

Delhi Technological University
19th November 2020

Graph Theory Lab File

Graph Theory (MC-405)



Anish Sachdeva
DTU / 2K16 / MC / 13

Lab Supervisor: Dr. Anjana Gupta
Lab Assistant: Ms. Payal

Contents

No.	Experiment Name	Date
1	Program to Find the Number of Vertices, Even Vertices, Odd Vertices and Number of Edges in a Graph	20 th August 2020
2	Program to Find Union, Intersection and Ring-Sum of 2 Graphs	27 th September 2020
3	Program to Find Minimum Spanning Tree Using Prim's Algorithm	3 ^d September 2020
4	Program to Find Minimum Spanning Tree Using Kruskal's Algorithm	17 th September 2020
5	Program to find Shortest Path between 2 Vertices using Dijkstra Algorithm	21 st September 2020
6	Program to Find Shortest Path Between Every Pair of vertices in a Graph Using Floyd-Warshall's Algorithm	22 nd October 2020
7	Program to find Shortest Path between Every Pair of Vertices using Bellman Ford's Algorithm	22 nd October 2020
8	Program For Finding Maximal Matching for Bipartite Graph	29 th October 2020
9	Program For Finding Maximal Matching for General Path	29 th October 2020
10	Program to Find Maximum Flow From Source Node to Sink Node Using Ford-Fulkerson Algorithm	29 th October 2020

See all programs in both C++ and Java at github.com/anishLearnsToCode/graph-theory-lab

1 Program to Find the Number of Vertices, Even Vertices, Odd Vertices and Number of Edges in a Graph

1.1 Code

```
// Program to Find the Number of Vertices, Even Vertices, Odd Vertices and
// Number of Edges in a Graph

#include<bits/stdc++.h>

using namespace std;

int main() {
    int n;
    cin >> n;

    int a[n][n];

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cin >> a[i][j];
        }
    }

    int degree[n] = {};

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            degree[i] += a[i][j];
        }
    }

    int even = 0, odd = 0, edges = 0;

    for (int i = 0; i < n; i++) {
        even += (degree[i] % 2 == 0);
        odd += (degree[i] % 2 != 0);
        edges += degree[i];
    }

    cout << "number of even vertices" << " = " << even << '\n';
    cout << "number of odd vertices" << " = " << odd << '\n';
    cout << "number of edges" << " = " << edges / 2 << '\n';
}
```

1.2 Input/Output

Input

```
5
00111
00101
11010
10100
11000
```

(a) Input

Output

```
number of even vertices = 3
number of odd vertices = 2
number of edges = 6
```

(b) Output

Figure 1: Input and Output for Program 1

2 Program to Find Union, Intersection and Ring-Sum of 2 Graphs

2.1 Code

```
#include <iostream>
#include <vector>

using namespace std;

bool IsValid(vector<vector<int> >&adj){
    int n=adj.size();
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            if(adj[i][j]!=adj[j][i]){
                return false;
            }
        }
    }
    return true;
}

void Union(vector<vector<int> >&adj1,vector<vector<int> >&adj2) {
    cout<<"Union ...."<<endl;
    int n1=adj1.size();
    int n2=adj2.size();
    int n=max(n1,n2);
    vector<vector<int> >adj(n,vector<int>(n));
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            adj[i][j]=((i<n1 && j<n1)?adj1[i][j]:0) || ((i<n2 && j<n2)?adj2[i][j]:0);
            cout<<adj[i][j]<< " ";
        }
        cout<<endl;
    }
    cout<<endl;
}

void Intersection(vector<vector<int> >&adj1,vector<vector<int> >&adj2){
    cout<<"Intersection ..."<<endl;
    int n1=adj1.size();
    int n2=adj2.size();
    int n=min(n1,n2);
    vector<vector<int> >adj(n,vector<int>(n));
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            adj[i][j]=adj1[i][j] && adj2[i][j];
            cout<<adj[i][j]<< " ";
        }
        cout<<endl;
    }
    cout<<endl;
}

void RingSum(vector<vector<int> >&adj1,vector<vector<int> >&adj2){
    cout<<"RingSum ..."<<endl;
    int n1=adj1.size();
    int n2=adj2.size();
```

```

int n=max(n1,n2);
vector<vector<int>>adj(n,vector<int>(n));
for(int i=0;i<n;i++){
    for(int j=0;j<n;j++){
        adj[i][j]=((i<n1 && j<n1)?adj1[i][j]:0) ^ ((i<n2 && j<n2)?adj2[i][j]:0);
        cout<<adj[i][j]<<" ";
    }cout<<endl;
}
cout<<endl;
}

int main(){
#ifndef ONLINE_JUDGE
freopen("input.txt", "r", stdin);
freopen("output.txt", "w", stdout);
#endif
int n1;
cin>>n1;
vector<vector<int>>adj1(n1,vector<int>(n1));
for(int i=0;i<n1;i++){
    for(int j=0;j<n1;j++){
        cin>>adj1[i][j];
    }
}
int n2;
cin>>n2;
vector<vector<int>>adj2(n2,vector<int>(n2));
for(int i=0;i<n2;i++){
    for(int j=0;j<n2;j++){
        cin>>adj2[i][j];
    }
}
if(IsValid(adj1) && IsValid(adj2)){
    Union(adj1,adj2);
    Intersection(adj1,adj2);
    RingSum(adj1,adj2);
} else{
    cout<<"Entered Graph is InValid...."<<endl;
}
return 0;
}

```

2.2 Input/Output

```
int[][] matrix1 = {
    {0, 1, 0, 0},
    {1, 0, 0, 0},
    {0, 0, 0, 0},
    {0, 0, 0, 0}
};

int[][] matrix2 = {
    {0, 1, 1, 1},
    {1, 0, 1, 0},
    {1, 1, 0, 0},
    {1, 0, 0, 0}
};
```

(a) Input

```
Union of 2 Graphs
Graph{
adjacencyMatrix=
[0, 1, 1, 1]
[1, 0, 1, 0]
[1, 1, 0, 0]
[1, 0, 0, 0]
, numberOfVertices=4, degree=16
}
```

(b) Union of 2 Graphs

```
Intersection of 2 Graphs
Graph{
adjacencyMatrix=
[0, 1, 0, 0]
[1, 0, 0, 0]
[0, 0, 0, 0]
[0, 0, 0, 0]
, numberOfVertices=4, degree=4
}
```

(c) Intersection of 2 Graphs

```
Ring Sum of 2 Graphs
Graph{
adjacencyMatrix=
[0, 0, 1, 1]
[0, 0, 1, 0]
[1, 1, 0, 0]
[1, 0, 0, 0]
, numberOfVertices=4, degree=12
}
```

```
Complement of Graph
Graph{
adjacencyMatrix=
[1, 0, 1, 1]
[0, 1, 1, 1]
[1, 1, 1, 1]
[1, 1, 1, 1]
, numberOfVertices=4, degree=28
}
```

(d) Ring Sum of 2 Graphs

(e) Complement of 2 Graphs

Figure 2: Input and Output for Program 2

3 Program to Find Minimum Spanning Tree Using Prim's Algorithm

3.1 Code

```
#include <bits/stdc++.h>
using namespace std;

int size;

int minKey(int key[], bool mstSet[]) {
    int min = INT_MAX, min_index;

    for (int v = 0; v < size; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

void printMST(int parent[], vector<vector<int>>& graph) {
    cout << "Edge \tWeight\n";
    for (int i = 0; i < graph.size() - 1; i++)
        cout << parent[i] << " - " << i << " \t" << graph[i][parent[i]] << "\n";
}

void primMST(vector<vector<int>>& graph) {
    size = graph.size();
    int parent[graph.size()];
    bool mstSet[graph.size()];
    int key[graph.size()];

    for (int i = 0; i < graph.size(); i++) {
        mstSet[i] = false;
        key[i] = INT_MAX;
    }

    key[0] = 0;
    parent[0] = -1;
    for (int c = 0; c < graph.size() - 1; c++) {
        int u = minKey(key, mstSet);
        mstSet[u] = true;
        for (int v = 0; v < graph.size(); v++) {
            if (graph[u][v] && key[v] > graph[u][v] && u != v) {
                parent[v] = u, key[v] = graph[u][v];
            }
        }
    }
    printMST(parent, graph);
}

int main() {
#ifndef ONLINE_JUDGE
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);
#endif
}
```

```

int vertex;
cin>>vertex;
vector<vector<int>> graph(vertex, vector<int>(vertex, 0));

for(int i=0;i<vertex;i++){
    for(int j=0;j<vertex;j++){
        cin>>graph[i][j];
    }
}

//Parallel Edges
for(int i=0;i<graph.size();i++){
    for(int j=0;j<graph.size();j++){
        graph[i][j] = min(graph[i][j], graph[j][i]);
    }
}

primMST(graph);

return 0;
}

```

3.2 Input/Output

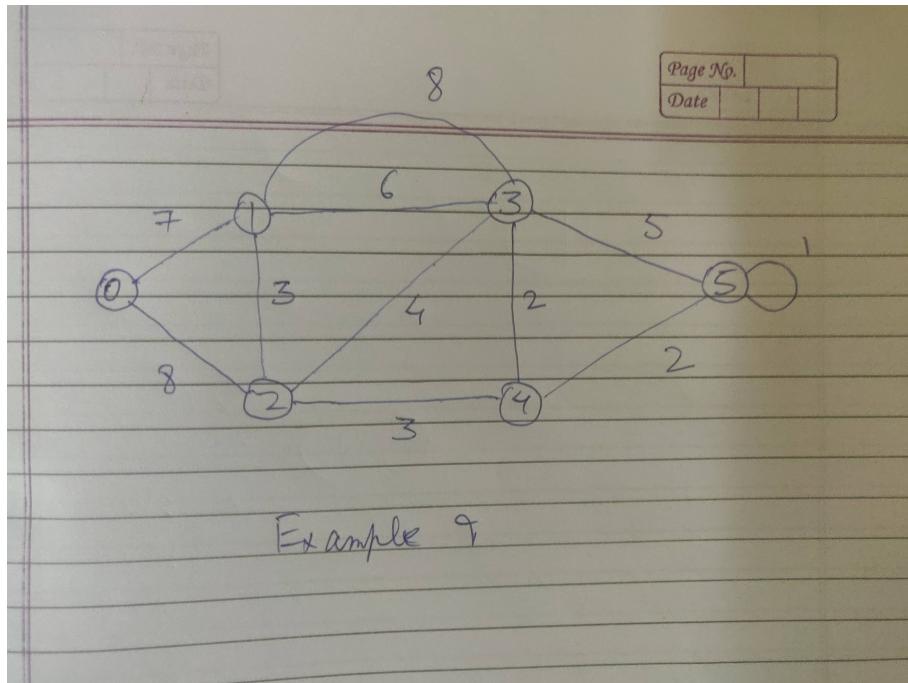


Figure 3: Un-directed Graph with Parallel Edges ad Self Loops Used In Prim's Algorithm

```
The Original Graph
Graph{order=6, size=11}
Edge Sum: 49
Vertex{data=0, degree=2, edges= [1 2 ]}
Vertex{data=1, degree=4, edges= [2 0 3 3 ]}
Vertex{data=2, degree=4, edges= [4 3 1 0 ]}
Vertex{data=3, degree=5, edges= [2 4 1 5 1 ]}
Vertex{data=4, degree=3, edges= [2 3 5 ]}
Vertex{data=5, degree=4, edges= [4 3 5 ]}

The new Graph After Applying Prim's Algorithm
Graph{order=6, size=5}
Edge Sum: 17
Vertex{data=0, degree=1, edges= [1 ]}
Vertex{data=1, degree=2, edges= [0 2 ]}
Vertex{data=2, degree=2, edges= [4 1 ]}
Vertex{data=3, degree=1, edges= [4 ]}
Vertex{data=4, degree=3, edges= [5 2 3 ]}
Vertex{data=5, degree=1, edges= [4 ]}

Process finished with exit code 0
```

Figure 4: Output for Prim's Algorithm

4 Program to Find Minimum Spanning Tree Using Kruskal's Algorithm

4.1 Code

```
#include <bits/stdc++.h>
using namespace std;

// a structure to represent a
// weighted edge in graph
class Edge {
public:
    int src, dest, weight;
};

// a structure to represent a connected,
// undirected and weighted graph
class Graph {
public:

    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    // Since the graph is undirected, the edge
    // from src to dest is also edge from dest
    // to src. Both are counted as 1 edge here.
    Edge* edge;
};

// Creates a graph with V vertices and E edges
Graph* createGraph(int V, int E)
{
    Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;

    graph->edge = new Edge[E];

    return graph;
}

// A structure to represent a subset for union-find
class subset {
public:
    int parent;
    int rank;
};

// A utility function to find set of an element i
// (uses path compression technique)
int find(subset subsets[], int i)
{
    // find root and make root as parent of i
    // (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent

```

```

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

    return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high
    // rank tree (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as root and
    // increment its rank by one
    else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Compare two edges according to their weights.
// Used in qsort() for sorting an array of edges
int myComp(const void* a, const void* b)
{
    Edge* a1 = (Edge*)a;
    Edge* b1 = (Edge*)b;
    return a1->weight > b1->weight;
}

// The main function to construct MST using Kruskal's
// algorithm
void KruskalMST(Graph* graph)
{
    int V = graph->V;
    Edge result[V]; // This will store the resultant MST
    int e = 0; // An index variable, used for result[]
    int i = 0; // An index variable, used for sorted edges

    // Step 1: Sort all the edges in non-decreasing
    // order of their weight. If we are not allowed to
    // change the given graph, we can create a copy of
    // array of edges
    qsort(graph->edge, graph->E, sizeof(graph->edge[0]),
          myComp);

    // Allocate memory for creating V ssubsets
    subset* subsets = new subset[(V * sizeof(subset))];

    // Create V subsets with single elements
}

```

```

for (int v = 0; v < V; ++v)
{
    subsets[v].parent = v;
    subsets[v].rank = 0;
}

// Number of edges to be taken is equal to V-1
while (e < V - 1 && i < graph->E)
{
    // Step 2: Pick the smallest edge. And increment
    // the index for next iteration
    Edge next_edge = graph->edge[i++];

    int x = find(subsets, next_edge.src);
    int y = find(subsets, next_edge.dest);

    // If including this edge doesn't cause cycle,
    // include it in result and increment the index
    // of result for next edge
    if (x != y) {
        result[e++] = next_edge;
        Union(subsets, x, y);
    }
    // Else discard the next_edge
}

// print the contents of result[] to display the
// built MST
cout << "Following are the edges in the constructed "
      "MST\n";
int minimumCost = 0;
for (i = 0; i < e; ++i)
{
    cout << result[i].src << " -- " << result[i].dest
        << " == " << result[i].weight << endl;
    minimumCost = minimumCost + result[i].weight;
}
// return;
cout << "Minimum Cost Spanning Tree: " << minimumCost
    << endl;
}

// Driver code
int main()
{
/* Let us create following weighted graph
   10
   0-----1
   | \ |
   6| 5\ |15
   | \ |
   2-----3
   4 */
int V = 4; // Number of vertices in graph
int E = 5; // Number of edges in graph
Graph* graph = createGraph(V, E);
}

```

```
// add edge 0-1
graph->edge[0].src = 0;
graph->edge[0].dest = 1;
graph->edge[0].weight = 10;

// add edge 0-2
graph->edge[1].src = 0;
graph->edge[1].dest = 2;
graph->edge[1].weight = 6;

// add edge 0-3
graph->edge[2].src = 0;
graph->edge[2].dest = 3;
graph->edge[2].weight = 5;

// add edge 1-3
graph->edge[3].src = 1;
graph->edge[3].dest = 3;
graph->edge[3].weight = 15;

// add edge 2-3
graph->edge[4].src = 2;
graph->edge[4].dest = 3;
graph->edge[4].weight = 4;

// Function call
KruskalMST(graph);

return 0;
}
```

4.2 Input/Output

```
int V = 4; // Number of vertices in graph
int E = 5; // Number of edges in graph
Graph* graph = createGraph(V, E);

// add edge 0-1
graph->edge[0].src = 0;
graph->edge[0].dest = 1;
graph->edge[0].weight = 10;

// add edge 0-2
graph->edge[1].src = 0;
graph->edge[1].dest = 2;
graph->edge[1].weight = 6;

// add edge 0-3
graph->edge[2].src = 0;
graph->edge[2].dest = 3;
graph->edge[2].weight = 5;

// add edge 1-3
graph->edge[3].src = 1;
graph->edge[3].dest = 3;
graph->edge[3].weight = 15;

// add edge 2-3
graph->edge[4].src = 2;
graph->edge[4].dest = 3;
graph->edge[4].weight = 4;
```

Figure 5: Input for Minimum Spanning Tree Kruskal's Algorithm

```
Following are the edges in the constructed MST
0 -- 2 == 6
0 -- 1 == 10
0 -- 3 == 5
Minimum Cost Spanning Tree: 21
```

Figure 6: Output for Minimum Spanning Tree Kruskal's Algorithm

5 Program to find Shortest Path between 2 Vertices using Dijkstra Algorithm

5.1 Code

```
// A C++ program for Dijkstra's single source shortest path algorithm.  
// The program is for adjacency matrix representation of the graph  
  
#include <limits.h>  
#include <stdio.h>  
  
// Number of vertices in the graph  
#define V 9  
  
// A utility function to find the vertex with minimum distance value, from  
// the set of vertices not yet included in shortest path tree  
int minDistance(int dist[], bool sptSet[])  
{  
    // Initialize min value  
    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;  
}  
  
// A utility function to print the constructed distance array  
void printSolution(int dist[]){  
    printf("Vertex \t\t Distance from Source\n");  
    for (int i = 0; i < V; i++)  
        printf("%d \t\t %d\n", i, dist[i]);  
}  
  
// Function that implements Dijkstra's single source shortest path algorithm  
// for a graph represented using adjacency matrix representation  
void dijkstra(int graph[V][V], int src){  
    int dist[V]; // The output array. dist[i] will hold the shortest  
    // distance from src to i  
  
    bool sptSet[V]; // sptSet[i] will be true if vertex i is included in shortest  
    // path tree or shortest distance from src to i is finalized  
  
    // Initialize all distances as INFINITE and stpSet[] as false  
    for (int i = 0; i < V; i++)  
        dist[i] = INT_MAX, sptSet[i] = false;  
  
    // Distance of source vertex from itself is always 0  
    dist[src] = 0;  
  
    // Find shortest path for all vertices  
    for (int count = 0; count < V - 1; count++) {  
        // Pick the minimum distance vertex from the set of vertices not  
        // yet processed. u is always equal to src in the first iteration.
```

```

int u = minDistance(dist, sptSet);

// Mark the picked vertex as processed
sptSet[u] = true;

// Update dist value of the adjacent vertices of the picked vertex.
for (int v = 0; v < V; v++)

    // Update dist[v] only if is not in sptSet, there is an edge from
    // u to v, and total weight of path from src to v through u is
    // smaller than current value of dist[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];
}

// print the constructed distance array
printSolution(dist);
}

// driver program to test above function
int main() {
    /* Let us create the example graph discussed above */
    int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                        { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                        { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                        { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                        { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                        { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                        { 0, 0, 0, 0, 2, 0, 1, 6 },
                        { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                        { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

    dijkstra(graph, 0);
    return 0;
}

```

5.2 Input/Output

```
{  
    { 0, 4, 0, 0, 0, 0, 0, 8, 0 },  
    { 4, 0, 8, 0, 0, 0, 0, 11, 0 },  
    { 0, 8, 0, 7, 0, 4, 0, 0, 2 },  
    { 0, 0, 7, 0, 9, 14, 0, 0, 0 },  
    { 0, 0, 0, 9, 0, 10, 0, 0, 0 },  
    { 0, 0, 4, 14, 10, 0, 2, 0, 0 },  
    { 0, 0, 0, 0, 0, 2, 0, 1, 6 },  
    { 8, 11, 0, 0, 0, 0, 1, 0, 7 },  
    { 0, 0, 2, 0, 0, 0, 6, 7, 0 }  
}
```

Figure 7: Input for Dijkstra's Algorithm

Vertex	Distance from Source
0	0
1	4
2	12
3	19
4	21
5	11
6	9
7	8
8	14

Figure 8: Output for Dijkstra's Algorithm

6 Program to Find Shortest Path Between Every Pair of vertices in a Graph Using Floyd-Warshall's Algorithm

6.1 Code

```
// C++ Program for Floyd Warshall Algorithm
#include <bits/stdc++.h>
using namespace std;

// Number of vertices in the graph
#define V 4

/* Define Infinite as a large enough
value. This value will be used for
vertices not connected to each other */
#define INF 99999

// A function to print the solution matrix
void printSolution(int dist[][]);

// Solves the all-pairs shortest path
// problem using Floyd Warshall algorithm
void floydWarshall (int graph[][])
{
    /* dist[][] will be the output matrix
    that will finally have the shortest
    distances between every pair of vertices */
    int dist[V][V], i, j, k;

    /* Initialize the solution matrix same
    as input graph matrix. Or we can say
    the initial values of shortest distances
    are based on shortest paths considering
    no intermediate vertex. */
    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            dist[i][j] = graph[i][j];

    /* Add all vertices one by one to
    the set of intermediate vertices.
    ----> Before start of an iteration,
    we have shortest distances between all
    pairs of vertices such that the
    shortest distances consider only the
    vertices in set {0, 1, 2, .. k-1} as
    intermediate vertices.
    ----> After the end of an iteration,
    vertex no. k is added to the set of
    intermediate vertices and the set becomes {0, 1, 2, .. k} */
    for (k = 0; k < V; k++)
    {
        // Pick all vertices as source one by one
        for (i = 0; i < V; i++)
        {
            // Pick all vertices as destination for the
            // above picked source
```

```

        for (j = 0; j < V; j++)
        {
            // If vertex k is on the shortest path from
            // i to j, then update the value of dist[i][j]
            if (dist[i][k] + dist[k][j] < dist[i][j])
                dist[i][j] = dist[i][k] + dist[k][j];
        }
    }

    // Print the shortest distance matrix
    printSolution(dist);
}

/* A utility function to print solution */
void printSolution(int dist[][])
{
    cout<<"The following matrix shows the shortest distances"
        " between every pair of vertices \n";
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            if (dist[i][j] == INF)
                cout<<"INF" << " ";
            else
                cout<<dist[i][j]<< " ";
        }
        cout<<endl;
    }
}

// Driver code
int main()
{
    /* Let us create the following weighted graph
       10
       (0)----->(3)
          |   /|\
          5 |   |
          |   | 1
         \|/   |
       (1)----->(2)
          3   */
    int graph[V][V] = { {0, 5, INF, 10},
                        {INF, 0, 3, INF},
                        {INF, INF, 0, 1},
                        {INF, INF, INF, 0}
                      };

    // Print the solution
    floydWarshall(graph);
    return 0;
}

```

6.2 Input/Output

```
{  
    {0, 5, INF, 10},  
    {INF, 0, 3, INF},  
    {INF, INF, 0, 1},  
    {INF, INF, INF, 0}  
}
```

Figure 9: Input for Floyd-Warshall's Algorithm

```
The following matrix shows the shortest distances between every pair of vertices  
0 5 8 9  
INF 0 3 4  
INF INF 0 1  
INF INF INF 0
```

Figure 10: Output for Floyd-Warshall's Algorithm

7 Program to find Shortest Path between Every Pair of Vertices using Bellman Ford's Algorithm

7.1 Code

```
// A C++ program for Bellman-Ford's single source
// shortest path algorithm.
#include <bits/stdc++.h>

// a structure to represent a weighted edge in graph
struct Edge {
    int src, dest, weight;
};

// a structure to represent a connected, directed and
// weighted graph
struct Graph {
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
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;
}

// A utility function used to print the solution
void printArr(int dist[], int n)
{
    printf("Vertex Distance from Source\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

// The main function that finds shortest distances from src to
// all other vertices using Bellman-Ford algorithm. The function
// also detects negative weight cycle
void BellmanFord(struct Graph* graph, int src)
{
    int V = graph->V;
    int E = graph->E;
    int dist[V];

    // Step 1: Initialize distances from src to all other vertices
    // as INFINITE
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX;
    dist[src] = 0;
```

```

// Step 2: Relax all edges |V| - 1 times. A simple shortest
// path from src to any other vertex can have at-most |V| - 1
// edges
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;
    }
}

// Step 3: check for negative-weight cycles. The above step
// guarantees shortest distances if graph doesn't contain
// negative weight cycle. If we get a shorter path, then there
// is a cycle.
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; // If negative cycle is detected, simply return
    }
}

printArr(dist, V);

return;
}

// Driver program to test above functions
int main()
{
    /* Let us create the graph given in above example */
    int V = 7; // Number of vertices in graph
    int E = 10; // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1 (or A-B in above figure)
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = 6;

    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 5;

    graph->edge[2].src = 0;
    graph->edge[2].dest = 3;
    graph->edge[2].weight = 5;

    graph->edge[3].src = 1;
    graph->edge[3].dest = 4;
}

```

```
graph->edge[3].weight = -1;

graph->edge[4].src = 2;
graph->edge[4].dest = 1;
graph->edge[4].weight = -2;

graph->edge[5].src = 2;
graph->edge[5].dest = 4;
graph->edge[5].weight = 1;

graph->edge[6].src = 3;
graph->edge[6].dest = 2;
graph->edge[6].weight = -2;

graph->edge[7].src = 3;
graph->edge[7].dest = 5;
graph->edge[7].weight = -1;

graph->edge[8].src = 4;
graph->edge[8].dest = 6;
graph->edge[8].weight = 3;

graph->edge[9].src = 5;
graph->edge[9].dest = 6;
graph->edge[9].weight = 3;

BellmanFord(graph, 0);

return 0;
}
```

7.2 Input/Output

```

int V = 7; // Number of vertices in graph
int E = 10; // Number of edges in graph
struct Graph* graph = createGraph(V, E);

// add edge 0-1 (or A-B in above figure)
graph->edge[0].src = 0;
graph->edge[0].dest = 1;
graph->edge[0].weight = 6;

graph->edge[1].src = 0;
graph->edge[1].dest = 2;
graph->edge[1].weight = 5;

graph->edge[2].src = 0;
graph->edge[2].dest = 3;
graph->edge[2].weight = 5;

graph->edge[3].src = 1;
graph->edge[3].dest = 4;
graph->edge[3].weight = -1;

graph->edge[4].src = 2;
graph->edge[4].dest = 1;
graph->edge[4].weight = -2;

graph->edge[5].src = 2;
graph->edge[5].dest = 4;
graph->edge[5].weight = 1;

graph->edge[6].src = 3;
graph->edge[6].dest = 2;
graph->edge[6].weight = -2;

graph->edge[7].src = 3;
graph->edge[7].dest = 5;
graph->edge[7].weight = -1;

graph->edge[8].src = 4;
graph->edge[8].dest = 6;
graph->edge[8].weight = 3;

graph->edge[9].src = 5;
graph->edge[9].dest = 6;
graph->edge[9].weight = 3;

```

Figure 11: Input for Bellman Ford's Algorithm

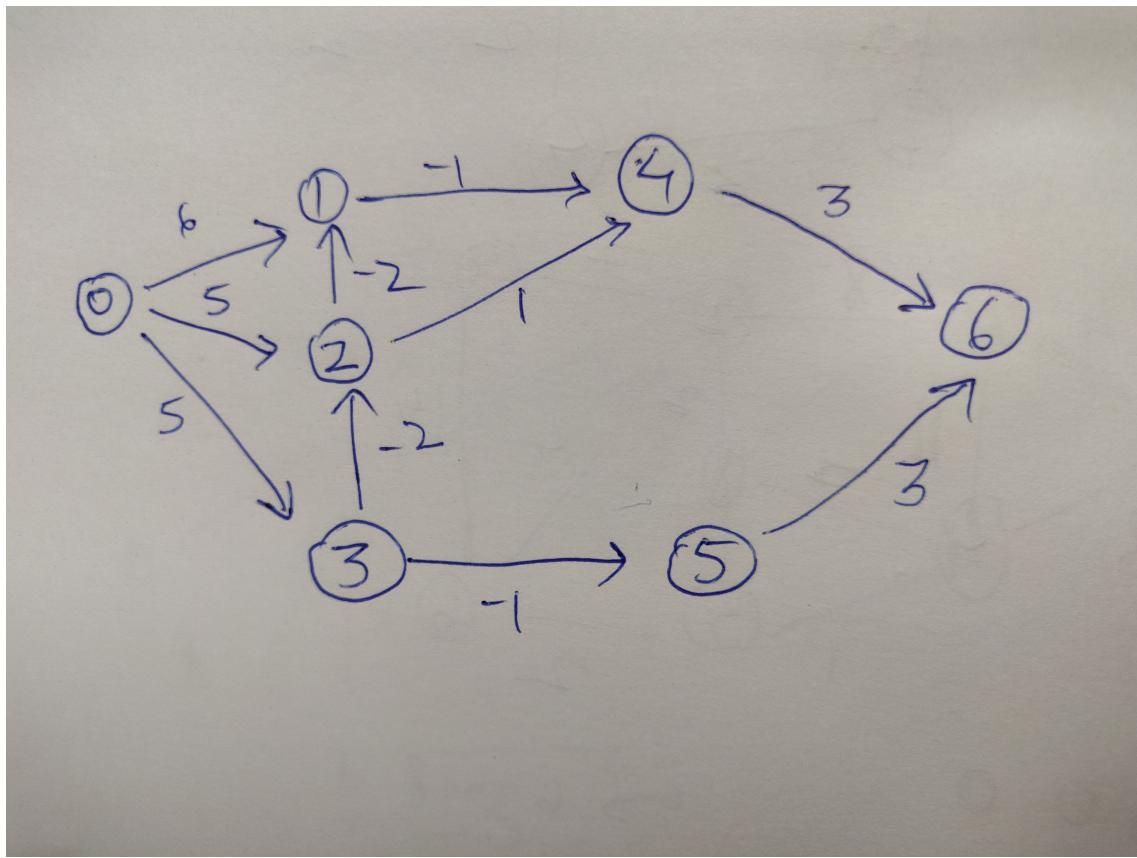


Figure 12: Directed Weighted Graph Taken As Input For Bellman Ford's Algorithm

Vertex	Distance from Source
0	0
1	1
2	3
3	5
4	0
5	4
6	3

Figure 13: Output for Bellman Ford's Algorithm

8 Program For Finding Maximal Matching for Bipartite Graph

8.1 Code

```
// A C++ program to find maximal
// Bipartite matching.
#include <iostream>
#include <string.h>
using namespace std;

// M is number of applicants
// and N is number of jobs
#define M 6
#define N 6

// A DFS based recursive function
// that returns true if a matching
// for vertex u is possible
bool bpm(bool bpGraph[M][N], int u,
         bool seen[], int matchR[])
{
    // Try every job one by one
    for (int v = 0; v < N; v++)
    {
        // If applicant u is interested in
        // job v and v is not visited
        if (bpGraph[u][v] && !seen[v])
        {
            // Mark v as visited
            seen[v] = true;

            // If job 'v' is not assigned to an
            // applicant OR previously assigned
            // applicant for job v (which is matchR[v])
            // has an alternate job available.
            // Since v is marked as visited in
            // the above line, matchR[v] in the following
            // recursive call will not get job 'v' again
            if (matchR[v] < 0 || bpm(bpGraph, matchR[v],
                                      seen, matchR))
            {
                matchR[v] = u;
                return true;
            }
        }
    }
    return false;
}

// Returns maximum number
// of matching from M to N
int maxBPM(bool bpGraph[M][N])
{
    // An array to keep track of the
    // applicants assigned to jobs.
    // The value of matchR[i] is the
```

```

// applicant number assigned to job i,
// the value -1 indicates nobody is
// assigned.
int matchR[N];

// Initially all jobs are available
memset(matchR, -1, sizeof(matchR));

// Count of jobs assigned to applicants
int result = 0;
for (int u = 0; u < M; u++)
{
    // Mark all jobs as not seen
    // for next applicant.
    bool seen[N];
    memset(seen, 0, sizeof(seen));

    // Find if the applicant 'u' can get a job
    if (bpm(bpGraph, u, seen, matchR))
        result++;
}
return result;
}

// Driver Code
int main()
{
    // Let us create a bpGraph
    // shown in the above example
    bool bpGraph[M][N] = {{0, 0, 0, 1, 1},
                          {0, 0, 0, 1, 1},
                          {0, 0, 0, 0, 1},
                          {0, 0, 0, 0, 0},
                          {0, 0, 0, 0, 0}};

    cout << "Maximum number of applicants that can get job is "
         << maxBPM(bpGraph);

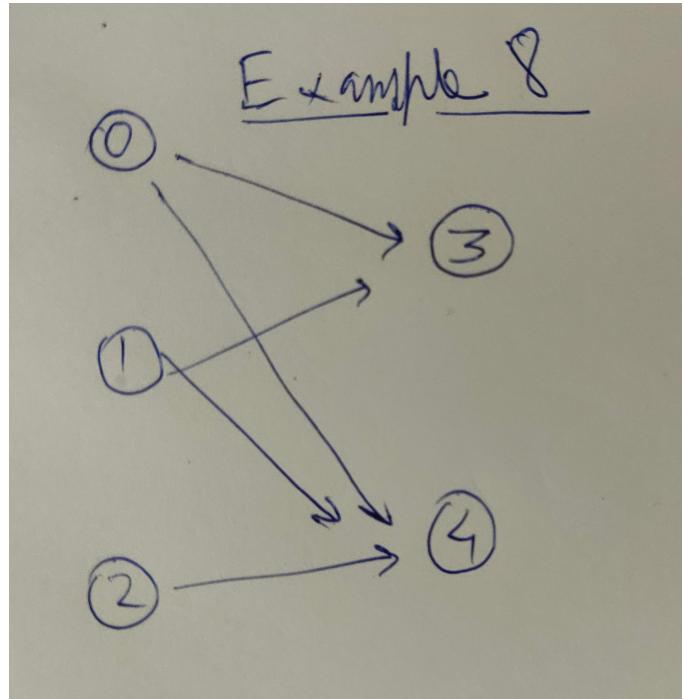
    return 0;
}

```

8.2 Input/Output

```
{  
    {0, 0, 0, 1, 1},  
    {0, 0, 0, 1, 1},  
    {0, 0, 0, 0, 1},  
    {0, 0, 0, 0, 0},  
    {0, 0, 0, 0, 0}  
}
```

Figure 14: Input for Maximum Matching for Bipartite Graph Algorithm



(a) Directed Un-weighted Graph Taken As Input For Maximum Matching for Bipartite Graph Algorithm

Figure 15: Input for Program 8

```
Maximum number of applicants that can get job is 2
```

Figure 16: Output for Maximum Matching for Bipartite Graph Algorithm

9 Program For Finding Maximal Matching for General Path

9.1 Code

```
// C++ implementation of Hopcroft Karp algorithm for
// maximum matching
#include<bits/stdc++.h>
using namespace std;
#define NIL 0
#define INF INT_MAX

// A class to represent Bipartite graph for Hopcroft
// Karp implementation
class BipGraph {
    // m and n are number of vertices on left
    // and right sides of Bipartite Graph
    int m, n;
    // adj[u] stores adjacents of left side
    // vertex 'u'. The value of u ranges from 1 to m.
    // 0 is used for dummy vertex
    list<int> *adj;
    // These are basically pointers to arrays needed
    // for hopcroftKarp()
    int *pairU, *pairV, *dist;
public:
    BipGraph(int m, int n); // Constructor
    void addEdge(int u, int v); // To add edge
    // Returns true if there is an augmenting path
    bool bfs();
    // Adds augmenting path if there is one beginning
    // with u
    bool dfs(int u);
    // Returns size of maximum matcing
    int hopcroftKarp();
};

// Returns size of maximum matching
int BipGraph::hopcroftKarp() {
    // pairU[u] stores pair of u in matching where u
    // is a vertex on left side of Bipartite Graph.
    // If u doesn't have any pair, then pairU[u] is NIL
    pairU = new int[m+1];
    // pairV[v] stores pair of v in matching. If v
    // doesn't have any pair, then pairU[v] is NIL
    pairV = new int[n+1];
    // dist[u] stores distance of left side vertices
    // dist[u] is one more than dist[u'] if u is next
    // to u' in augmenting path
    dist = new int[m+1];

    // Initialize NIL as pair of all vertices
    for (int u=0; u<m; u++)
        pairU[u] = NIL;
    for (int v=0; v<n; v++)
        pairV[v] = NIL;
```

```

// Initialize result
int result = 0;
// Keep updating the result while there is an
// augmenting path.
while (bfs()) {
    // Find a free vertex
    for (int u=1; u<=m; u++)
        // If current vertex is free and there is
        // an augmenting path from current vertex
        if (pairU[u]==NIL && dfs(u))
            result++;
}
return result;
}

// Returns true if there is an augmenting path, else returns
// false
bool BipGraph::bfs() {
    queue<int> Q; //an integer queue
    // First layer of vertices (set distance as 0)
    for (int u=1; u<=m; u++) {
        // If this is a free vertex, add it to queue
        if (pairU[u]==NIL) {
            // u is not matched
            dist[u] = 0;
            Q.push(u);
        }
        // Else set distance as infinite so that this vertex
        // is considered next time
        else dist[u] = INF;
    }

    // Initialize distance to NIL as infinite
    dist[NIL] = INF;
    // Q is going to contain vertices of left side only.
    while (!Q.empty()) {
        // Dequeue a vertex
        int u = Q.front();
        Q.pop();
        // If this node is not NIL and can provide a shorter path to NIL
        if (dist[u] < dist[NIL]) {
            // Get all adjacent vertices of the dequeued vertex u
            list<int>::iterator i;
            for (i=adj[u].begin(); i!=adj[u].end(); ++i) {
                int v = *i;
                // If pair of v is not considered so far
                // (v, pairV[V]) is not yet explored edge.
                if (dist[pairV[v]] == INF) {
                    // Consider the pair and add it to queue
                    dist[pairV[v]] = dist[u] + 1;
                    Q.push(pairV[v]);
                }
            }
        }
    }
}

```

```

// If we could come back to NIL using alternating path of distinct
// vertices then there is an augmenting path
return (dist[NIL] != INF);
}

// Returns true if there is an augmenting path beginning with free vertex u
bool BipGraph::dfs(int u) {
    if (u != NIL) {
        list<int>::iterator i;
        for (i=adj[u].begin(); i!=adj[u].end(); ++i) {
            // Adjacent to u
            int v = *i;
            // Follow the distances set by BFS
            if (dist[pairV[v]] == dist[u]+1) {
                // If dfs for pair of v also returns
                // true
                if (dfs(pairV[v]) == true) {
                    pairV[v] = u;
                    pairU[u] = v;
                    return true;
                }
            }
        }
    }

    // If there is no augmenting path beginning with u.
    dist[u] = INF;
    return false;
}
return true;
}

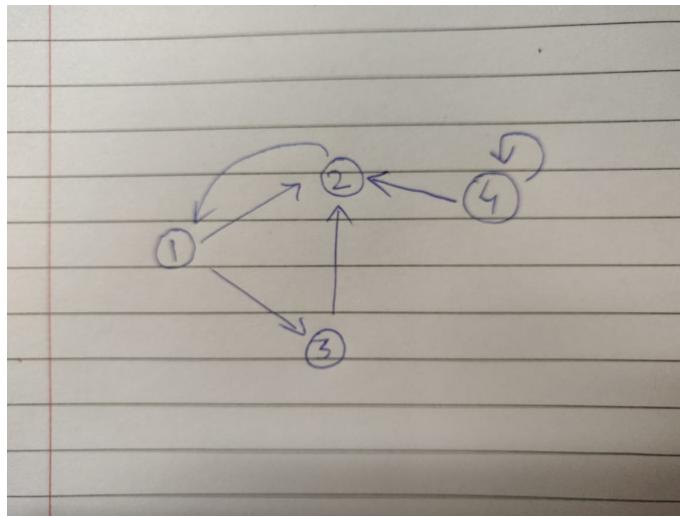
// Constructor
BipGraph::BipGraph(int m, int n) {
    this->m = m;
    this->n = n;
    adj = new list<int>[m+1];
}

// To add edge from u to v and v to u
void BipGraph::addEdge(int u, int v) {
    adj[u].push_back(v); // Add u to v's list.
}

// Driver Program
int main() {
    BipGraph g(4, 4);
    g.addEdge(1, 2);
    g.addEdge(1, 3);
    g.addEdge(2, 1);
    g.addEdge(3, 2);
    g.addEdge(4, 2);
    g.addEdge(4, 4);
    cout << "Size of maximum matching is " << g.hopcroftKarp();
    return 0;
}

```

9.2 Input/Output



(a) Directed Un-weighted Graph Taken As Input For Maximal Matching for General Graph Algorithm

Figure 17: Input for Program 9

```
Size of Maximum Matching is 4
```

Figure 18: Output for Maximal Matching for General Path Algorithm

10 Program to Find Maximum Flow From Source Node to Sink Node Using Ford-Fulkerson Algorithm

10.1 Code

```
// C++ program for implementation of Ford Fulkerson algorithm
#include <iostream>
#include <limits.h>
#include <string.h>
#include <queue>
using namespace std;

// Number of vertices in given graph
#define V 6

/* Returns true if there is a path from source 's' to sink 't' in
residual graph. Also fills parent[] to store the path */
bool bfs(int rGraph[V][V], int s, int t, int parent[])
{
    // Create a visited array and mark all vertices as not visited
    bool visited[V];
    memset(visited, 0, sizeof(visited));

    // Create a queue, enqueue source vertex and mark source vertex
    // as visited
    queue <int> q;
    q.push(s);
    visited[s] = true;
    parent[s] = -1;

    // Standard BFS Loop
    while (!q.empty())
    {
        int u = q.front();
        q.pop();

        for (int v=0; v<V; v++)
        {
            if (visited[v]==false && rGraph[u][v] > 0)
            {
                q.push(v);
                parent[v] = u;
                visited[v] = true;
            }
        }
    }

    // If we reached sink in BFS starting from source, then return
    // true, else false
    return (visited[t] == true);
}

// Returns the maximum flow from s to t in the given graph
int fordFulkerson(int graph[V][V], int s, int t)
{
    int u, v;
```

```

// Create a residual graph and fill the residual graph with
// given capacities in the original graph as residual capacities
// in residual graph
int rGraph[V][V]; // Residual graph where rGraph[i][j] indicates
    // residual capacity of edge from i to j (if there
    // is an edge. If rGraph[i][j] is 0, then there is not)
for (u = 0; u < V; u++)
    for (v = 0; v < V; v++)
        rGraph[u][v] = graph[u][v];

int parent[V]; // This array is filled by BFS and to store path

int max_flow = 0; // There is no flow initially

// Augment the flow while there is path from source to sink
while (bfs(rGraph, s, t, parent))
{
    // Find minimum residual capacity of the edges along the
    // path filled by BFS. Or we can say find the maximum flow
    // through the path found.
    int path_flow = INT_MAX;
    for (v=t; v!=s; v=parent[v])
    {
        u = parent[v];
        path_flow = min(path_flow, rGraph[u][v]);
    }

    // update residual capacities of the edges and reverse edges
    // along the path
    for (v=t; v != s; v=parent[v])
    {
        u = parent[v];
        rGraph[u][v] -= path_flow;
        rGraph[v][u] += path_flow;
    }

    // Add path flow to overall flow
    max_flow += path_flow;
}

// Return the overall flow
return max_flow;
}

// Driver program to test above functions
int main()
{
    // Let us create a graph shown in the above example
    int graph[V][V] = {
        {0, 3, 0, 3, 0, 0, 0},
        {0, 0, 4, 0, 0, 0, 0},
        {0, 0, 0, 1, 2, 0, 0},
        {0, 0, 0, 0, 2, 6, 0},
        {0, 1, 0, 0, 0, 0, 1},
        {0, 0, 0, 0, 0, 0, 9},
}

```

```
    {0, 0, 0, 0, 0, 0, 0}
};

cout << "The maximum possible flow is " << fordFulkerson(graph, 0, 5);

return 0;
}
```

10.2 Input/Output

```
int graph[V][V] = {  
    {0, 3, 0, 3, 0, 0, 0},  
    {0, 0, 4, 0, 0, 0, 0},  
    {0, 0, 0, 1, 2, 0, 0},  
    {0, 0, 0, 0, 2, 6, 0},  
    {0, 1, 0, 0, 0, 0, 1},  
    {0, 0, 0, 0, 0, 0, 9},  
    {0, 0, 0, 0, 0, 0, 0}  
};
```

Figure 19: Input for Maximum Matching for Bipartite Graph Algorithm

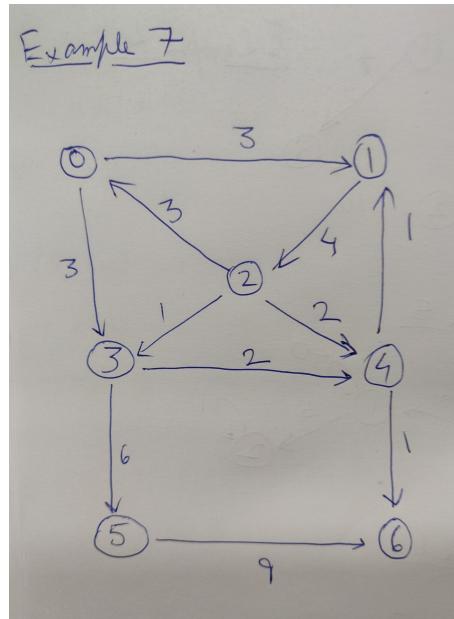


Figure 20: Directed Weighted Graph Taken As Input For Ford-Fulkerson Algorithm

```
The maximum possible flow is 4
```

Figure 21: Output for Ford-Fulkerson Algorithm