Task1:

#include <iostream>

using namespace std;

class Graph {

int\*\* matrix;

bool\* visited;

int offices;

public:

Graph(int n) {

offices = n;

matrix = new int\* [n];

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

matrix[i] = new int[n];

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

matrix[i][j] = 0;

}

visited = new bool[n];

}

~Graph() {

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

delete[] matrix[i];

delete[] matrix;

delete[] visited;

}

void addEdge(int x, int y) {

x--;

y--;

matrix[x][y] = 1;

matrix[y][x] = 1;

}

void reset\_visited() {

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

visited[i] = false;

}

void dfs(int start) {

visited[start] = true;

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

if (matrix[start][i] == 1 && !visited[i]) {

dfs(i);

}

}

}

int countGroups() {

int groups = 0;

reset\_visited();

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

if (!visited[i]) {

dfs(i);

groups++;

}

}

return groups;

}

};

int main() {

int n = 6;

Graph network(n);

network.addEdge(1, 2);

network.addEdge(2, 3);

network.addEdge(4, 5);

int groups = network.countGroups();

if (groups == 1)

cout << "Network is fully connected!" << endl;

else

cout << "Network is not fully connected. Number of isolated groups: " << groups << endl;

return 0;

} A screenshot of a computer

Description automatically generated

Task2:

#include <iostream>

using namespace std;

class Queue

{

int data[100];

int front, rear;

public:

Queue()

{

front = rear = -1;

}

void enqueue(int value)

{

if (rear == 99)

{

return;

}

if (front == -1)

{

front = 0;

}

data[++rear] = value;

}

int dequeue()

{

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

{

return -1;

}

return data[front++];

}

bool isEmpty()

{

return front == -1 || front > rear;

}

};

void findShortestPath(int graph[][7], int start, int end, int n)

{

bool visited[7] = { false };

int parent[7] = { -1 };

Queue q;

q.enqueue(start);

visited[start] = true;

while (!q.isEmpty())

{

int current = q.dequeue();

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

{

if (graph[current][i] == 1 && !visited[i])

{

visited[i] = true;

parent[i] = current;

q.enqueue(i);

if (i == end)

{

break;

}

}

}

}

if (!visited[end])

{

cout << "No path exists.\n";

return;

}

int path[7];

int index = 0;

for (int i = end; i != -1; i = parent[i])

{

path[index++] = i;

}

cout << "Shortest route length: " << index - 1 << " and Route: ";

for (int i = index - 1; i >= 0; i--)

{

cout << path[i] + 1;

if (i > 0)

{

cout << " -> ";

}

}

cout << endl;

}

int main()

{

int graph[7][7] = { 0 };

graph[0][1] = 1;

graph[0][2] = 1;

graph[1][3] = 1;

graph[2][4] = 1;

graph[4][5] = 1;

graph[5][3] = 1;

int start = 0, end = 5;

findShortestPath(graph, start, end, 7);

return 0;

}

A computer screen shot of a black screen

Description automatically generated

Task3:

#include <iostream>

using namespace std;

struct Edge

{

int u, v, weight;

};

void sortEdges(Edge edges[], int n)

{

for (int i = 0; i < n - 1; i++)

{

for (int j = 0; j < n - i - 1; j++)

{

if (edges[j].weight > edges[j + 1].weight)

{

Edge temp = edges[j];

edges[j] = edges[j + 1];

edges[j + 1] = temp;

}

}

}

}

int findParent(int node, int parent[])

{

while (parent[node] != node)

{

node = parent[node];

}

return node;

}

void unionNodes(int u, int v, int parent[], int rank[])

{

int parentU = findParent(u, parent);

int parentV = findParent(v, parent);

if (rank[parentU] > rank[parentV])

{

parent[parentV] = parentU;

}

else if (rank[parentU] < rank[parentV])

{

parent[parentU] = parentV;

}

else

{

parent[parentV] = parentU;

rank[parentU]++;

}

}

void kruskal(Edge edges[], int numVertices, int numEdges)

{

int\* parent = new int[numVertices + 1];

int\* rank = new int[numVertices + 1];

for (int i = 1; i <= numVertices; i++)

{

parent[i] = i;

rank[i] = 0;

}

sortEdges(edges, numEdges);

int totalCost = 0;

cout << "Selected connections: ";

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

{

int u = edges[i].u;

int v = edges[i].v;

if (findParent(u, parent) != findParent(v, parent))

{

totalCost += edges[i].weight;

cout << "(" << u << ", " << v << ") ";

unionNodes(u, v, parent, rank);

}

}

cout << "\nTotal cost: " << totalCost << endl;

delete[] parent;

delete[] rank;

}

int main()

{

const int numVertices = 6;

const int numEdges = 9;

Edge edges[] = { {1, 2, 4}, {1, 3, 3}, {2, 3, 1}, {2, 4, 2},{3, 4, 5}, {3, 5, 7}, {4, 5, 6}, {4, 6, 8}, {5, 6, 9} };

kruskal(edges, numVertices, numEdges);

return 0;

}

A screenshot of a computer

Description automatically generated  
Task4:

#include <iostream>

using namespace std;

const int MAX = 100;

const int INF = 1e9;

void primAlgorithm(int graph[MAX][MAX], int vertices, int start)

{

int key[MAX], parent[MAX];

bool included[MAX];

int totalCost = 0;

for (int i = 1; i <= vertices; i++)

{

key[i] = INF;

included[i] = false;

parent[i] = -1;

}

key[start] = 0;

for (int i = 1; i <= vertices; i++)

{

int minKey = INF;

int current = -1;

for (int j = 1; j <= vertices; j++)

{

if (!included[j] && key[j] < minKey)

{

minKey = key[j];

current = j;

}

}

included[current] = true;

for (int j = 1; j <= vertices; j++)

{

if (graph[current][j] > 0 && !included[j] && graph[current][j] < key[j])

{

key[j] = graph[current][j];

parent[j] = current;

}

}

}

cout << "Selected connections: ";

for (int i = 2; i <= vertices; i++)

{

if (parent[i] != -1)

{

cout << "(" << parent[i] << ", " << i << ") ";

totalCost += graph[parent[i]][i];

}

}

cout << "\nTotal cost: " << totalCost << endl;

}

int main()

{

int vertices = 4;

int graph[MAX][MAX] = { 0 };

graph[1][2] = 1;

graph[2][1] = 1;

graph[1][3] = 4;

graph[3][1] = 4;

graph[1][4] = 5;

graph[4][1] = 5;

graph[2][3] = 2;

graph[3][2] = 2;

graph[3][4] = 3;

graph[4][3] = 3;

primAlgorithm(graph, vertices, 1);

return 0;

}

A screenshot of a computer

Description automatically generated