
    if (vis[i] == 0) {
        dfs(i, adj, vis, adjCount);    }
}
for (int i = 0; i < n; i++) {
    free(adj[i]);
}
free(adj);
free(vis);
free(adjCount);
return 0;
}

```

OUTPUT:

```

PS C:\DS Lab> cd C:\DS Lab\Assignment11\ ; if ($?) { gcc
; if ($?) { .\bfsDfs }
Enter number of nodes in the graph: 5
Enter the number of adjacent nodes for node 0: 2
Enter adjacent nodes: 1 2
Enter the number of adjacent nodes for node 1: 1
Enter adjacent nodes: 3
Enter the number of adjacent nodes for node 2: 1
Enter adjacent nodes: 4
Enter the number of adjacent nodes for node 3: 0
Enter the number of adjacent nodes for node 4: 0
The Breadth first traversal of the graph is: 0 1 2 3 4
The depth first traversal of the graph is: 0 1 3 2 4
PS C:\DS Lab\Assignment11>

```

3. Implement Prim's algorithm to find the minimum spanning tree (MST) of a given weighted graph.

⇒

```
#include<stdio.h>
#include<limits.h>
#include<stdbool.h>
#define V 5

int printMST(int res[], int G[V][V]){
    printf("Edge:\tWeight\n");
    for(int i = 1; i < V; i++){
        printf("%d -- %d\t %d\n", res[i], i, G[i][res[i]]);
    }
}

int minKey(int key[], bool mstSet[]){
    int min = INT_MAX, min_index;
    for(int i = 0; i < V; i++){
        if(mstSet[i] == false && key[i] < min){
            min = key[i];
            min_index = i;
        }
    }
    return min_index;
}

void Prims(int G[V][V]){
    int res[V];
    int key[V];
    bool mstSet[V];
    for(int i = 0; i < V; i++){
        key[i] = INT_MAX;
        mstSet[i] = false;
    }
    key[0] = 0;
    res[0] = -1;
    for(int i = 0; i < V - 1; i++){
        int u = minKey(key, mstSet);
```

```

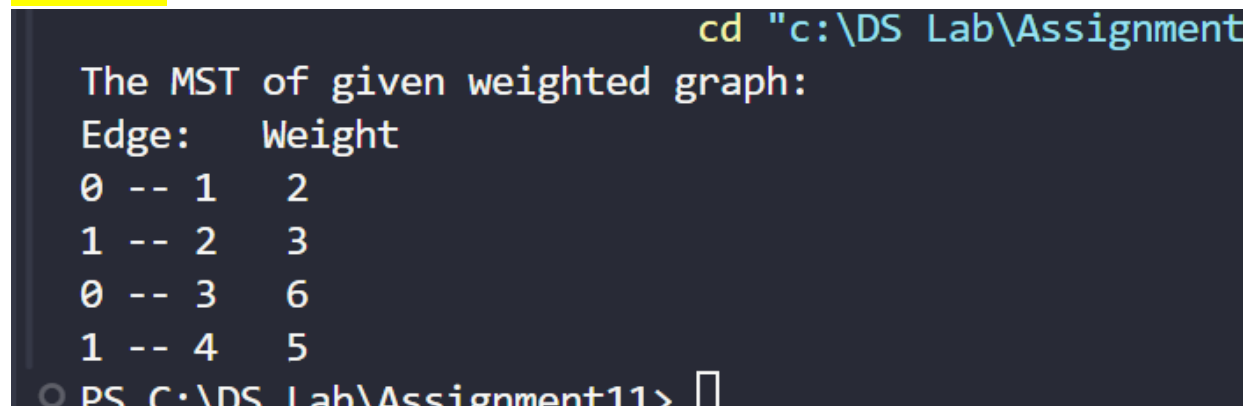
    mstSet[u] = true;
    for(int j = 0; j < V; j++){
        if(G[u][j] && mstSet[j] == false && G[u][j] < key[j]){
            res[j] = u;
            key[j] = G[u][j];
        }
    }
}
printMST(res, G);
}

int main(){
    int G[V][V] = {{0,2,0,6,0},
                    {2,0,3,8,5},
                    {0,3,0,0,7},
                    {6,8,0,0,9},
                    {0,5,7,9,0}};

    printf("The MST of given weighted graph:\n");
    Prims(G);
}

```

OUTPUT:



```

cd "c:\DS Lab\Assignment11"
The MST of given weighted graph:
Edge:  Weight
0 -- 1    2
1 -- 2    3
0 -- 3    6
1 -- 4    5
PS C:\DS Lab\Assignment11>

```

4. Implement Kruskal's algorithm to find the MST of a given graph, adding a constraint that disallows specific edges (selected by the user at runtime). Your algorithm should handle edge constraints, adjust the MST accordingly, and output the cost difference compared to an unconstrained MST.

⇒

```
#include <stdio.h>
#include <limits.h>
#define V 7
#define E 9

void printMST(int T[][V - 1], int edges[3][E], int mstCost) {
    printf("MST Edges:\n");
    for (int i = 0; i < V - 1; i++) {
        printf("%d --> %d (Weight: %d)\n", T[0][i], T[1][i], edges[2][T[2][i]]);
    }
    printf("Total MST Weight: %d\n", mstCost);
}

void Union(int u, int v, int s[]) {
    if (s[u] < s[v]) {
        s[u] += s[v];
        s[v] = u;
    } else {
        s[v] += s[u];
        s[u] = v;
    }
}

int find(int u, int s[]) {
    int x = u, v = 0;
    while (s[x] > 0) {
        x = s[x];
    }
    while (u != x) {
        v = s[u];
        s[u] = x;
        u = v;
    }
    return x;
}
```

```

int Kruskal(int edges[3][E], int excludedEdges[], int excludedCount) {
    int T[3][V - 1];
    int track[E] = {0};
    int set[V + 1];
    for (int i = 1; i <= V; i++) set[i] = -1;
    int mstCost = 0;
    int i = 0;
    while (i < V - 1) {
        int min = INT_MAX;
        int u = 0, v = 0, k = -1;
        for (int j = 0; j < E; j++) {
            int skip = 0;
            for (int x = 0; x < excludedCount; x++) {
                if (excludedEdges[x] == j) {
                    skip = 1;
                    break;
                }
            }
            if (!track[j] && !skip && edges[2][j] < min) {
                min = edges[2][j];
                u = edges[0][j];
                v = edges[1][j];
                k = j;
            }
        }
        if (k == -1) break
        if (find(u, set) != find(v, set)) {
            T[0][i] = u;      T[1][i] = v;      T[2][i] = k;
            mstCost += edges[2][k];
            Union(find(u, set), find(v, set), set);
            i++;
        }
        track[k] = 1;
    }
}

```

```

    }
    if (i == V - 1) {
        printMST(T, edges, mstCost);
        return mstCost;
    } else {
        printf("The graph is disconnected, so no MST can be formed.\n");
        return -1;
    }
}

int main() {
    int edges[3][E] = {
        {1, 1, 2, 2, 3, 4, 4, 5, 5},
        {2, 6, 3, 7, 4, 5, 7, 6, 7},
        {25, 5, 12, 10, 8, 16, 14, 20, 18}
    };
    printf("Original Graph Edges:\n");
    for (int i = 0; i < E; i++) {
        printf("Edge %d: %d --> %d (Weight: %d)\n", i, edges[0][i], edges[1][i],
edges[2][i]);
    }
    printf("\nCalculating MST without constraints...\n");
    int unconstrainedCost = Kruskal(edges, NULL, 0);
    int excludedCount;
    printf("\nEnter the number of edges to exclude: ");
    scanf("%d", &excludedCount);
    int excludedEdges[excludedCount];
    printf("Enter the edge indices to exclude (0 to %d):\n", E - 1);
    for (int i = 0; i < excludedCount; i++) {
        scanf("%d", &excludedEdges[i]);
    }
    printf("\nCalculating MST with constraints...\n");
    int constrainedCost = Kruskal(edges, excludedEdges, excludedCount);
    if (unconstrainedCost != -1 && constrainedCost != -1) {

```



```

        printf("\nCost difference between unconstrained and constrained MST:
%d\n",
        abs(unconstrainedCost - constrainedCost));
    }
    return 0;
}

```

OUTPUT:

Original Graph Edges:

```

Edge 0: 1 --> 2 (Weight: 25)
Edge 1: 1 --> 6 (Weight: 5)
Edge 2: 2 --> 3 (Weight: 12)
Edge 3: 2 --> 7 (Weight: 10)
Edge 4: 3 --> 4 (Weight: 8)
Edge 5: 4 --> 5 (Weight: 16)
Edge 6: 4 --> 7 (Weight: 14)
Edge 7: 5 --> 6 (Weight: 20)
Edge 8: 5 --> 7 (Weight: 18)

```

Calculating MST without constraints...

MST Edges:

```

1 --> 6 (Weight: 5)
3 --> 4 (Weight: 8)
2 --> 7 (Weight: 10)
2 --> 3 (Weight: 12)
4 --> 5 (Weight: 16)
5 --> 6 (Weight: 20)
Total MST Weight: 71

```

Enter the number of edges to exclude: 2

Enter the edge indices to exclude (0 to 8):

3 7

Calculating MST with constraints...

MST Edges:

```

1 --> 6 (Weight: 5)
3 --> 4 (Weight: 8)
2 --> 3 (Weight: 12)
4 --> 7 (Weight: 14)
4 --> 5 (Weight: 16)
1 --> 2 (Weight: 25)
Total MST Weight: 80

```

Cost difference between unconstrained and constrained MST: 9

5. Implement the adjacency list representation of a graph with m vertices and n edges. Extend this functionality to check if a path of exactly k edges exists between any given pair of vertices.

⇒

```
#include<stdio.h>
#include<malloc.h>
#include<stdbool.h>

struct node{
    int data;
    struct node* next;
};

struct node* newNode(int data){
    struct node* new = (struct node*)malloc(sizeof(struct node));
    new->data = data;
    new->next = NULL;
    return new;
}

struct adjList{
    struct node* head;
};

struct graph{
    int V;
    struct adjList* arr;
};

struct graph* createGraph(int V){
    struct graph* g = (struct graph*)malloc(sizeof(struct graph));
    g->V = V;
    g->arr = (struct adjList*)malloc(sizeof(struct adjList) * V);
    for(int i = 0; i < V; i++){
        g->arr[i].head = NULL;
    }
}
```

```

    return g;
}

void addEdge(struct graph* g, int src, int dest){
    struct node* node = newNode(dest);
    node->next = g->arr[src].head;
    g->arr[src].head = node;
}

void printGraph(struct graph* g){
    for(int i = 0; i < g->V; i++){
        struct node* temp = g->arr[i].head;
        printf("Adjacency list of vertex %d: ", i);
        while(temp){
            printf("-> %d", temp->data);
            temp = temp -> next;
        }
        printf("\n");
    }
}

bool checkPath(struct graph* g,int src, int dest, int k){
    struct Node{
        int vertex, depth;
        struct Node* next;
    };
    struct Node *front = NULL, *rear = NULL;
    struct Node* start = (struct Node*)malloc(sizeof(struct Node));
    start->vertex = src;
    start->depth = 0;
    start->next = NULL;
    front = rear = start;
    while (front) {
        int currVertex = front->vertex;
        int currDepth = front->depth;

        struct Node* temp = front;

```

```

front = front->next;
free(temp);
if (currVertex == dest && currDepth == k) return true;
if (currDepth >= k) continue;
struct node* adj = g->arr[currVertex].head;
while (adj) {
    struct Node* EnewNode = (struct Node*)malloc(sizeof(struct Node));
    EnewNode->vertex = adj->data;
    EnewNode->depth = currDepth + 1;
    EnewNode->next = NULL;

    if (rear) rear->next = EnewNode;
    rear = EnewNode;
    if (!front) front = rear;
    adj = adj->next;
}
}
return 0;
}

```

```

int main(){
    int V, E;
    printf("Enter the number of vertices of the graph: ");
    scanf("%d", &V);
    printf("Enter the number of edges: ");
    scanf("%d", &E);
    struct graph* g = createGraph(V);
    printf("Enter the edges (src dest):\n");
    for (int i = 0; i < E; i++) {
        int src, dest;
        scanf("%d %d", &src, &dest);
        addEdge(g, src, dest);
    }
    printGraph(g);
}

```

```

int src, dest, k;
printf("Enter the source, destination, and value of k: ");
scanf("%d %d %d", &src, &dest, &k);
if (checkPath(g, src, dest, k)) {
    printf("A path of length %d exists from %d to %d.\n", k, src, dest);
} else {
    printf("No path of length %d exists from %d to %d.\n", k, src, dest);
}
return 0;
}

```

OUTPUT:

```

Enter the number of vertices of the graph: 5
Enter the number of edges: 7
Enter the edges (src dest):
1 2
1 3
1 4
2 3
2 4
3 0
3 4
Adjacency list of vertex 0:
Adjacency list of vertex 1: -> 4-> 3-> 2
Adjacency list of vertex 2: -> 4-> 3
Adjacency list of vertex 3: -> 4-> 0
Adjacency list of vertex 4:
Enter the source, destination, and value of k: 2 0 2
A path of length 2 exists from 2 to 0.

```