**DELHI TECHNOLOGICAL UNIVERSITY**



**GRAPH THEORY**

**MC-405**

**Practical File**

**SUBMITTED TO:                                              SUBMITTED BY:**

Sangita Kansal Aiman Siddiqua

Radhika Kavra 2K18/MC/008

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| **S No.** | **TOPIC** | **DATE** | **SIGNATURE** |
| 1. | Program to find the number of vertices, even vertices, odd vertices and number of edges in a graph. | 17/08/2021 |  |
| 2. | Program to find union, intersection and ring sum of two graphs. | 17/08/2021 |  |
| 3. | Program to find minimal spanning tree of a graph using Prim’s Algorithm. | 21/09/2021 |  |
| 4. | Program to find minimal spanning tree of a graph using Kruskal’s Algorithm. | 21/09/2021 |  |
| 5. | Program to find shortest path between two vertices in a graph using Dijkstra’s Algorithm. | 07/09/2021 |  |
| 6. | Program to find shortest path between every pair of vertices in a graph using Floyd-Warshall’s Algorithm. | 31/08/2021 |  |
| 7. | Program to find shortest path between two vertices in a graph using Bellman-Ford’s Algorithm. | 31/08/2021 |  |
| 8. | Program to find maximum matching for bipartite graph. | 26/10/2021 |  |
| 9. | Program to find maximum matching for general graph. | 26/10/2021 |  |
| 10. | Program to find maximum flow from source node to sink node using Ford-Fulkerson Algorithm. | 05/10/2021 |  |

PRACTICAL – 1

AIM: To write a program to find the number of vertices, even vertices, odd vertices and number of edges in a graph.

CODE:

#include <bits/stdc++.h>

**using** **namespace** std;

**class** **Graph**

{

**int** V;

list<**int**> \*adj;

**public:**

Graph(**int** V)

{

**this**->V = V;

adj = **new** list<**int**>[V];

}

**void** addEdge(**int** u, **int** v)

{

adj[u].push\_back(v);

adj[v].push\_back(u);

}

**int** noOfVertices() { **return** **this**->V; }

**int** countEdges()

{

**int** sum = **0**;

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

sum += adj[i].size();

**return** sum / **2**;

}

**int** evenVertices()

{

**int** count = **0**;

**for** (**int** i = **0**; i < **this**->V; i++)

{

**if** (adj[i].size() % **2** == **0**)

count++;

}

**return** count;

}

**int** oddVertices()

{

**return** **this**->V - evenVertices();

}

};

**int** **main**()

{

**int** V = **5**;

Graph g(V);

g.addEdge(**0**, **1**);

g.addEdge(**3**, **2**);

g.addEdge(**0**, **3**);

g.addEdge(**1**, **3**);

g.addEdge(**2**, **4**);

g.addEdge(**1**, **4**);

cout << "Number of Vertices: " << g.noOfVertices() << endl;

cout << "Number of Even Vertices: " << g.evenVertices() << endl;

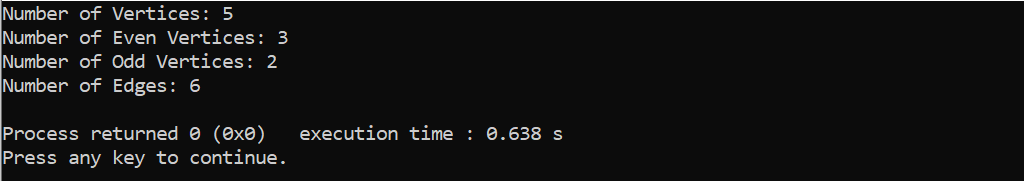
cout << "Number of Odd Vertices: " << g.oddVertices() << endl;

cout << "Number of Edges: " << g.countEdges() << endl;

**return** **0**;

}

OUTPUT:



PRACTICAL – 2

AIM: To write a program to find union, intersection and ring-sum of two graphs.

CODE:

**UNION**

#include <iostream>

**using** **namespace** std;

**int** V1[] = {**0**, **1**};

**int** V2[] = {**0**, **1**, **2**};

**int** E1[**2**][**2**], E2[**3**][**3**], E3[**5**][**5**];

**void** **Union**(**int** arr1[], **int** arr2[], **int** m, **int** n)

{

cout << "**\n**Set of vertices in union of the graphs G1 and G2 is:**\n**";

**int** i = **0**, j = **0**;

**while** (i < m && j < n)

{

**if** (arr1[i] < arr2[j])

cout << arr1[i++]<<" ";

**else** **if** (arr2[j] < arr1[i])

cout << arr2[j++]<<" ";

**else**

{

cout << arr2[j++]<<" ";

i++;

}

}

**while** (i < m)

cout << arr1[i++]<<" ";

**while** (j < n)

cout << arr2[j++]<<" ";

cout << "**\n**";

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

{

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

{

**if** (i < m && j < m && E1[i][j] > E2[i][j])

E3[i][j] = E1[i][j];

**else** **if** (i < m && j < m && E1[i][j] < E2[i][j])

E3[i][j] = E2[i][j];

**else**

E3[i][j] = E2[i][j];

}

}

cout << "**\n**Adjacency matrix of union of graphs G1 and G2 is:**\n**";

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

{

cout << "**\t**" << i;

}

cout << "**\n\t**";

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

{

cout << " ";

}

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

{

cout << "**\n**"

<< i << "|**\t**";

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

{

cout << E3[i][j] << "**\t**";

}

}

cout << "**\n**";

}

**int** **main**()

{

**int** m = **sizeof**(V1) / **sizeof**(V1[**0**]);

**int** n = **sizeof**(V2) / **sizeof**(V2[**0**]);

**int** i, j, k;

cout << "Enter the adjacency matrix(symmetric) for graph 1:" << endl;

**for** (i = **0**; i < m; i++)

{

**for** (j = **0**; j < m; j++)

cin >> E1[i][j];

}

cout << "**\n**Enter the adjacency matrix(symmetric) for graph 2"<<endl;

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

{

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

cin >> E2[i][j];

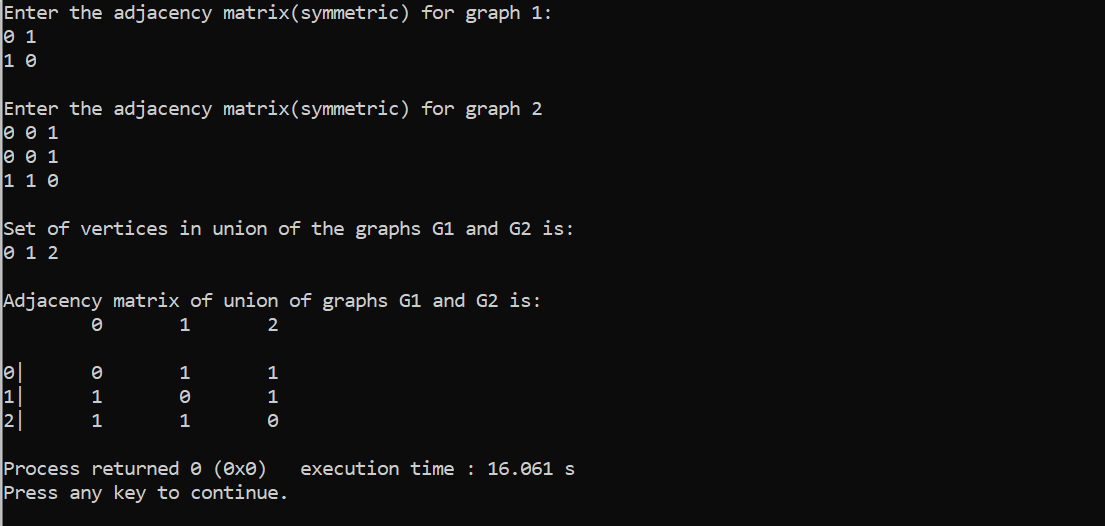
}

Union(V1, V2, m, n);

**return** **0**;

}

Output



**INTERSECTION**

**void** **intersection**(**int** arr1[], **int** arr2[], **int** m, **int** n)

{

cout << "**\n**Set of vertices in intersection of the graphs G1 and G2 is:**\n**";

**int** i = **0**, j = **0**;

**while** (i < m && j < n)

{

**if** (arr1[i] < arr2[j])

i++;

**else** **if** (arr2[j] < arr1[i])

j++;

**else**

{

cout << arr2[j++]<<" ";

i++;

}

}

cout << "**\n**";

**for** (i = **0**; i < m; i++)

**for** (j = **0**; j < m; j++)

{

**if** (E1[i][j] == E2[i][j])

E3[i][j] = E1[i][j];

**else**

E3[i][j] = **0**;

}

cout << "**\n**Adjacency matrix of intersection of graphs G1 and G2 is:**\n\t**";

**for** (i = **0**; i < m; i++)

cout << i << "**\t**";

cout << "**\n\t**";

**for** (i = **0**; i < m; i++)

cout << " ";

**for** (i = **0**; i < m; i++)

{

cout << "**\n**"

<< i << "|**\t**";

**for** (j = **0**; j < m; j++)

{

cout << E3[i][j] << "**\t**";

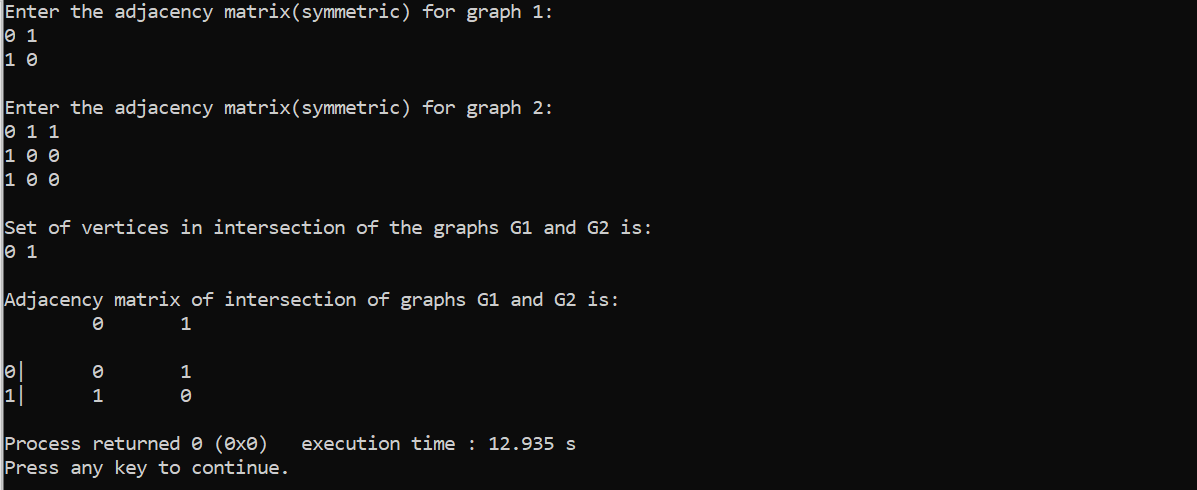
}

}

cout << endl;

}

**Output**



**RING SUM**

**void** **ring\_sum**(**int** arr1[], **int** arr2[], **int** m, **int** n)

{

cout << "**\n**Set of vertices in ring sum of the graphs G1 and G2 are:**\n**";

**int** i = **0**, j = **0**;

**while** (i < m && j < n)

{

**if** (arr1[i] < arr2[j])

cout << arr1[i++]<<" ";

**else** **if** (arr2[j] < arr1[i])

cout << arr2[j++]<<" ";

**else**

{

cout << arr2[j++]<<" ";

i++;

}

}

**while** (i < m)

cout << arr1[i++];

**while** (j < n)

cout << arr2[j++];

cout << "**\n**";

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

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

{

**if** (i<m && j<m && E1[i][j] == E2[i][j])

E3[i][j] = **0**;

**else** **if** (i<m && j<m && E1[i][j]>E2[i][j])

E3[i][j] = E1[i][j];

**else**

E3[i][j] = E2[i][j];

}

}

cout << "**\n**Adjacency matrix of ring sum of graphs G1 and G2 is:**\n\t**";

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

cout << i << "**\t**";

cout << "**\n\t**";

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

cout << " ";

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

{

cout << "**\n**"

<< i << "|**\t**";

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

{

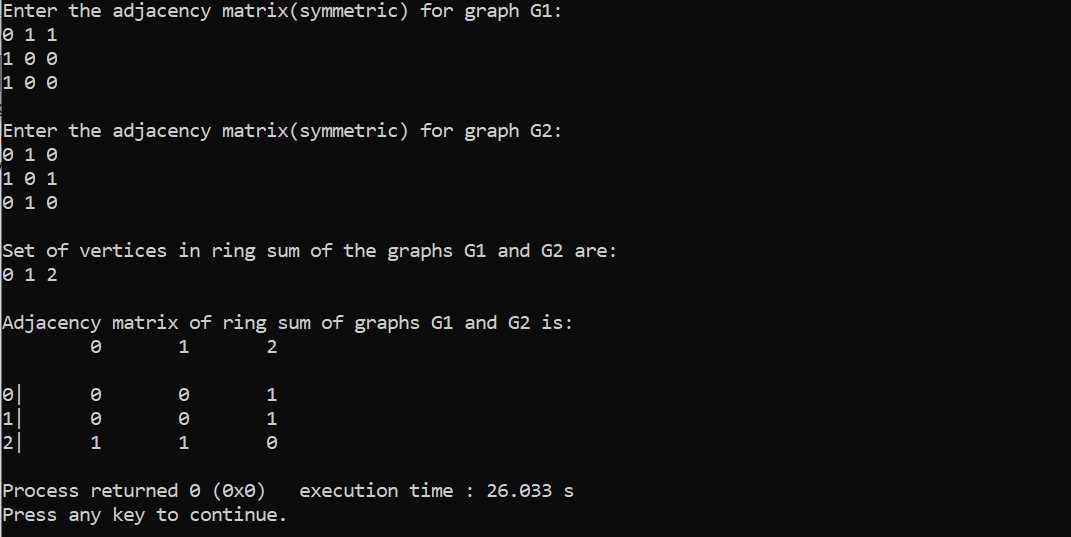
cout << E3[i][j] << "**\t**";

}

}

}

Output

****

PRACTICAL – 3

AIM: To write a program to find the minimum spanning tree of a graph using Prim’s Algorithm.

CODE:

#include <bits/stdc++.h>

**using** **namespace** std;

#define V 5

**int** **minKey**(**int** key[], **bool** mstSet[])

{

**int** min = INT\_MAX, min\_index;

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

**if** (mstSet[v] == false && key[v] < min)

min = key[v], min\_index = v;

**return** min\_index;

}

**void** **printMST**(**int** parent[], **int** graph[V][V])

{

cout<<"Edge **\t**Weight**\n**";

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

cout<<parent[i]<<" - "<<i<<" **\t**"<<graph[i][parent[i]]<<" **\n**";

}

**void** **primMST**(**int** graph[V][V])

{

**int** parent[V];

**int** key[V];

**bool** mstSet[V];

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

key[i] = INT\_MAX, mstSet[i] = false;

key[**0**] = **0**;

parent[**0**] = -**1**;

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

{

**int** u = minKey(key, mstSet);

mstSet[u] = true;

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

**if** (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])

parent[v] = u, key[v] = graph[u][v];

}

printMST(parent, graph);

}

**int** **main**()

{

**int** graph[V][V] = { { **0**, **2**, **0**, **6**, **0** },

{ **2**, **0**, **3**, **8**, **5** },

{ **0**, **3**, **0**, **0**, **7** },

{ **6**, **8**, **0**, **0**, **9** },

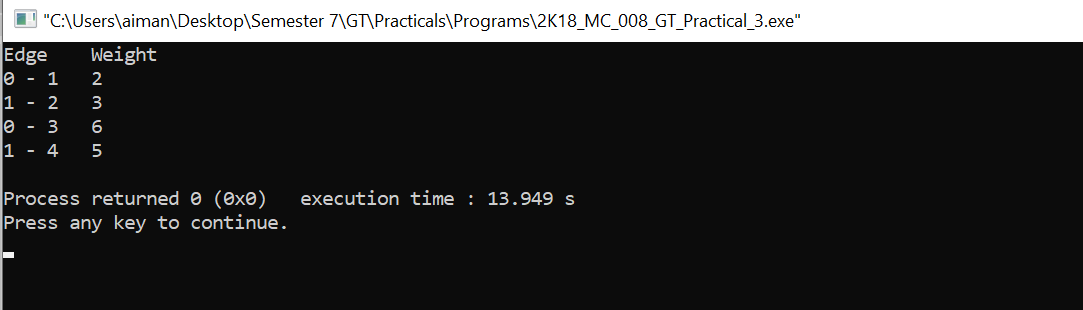
{ **0**, **5**, **7**, **9**, **0** } };

primMST(graph);

**return** **0**;

}

OUTPUT:



PRACTICAL – 4

AIM: To write a program to find the minimum spanning tree of a graph using Kruskal’s Algorithm.

CODE:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

**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(**sizeof**( **struct** Edge)\*E);

**return** graph;

}

**struct** subset {

**int** parent;

**int** rank;

};

**int** **find**(**struct** subset subsets[], **int** i)

{

**if** (subsets[i].parent != i)

subsets[i].parent

= find(subsets, subsets[i].parent);

**return** subsets[i].parent;

}

**void** **Union**(**struct** subset subsets[], **int** x, **int** y)

{

**int** xroot = find(subsets, x);

**int** yroot = find(subsets, y);

**if** (subsets[xroot].rank < subsets[yroot].rank)

subsets[xroot].parent = yroot;

**else** **if** (subsets[xroot].rank > subsets[yroot].rank)

subsets[yroot].parent = xroot;

**else**

{

subsets[yroot].parent = xroot;

subsets[xroot].rank++;

}

}

**int** **myComp**(**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;

**struct** Edge

result[V];

**int** e = **0**;

**int** i = **0**;

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

myComp);

**struct** subset\* subsets

= (**struct** subset\*)malloc(V \* **sizeof**(**struct** subset));

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

subsets[v].parent = v;

subsets[v].rank = **0**;

}

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

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

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

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

**if** (x != y) {

result[e++] = next\_edge;

Union(subsets, x, y);

}

}

printf(

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

**int** minimumCost = **0**;

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

{

printf("%d -- %d == %d**\n**", result[i].src,

result[i].dest, result[i].weight);

minimumCost += result[i].weight;

}

printf("**\n**Minimum Cost Spanning tree : %d",minimumCost);

**return**;

}

**int** **main**()

{

**int** V = **4**;

**int** 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**;

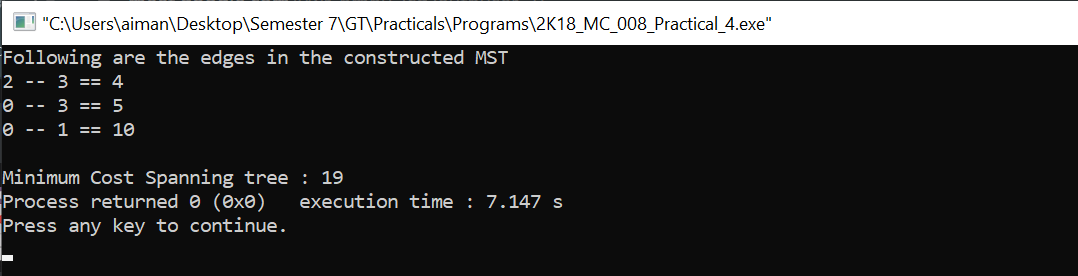
graph->edge[**4**].weight = **4**;

KruskalMST(graph);

**return** **0**;

}

OUTPUT:



PRACTICAL – 5

AIM: To write a program to find the shortest path between two vertices in a graph using Dijkstra’s Algorithm.

CODE:

#include <bits/stdc++.h>

**using** **namespace** std;

#define V 8

**int** **minDistance**(**int** dist[], **bool** sptSet[])

{

**int** min = INT\_MAX, min\_index;

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

**if** (sptSet[v] == false && dist[v] <= min)

min = dist[v], min\_index = v;

**return** min\_index;

}

**void** **dijkstra**(**int** graph[V][V], **int** src)

{

**int** dist[V];

**bool** sptSet[V];

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

dist[i] = INT\_MAX, sptSet[i] = false;

dist[src] = **0**;

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

**int** u = minDistance(dist, sptSet);

sptSet[u] = true;

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

**if** (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX

&& dist[u] + graph[u][v] < dist[v])

dist[v] = dist[u] + graph[u][v];

}

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

cout <<"Distance of vertex " << i << " from source is "<<dist[i]<< endl;

}

**int** **main**()

{

**int** graph[V][V] = { { **0**, **8**, **0**, **0**, **0**, **0**, **0**, **4**},

{ **8**, **0**, **4**, **0**, **0**, **0**, **0**, **1**},

{ **0**, **7**, **0**, **4**, **0**, **8**, **0**, **0**},

{ **0**, **0**, **7**, **0**, **9**, **4**, **0**, **0**},

{ **0**, **0**, **0**, **9**, **0**, **10**, **0**, **0**},

{ **0**, **0**, **4**, **11**, **10**, **0**, **2**, **0**},

{ **0**, **0**, **0**, **0**, **0**, **2**, **0**, **1**},

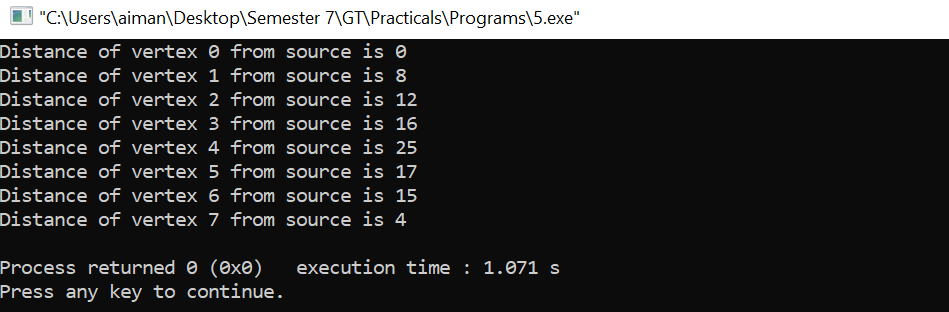
{ **8**, **6**, **0**, **0**, **0**, **0**, **11**, **0**}};

dijkstra(graph, **0**);

**return** **0**;

}

OUTPUT:



PRACTICAL – 6

AIM: To write a program to find the shortest path between every pair of vertices in a graph using Floyd-Warshall’s Algorithm.

**CODE**:

#include<bits/stdc++.h>

**using** **namespace** std;

#define N 4

#define INF 100000

**void** **floydWarshall**(**int** g[][N])

{

**int** dist[N][N], i, j, k;

**for** (i = **0**; i < N; i++)

**for** (j = **0**; j < N; j++)

dist[i][j] = g[i][j];

**for** (k = **0**; k < N; k++) {

**for** (i = **0**; i < N; i++) {

**for** (j = **0**; j < N; j++) {

**if** (dist[i][j] > (dist[i][k] + dist[k][j])

&& (dist[k][j] != INF

&& dist[i][k] != INF))

dist[i][j] = dist[i][k] + dist[k][j];

}

}

}

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

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

**if** (dist[i][j] == INF)

cout << "INF"

<< " ";

**else**

cout << dist[i][j] << " ";

}

cout << endl;

}

}

**int** **main**(){

**int** g[**4**][**4**] = {{**0** ,**5**,INF,**2**},

{INF,**2**,**4**,**1**},

{INF,INF,**0**,**1**},

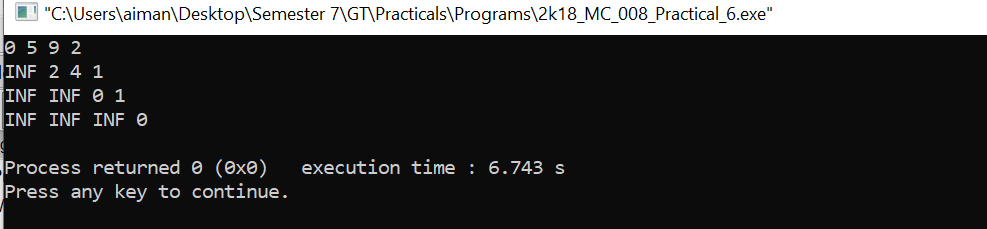
{INF,INF,INF,**0**}};

floydWarshall(g);

**return** **0**;

}

OUTPUT:



PRACTICAL – 7

AIM: To write a program to find the shortest path between two vertices in a graph using Bellaman Ford’s Algorithm.

**CODE**:

#include<bits/stdc++.h>

**using** **namespace** std;

**struct** Edge {

**int** src, dest, weight;

};

**struct** Graph {

**int** V, E;

**struct** Edge\* edge;

};

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

{

**struct** Graph\* graph = **new** Graph;

graph->V = V;

graph->E = E;

graph->edge = **new** Edge[E];

**return** graph;

}

**void** **BellmanFord**(**struct** Graph\* graph, **int** src)

{

**int** V = graph->V;

**int** E = graph->E;

**int** dist[V];

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

dist[i] = INT\_MAX;

dist[src] = **0**;

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

**for** (**int** j = **0**; j < E; j++) {

**int** u = graph->edge[j].src;

**int** v = graph->edge[j].dest;

**int** weight = graph->edge[j].weight;

**if** (dist[u] != INT\_MAX && dist[u] + weight < dist[v])

dist[v] = dist[u] + weight;

}

}

**for** (**int** i = **0**; i < E; i++) {

**int** u = graph->edge[i].src;

**int** v = graph->edge[i].dest;

**int** weight = graph->edge[i].weight;

**if** (dist[u] != INT\_MAX && dist[u] + weight < dist[v]) {

printf("Graph contains negative weight cycle");

**return**;

}

}

printf("Vertex Distance from Source**\n**");

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

printf("%d **\t\t** %d**\n**", i, dist[i]);

}

**int** **main**()

{

**int** V = **5**;

**int** E = **8**;

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

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

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

graph->edge[**0**].weight = -**1**;

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

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

graph->edge[**1**].weight = **4**;

graph->edge[**2**].src = **1**;

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

graph->edge[**2**].weight = **3**;

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

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

graph->edge[**3**].weight = **2**;

graph->edge[**4**].src = **1**;

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

graph->edge[**4**].weight = **2**;

graph->edge[**5**].src = **3**;

graph->edge[**5**].dest = **2**;

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

graph->edge[**6**].src = **3**;

graph->edge[**6**].dest = **1**;

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

graph->edge[**7**].src = **4**;

graph->edge[**7**].dest = **3**;

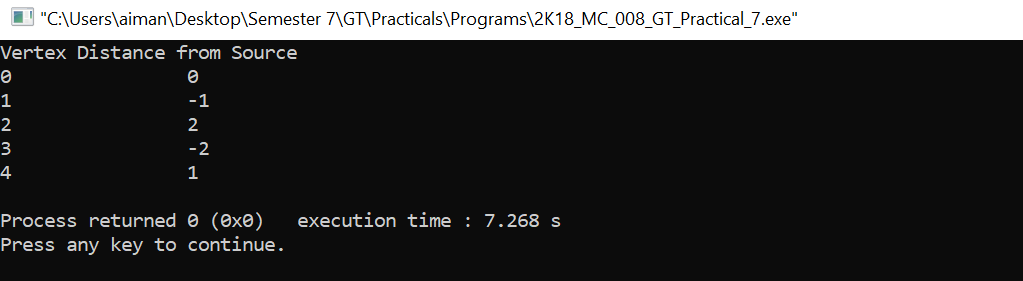
graph->edge[**7**].weight = -**3**;

BellmanFord(graph, **0**);

**return** **0**;

}

OUTPUT:



PRACTICAL – 8

AIM: To write a program to find maximum matching in bipartite graph.

CODE:

#include <bits/stdc++.h>

**using** **namespace** std;

**int** n, k;

vector<vector<**int**>> g;

vector<**int**> mt;

vector<**bool**> used;

**bool** **try\_kuhn**(**int** v) {

**if** (used[v])

**return** false;

used[v] = true;

**for** (**int** to : g[v]) {

**if** (mt[to] == -**1** || try\_kuhn(mt[to])) {

mt[to] = v;

**return** true;

}

}

**return** false;

}

**int** **main**(){

cin>>n>>k;

**int** m;

cin>>m;

g.assign(n, vector<**int**>());

**for**(**int** i =**0**; i<m; i++){

**int** a, b;

cin>>a>>b;

g[--a].push\_back(--b);

}

mt.assign(k, -**1**);

**for** (**int** v = **0**; v < n; ++v) {

used.assign(n, false);

try\_kuhn(v);

}

**int** count = **0**;

**for** (**int** i = **0**; i < k; ++i)

**if** (mt[i] != -**1**)

count++;

cout<<"**\n**Maximum matching in given bipartite graph : **\n**"<<count<<"**\n**";

cout<<"**\n**Included Edges: **\n**";

**for** (**int** i = **0**; i < k; ++i)

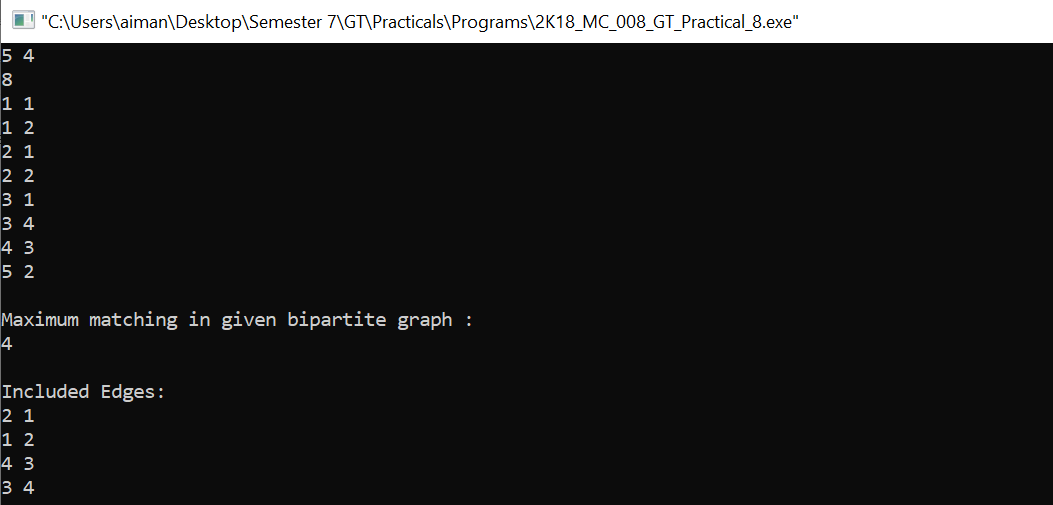
**if** (mt[i] != -**1**)

printf("%d %d**\n**", mt[i] + **1**, i + **1**);

**return** **0**;

}

OUTPUT:



PRACTICAL – 9

AIM: To write a program to find maximum matching in a general graph.

CODE:

#include <bits/stdc++.h>

**using** **namespace** std;

**const** **int** M=**500**;

**struct** struct\_edge{**int** v;struct\_edge\* n;};

**typedef** struct\_edge\* edge;

struct\_edge pool[M\*M\***2**];

edge top=pool,adj[M];

**int** V,E,match[M],qh,qt,q[M],father[M],base[M];

**bool** inq[M],inb[M],ed[M][M];

**void** **add\_edge**(**int** u,**int** v){

top->v=v,top->n=adj[u],adj[u]=top++;

top->v=u,top->n=adj[v],adj[v]=top++;

}

**int** **LCA**(**int** root,**int** u,**int** v){

**static** **bool** inp[M];

memset(inp,**0**,**sizeof**(inp));

**while**(**1**){

inp[u=base[u]]=true;

**if** (u==root)

**break**;

u=father[match[u]];

}

**while**(**1**){

**if** (inp[v=base[v]])

**return** v;

**else**

v=father[match[v]];

}

}

**void** **mark\_blossom**(**int** lca,**int** u)

{

**while** (base[u]!=lca)

{

**int** v=match[u];

inb[base[u]]=inb[base[v]]=true;

u=father[v];

**if** (base[u]!=lca)

father[u]=v;

}

}

**void** **blossom\_contraction**(**int** s,**int** u,**int** v)

{

**int** lca=LCA(s,u,v);

memset(inb,**0**,**sizeof**(inb));

mark\_blossom(lca,u);

mark\_blossom(lca,v);

**if** (base[u]!=lca)

father[u]=v;

**if** (base[v]!=lca)

father[v]=u;

**for** (**int** u=**0**;u<V;u++)

**if** (inb[base[u]]){

base[u]=lca;

**if** (!inq[u])

inq[q[++qt]=u]=true;

}

}

**int** **find\_augmenting\_path**(**int** s)

{

memset(inq,**0**,**sizeof**(inq));

memset(father,-**1**,**sizeof**(father));

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

base[i]=i;

inq[q[qh=qt=**0**]=s]=true;

**while** (qh<=qt)

{

**int** u=q[qh++];

**for** (edge e=adj[u];e;e=e->n)

{

**int** v=e->v;

**if** (base[u]!=base[v]&&match[u]!=v)

**if** ((v==s)||(match[v]!=-**1** && father[match[v]]!=-**1**))

blossom\_contraction(s,u,v);

**else** **if** (father[v]==-**1**)

{

father[v]=u;

**if** (match[v]==-**1**)

**return** v;

**else** **if** (!inq[match[v]])

inq[q[++qt]=match[v]]=true;

}

}

}

**return** -**1**;

}

**int** **augment\_path**(**int** s,**int** t)

{

**int** u=t,v,w;

**while** (u!=-**1**)

{

v=father[u];

w=match[v];

match[v]=u;

match[u]=v;

u=w;

}

**return** t!=-**1**;

}

**int** **edmonds**()

{

**int** matchc=**0**;

memset(match,-**1**,**sizeof**(match));

**for** (**int** u=**0**;u<V;u++)

**if** (match[u]==-**1**)

matchc+=augment\_path(u,find\_augmenting\_path(u));

**return** matchc;

}

**int** **main**()

{

**int** u,v;

cin>>V>>E;

**while**(E--){

cin>>u>>v;

**if** (!ed[u-**1**][v-**1**]){

add\_edge(u-**1**,v-**1**);

ed[u-**1**][v-**1**]=ed[v-**1**][u-**1**]=true;

}

}

cout<<"**\n**Maximum matching in given non-bipartite graph is :**\n**"; cout<<edmonds()<<endl;

cout<<"**\n**Included edges :**\n**";

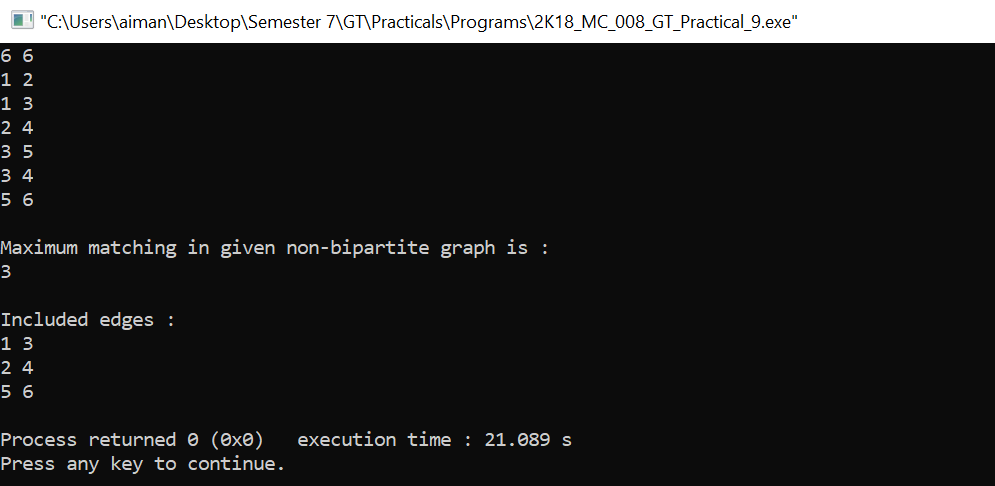
**for** (**int** i=**0**;i<V;i++)

**if** (i<match[i])

cout<<i+**1**<<" "<<match[i]+**1**<<endl;

}

OUTPUT:



PRACTICAL – 10

AIM: To write a program to find the maximum flow from source node to sink node using Ford-Fulkerson Algorithm.

CODE:

#include<bits/stdc++.h>

**using** **namespace** std;

#define V 6

**bool** **bfs**(**int** rGraph[V][V], **int** s, **int** t, **int** parent[])

{

**bool** visited[V];

memset(visited, **0**, **sizeof**(visited));

queue<**int**> q;

q.push(s);

visited[s] = true;

parent[s] = -**1**;

**while** (!q.empty()) {

**int** u = q.front();

q.pop();

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

**if** (visited[v] == false && rGraph[u][v] > **0**) {

**if** (v == t) {

parent[v] = u;

**return** true;

}

q.push(v);

parent[v] = u;

visited[v] = true;

}

}

}

**return** false;

}

**int** **fordFulkerson**(**int** graph[V][V], **int** s, **int** t)

{

**int** u, v;

**int** rGraph[V][V];

**for** (u = **0**; u < V; u++)

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

rGraph[u][v] = graph[u][v];

**int** parent[V];

**int** max\_flow = **0**;

**while** (bfs(rGraph, s, t, parent)) {

**int** path\_flow = INT\_MAX;

**for** (v = t; v != s; v = parent[v]) {

u = parent[v];

path\_flow = min(path\_flow, rGraph[u][v]);

}

**for** (v = t; v != s; v = parent[v]) {

u = parent[v];

rGraph[u][v] -= path\_flow;

rGraph[v][u] += path\_flow;

}

max\_flow += path\_flow;

}

**return** max\_flow;

}

**int** **main**()

{

**int** graph[V][V]

= { { **0**, **16**, **13**, **0**, **0**, **0** }, { **0**, **0**, **10**, **12**, **0**, **0** },

{ **0**, **4**, **0**, **0**, **14**, **0** }, { **0**, **0**, **9**, **0**, **0**, **20** },

{ **0**, **0**, **0**, **7**, **0**, **4** }, { **0**, **0**, **0**, **0**, **0**, **0** } };

cout << "The maximum possible flow from source to sink is "

<< fordFulkerson(graph, **0**, **5**);

**return** **0**;

}

OUTPUT:

