GRAPH THEORY

MC405 Lab

ANEESH PANCHAL

2K20/MC/21

Department of Applied Mathematics,

Delhi Technological University

Submitted To –

Ms. Payal

and Mr. Kishore Kumar

INDEX

S. No.	Experiment	Date	Sign & Remark
01.	Program to find the number of vertices, even vertices, odd vertices and number of edges in a graph.	17/08/2023	
02.	Program to find union, intersection and ring-sum of two graphs.	24/08/2023	
03.	Program to find shortest path between two vertices in a graph using Dijkstra's algorithm.	31/08/2023	
04.	Program to find shortest path between every pair of vertices in a graph using Floyd-Warshall's algorithm.	14/09/2023	
05.	Program to find shortest path between two vertices in a graph using Bellman-Ford's algorithm.	14/09/2023	
06.	Program to find minimal spanning tree of a graph using Prim's algorithm.	12/10/2023	
07.	Program to find minimal spanning tree of a graph using Kruskal's algorithm.	12/10/2023	
08.	Program to maximum flow from source node to sink node using Ford-Fulkerson Algorithm.	19/10/2023	
09.	Program to find maximum matching for bipartite graph.	02/11/2023	
10.	Program to find maximum matching for general graph.	02/11/2023	

Experiment 1

Aim:

Program to find the number of vertices, even vertices, odd vertices and number of edges in a graph.

Theory:

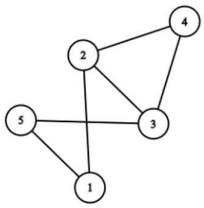
In graph theory, a graph is made up of vertices and edges. The number of vertices in a graph is simply the count of unique nodes. The number of edges is the count of unique links between nodes.

The degree of a vertex is the number of edges incident on it. Vertices can be classified into even or odd vertices based on their degrees:

1. An even vertex has an even degree.
2. An odd vertex has an odd degree.

Graph Considered:

Number of vertices = 5
Even vertices = 3
Odd vertices = 2
Number of Edges = 6



Code:

```
// Aneesh Panchal 2K20/MC/21

#include<iostream>
#include<vector>
using namespace std;

void find_degree(vector<vector<int>> adjacency){
    int len = adjacency.size();
    vector<int> edges(len,0);
    for(int i=0;i<len;i++)
        for(int j=0;j<len;j++)
            edges[i] = edges[i] + adjacency[i][j];
    vector<int> solution = {0,0,0};
    int tot_edges = 0;
    for(int i=0;i<len;i++){
        tot_edges = tot_edges + edges[i];
        if(edges[i]%2!=0)
            ++solution[0];
        else
            ++solution[1];
    }
    solution[2] = solution[0]+solution[1];
    int total_edges = tot_edges/2;
    cout<<endl<<"Odd vertices: "<<solution[0]<<endl;
    cout<<"Even vertices: "<<solution[1]<<endl;
    cout<<"Total vertices: "<<solution[2]<<endl<<endl;
    cout<<"Total Number of Edges: "<<total_edges<<endl<<endl;
}
```

```
int main(){
    vector<vector<int>> adjacency =
{{0,1,0,0,1},{1,0,1,1,0},{0,1,0,1,1},{0,1,1,0,0},{1,0,1,0,0}};
    for(int i=0;i<adjacency.size();i++){
        for(int j=0;j<adjacency.size();j++)
            cout<<adjacency[i][j]<<" ";
        cout<<endl;
    }
    find_degree(adjacency);
    return 0;
}
```

Output:

```
PS E:\Codes\MC405 Graph Theory> cd "e:\Codes\MC405 Graph Theory\" ; if ($?) { g++ Exp_1.cpp -o Exp_1 } ; if ($?) { .\Exp_1 }
0 1 0 0 1
1 0 1 1 0
0 1 0 1 1
0 1 1 0 0
1 0 1 0 0

Odd vertices: 2
Even vertices: 3
Total vertices: 5

Total Number of Edges: 6

PS E:\Codes\MC405 Graph Theory>
```

Experiment 2

Aim:

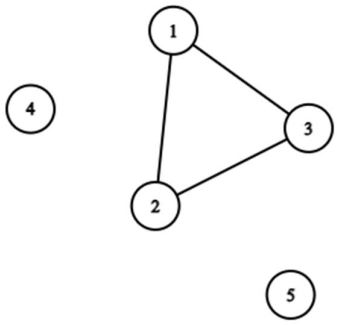
Program to find union, intersection and ring-sum of two graphs.

Theory:

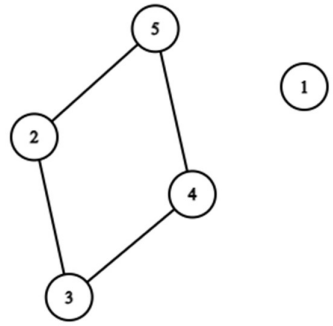
A graph G is typically defined by an ordered pair G = (V, E) where V is a set of vertices and E is a set of edges. Each edge connects two vertices. The graph can be either directed or undirected.

1. *Union of 2 graphs:*
   For 2 graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ the union of the graphs is given by,
   $$G_1 \cup G_2 = (V_1 \cup V_2, E_1 \cup E_2)$$
2. *Intersection of 2 graphs:*
   For 2 graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ the intersection of the graphs is given by,
   $$G_1 \cap G_2 = (V_1 \cup V_2, E_1 \cap E_2)$$
3. *Ring Sum of 2 graphs:*
   For 2 graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ the ring sum of the graphs is given by,
   $$G_1 \oplus G_2 = (V_1 \cup V_2, E_1 \oplus E_2)$$

Graphs Considered:



Graph **I**



Graph **II**

Code:

```
// Aneesh Panchal 2K20/MC/21

#include<iostream>
#include<vector>
using namespace std;

void union_intersection(vector<vector<int>> graph_one, vector<vector<int>> graph_two){
    int len = graph_one.size();
    vector<vector<int>> union_graph(len,vector<int>(len,0));
    vector<vector<int>> intersection_graph(len,vector<int>(len,0));
    vector<vector<int>> ring_sum(len,vector<int>(len,0));
    for(int i=0;i<len;i++){
        for(int j=0;j<len;j++){
```

```
            if(graph_one[i][j]==graph_two[i][j]){
                intersection_graph[i][j] = graph_one[i][j];
                union_graph[i][j] = graph_one[i][j];
            }
            else{
                if(graph_one[i][j]==1 || graph_two[i][j]==1)
                    union_graph[i][j] = 1;
                intersection_graph[i][j] = 0;
            }
            ring_sum[i][j] = union_graph[i][j]-intersection_graph[i][j];
        }
    }
    cout<<"Union Graph: "<<endl;
    for(int i=0;i<union_graph.size();i++){
        for(int j=0;j<union_graph.size();j++)
            cout<<union_graph[i][j]<<" ";
        cout<<endl;
    }
    cout<<endl;
    cout<<"Intersection Graph: "<<endl;
    for(int i=0;i<intersection_graph.size();i++){
        for(int j=0;j<intersection_graph.size();j++)
            cout<<intersection_graph[i][j]<<" ";
        cout<<endl;
    }
    cout<<endl;
    cout<<"Ring Sum Graph: "<<endl;
    for(int i=0;i<ring_sum.size();i++){
        for(int j=0;j<ring_sum.size();j++)
            cout<<ring_sum[i][j]<<" ";
        cout<<endl;
    }
    cout<<endl;
}

int main(){
    vector<vector<int>> graph_one =
{{0,1,1,0,0},{1,0,1,0,0},{1,1,0,0,0},{0,0,0,0,0},{0,0,0,0,0}};
    vector<vector<int>> graph_two =
{{0,0,0,0,0},{0,0,1,0,1},{0,1,0,1,0},{0,0,1,0,1},{0,1,0,1,0}};
    cout<<"Graph I: "<<endl;
    for(int i=0;i<graph_one.size();i++){
        for(int j=0;j<graph_one.size();j++)
            cout<<graph_one[i][j]<<" ";
        cout<<endl;
    }
    cout<<endl;
    cout<<"Graph II: "<<endl;
    for(int i=0;i<graph_two.size();i++){
        for(int j=0;j<graph_two.size();j++)
            cout<<graph_two[i][j]<<" ";
```

```
        cout<<endl;
    }
    cout<<endl;
    union_intersection(graph_one,graph_two);
    return 0;
}
```

Output:

```
PS E:\Codes\MC405 Graph Theory> cd "e:\Codes\MC405 Graph Theory\" ; if ($?) { g++ Exp_2.cpp -o Exp_2 } ; if ($?) { .\Exp_2 }
Graph I:
0 1 1 0 0
1 0 1 0 0
1 1 0 0 0
0 0 0 0 0
0 0 0 0 0

Graph II:
0 0 0 0 0
0 0 1 0 1
0 1 0 1 0
0 0 1 0 1
0 1 0 1 0

Union Graph:
0 1 1 0 0
1 0 1 0 1
1 1 0 1 0
0 0 1 0 1
0 1 0 1 0

Intersection Graph:
0 0 0 0 0
0 0 1 0 0
0 1 0 0 0
0 0 0 0 0
0 0 0 0 0

Ring Sum Graph:
0 1 1 0 0
1 0 0 0 1
1 0 0 1 0
0 0 1 0 1
0 1 0 1 0

PS E:\Codes\MC405 Graph Theory>
```

Experiment 3

Aim:

Program to find shortest path between two vertices in a graph using Dijkstra's algorithm.

Theory:

Dijkstra's algorithm is a greedy algorithm that finds the shortest path from a starting vertex to all vertices in a weighted graph with non-negative edge weights. The algorithm keeps track of the shortest distance from the start vertex to every other vertex and updates these distances if a shorter path is found.

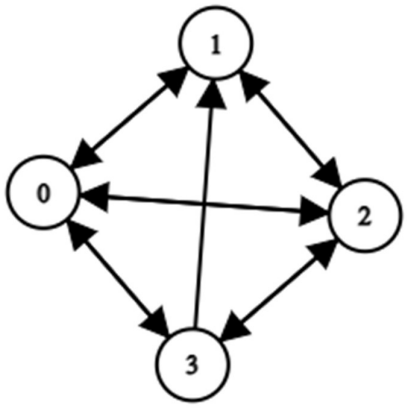
Properties: It is Greedy algorithm and works only for non-negative weights

Limitation: It will not work when negative weights exist in the graph.

Graph Considered:

Weight of the edges are as follows,

$\mathbf{0 \rightarrow 1 = 2}$
$\mathbf{0 \rightarrow 2 = 2}$
$\mathbf{0 \rightarrow 3 = 4}$
$\mathbf{1 \rightarrow 0 = 1}$
$\mathbf{1 \rightarrow 2 = 2}$
$\mathbf{2 \rightarrow 0 = 3}$
$\mathbf{2 \rightarrow 1 = 1}$
$\mathbf{2 \rightarrow 3 = 3}$
$\mathbf{3 \rightarrow 0 = 4}$
$\mathbf{3 \rightarrow 1 = 3}$
$\mathbf{3 \rightarrow 2 = 1}$



Code:

```
// Aneesh Panchal 2K20/MC/21

#include<iostream>
#include<vector>
#include<limits.h>
using namespace std;

int min_dist(vector<int> distance, vector<bool> visited){
    int len = distance.size();
    int min = INT_MAX;
    int index = INT_MAX;
    for(int i=0;i<len;i++)
        if(visited[i]==false && distance[i]<=min){
            min = distance[i];
            index = i;
        }
    return index;
}
```

```cpp
void Dijkstras(vector<vector<int>> adjacency, int source){
    int len = adjacency.size();
    vector<int> distance(len,INT_MAX);
    vector<bool> visited(len,false);
    distance[source] = 0;
    for(int count=0;count<len;count++){
        int min_index = min_dist(distance,visited);
        visited[min_index] = true;
        for(int i=0;i<len;i++)
            if(!visited[i] && adjacency[min_index][i] && distance[min_index]!=INT_MAX &&
distance[min_index]+adjacency[min_index][i]<distance[i])
                distance[i]=distance[min_index]+adjacency[min_index][i];
    }
    cout<<"Source Vertex: "<<source<<endl;
    for(int i=0;i<len;i++)
        cout<<"To vertex: "<<i<<" with Distance: "<<distance[i]<<endl;
    cout<<endl;
}


int main(){
    vector<vector<int>> adjacency = {{0,2,2,4},{1,0,2,0},{3,1,0,3},{4,3,1,0}};
    cout<<endl<<"Initial Adjacency Matrix: "<<endl;
    for(int i=0;i<adjacency.size();i++){
        for(int j=0;j<adjacency.size();j++)
            cout<<adjacency[i][j]<<" ";
        cout<<endl;
    }
    cout<<endl;
    Dijkstras(adjacency,1);
    return 0;
}
```

Output:

```
PS E:\Codes\MC405 Graph Theory> cd "e:\Codes\MC405 Graph Theory\" ; if ($?) { g++ Exp_3.cpp -o Exp_3 } ; if ($?) { .\Exp_3 }

Initial Adjacency Matrix:
0 2 2 4
1 0 2 0
3 1 0 3
4 3 1 0

Source Vertex: 1
To vertex: 0 with Distance: 1
To vertex: 1 with Distance: 0
To vertex: 2 with Distance: 2
To vertex: 3 with Distance: 5

PS E:\Codes\MC405 Graph Theory>
```

Experiment 4

Aim:

Program to find shortest path between every pair of vertices in a graph using Floyd-Warshall's algorithm.

Theory:

Floyd-Warshall algorithm addresses the all-pairs shortest path problem: determining the shortest paths between every pair of vertices in a weighted graph.

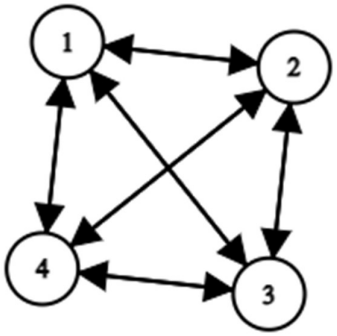
Properties: It uses Dynamic programming and allows negative edges.

Limitations: It will not work if negative weights cycles exist in graph.

Graph Considered:

Weight of the edges are as follows,

$\mathbf{1 \rightarrow 2 = 5}$
$\mathbf{1 \rightarrow 3 = \infty}$
$\mathbf{1 \rightarrow 4 = 10}$
$\mathbf{2 \rightarrow 1 = \infty}$
$\mathbf{2 \rightarrow 3 = 3}$
$\mathbf{2 \rightarrow 4 = \infty}$
$\mathbf{3 \rightarrow 1 = \infty}$
$\mathbf{3 \rightarrow 2 = \infty}$
$\mathbf{3 \rightarrow 4 = 1}$
$\mathbf{4 \rightarrow 1 = \infty}$
$\mathbf{4 \rightarrow 2 = \infty}$
$\mathbf{4 \rightarrow 3 = \infty}$



Code:

```
// Aneesh Panchal 2K20/MC/21

#include<iostream>
#include<vector>
#include<limits.h>
using namespace std;

void FloydWarshall(vector<vector<int>> adjacency){
    int len = adjacency.size();
    vector<vector<int>> temp = adjacency;
    vector<vector<int>> updated(len,vector<int>(len,0));
    for(int count=0;count<len;++count){
        for(int i=0;i<len;++i)
            for(int j=0;j<len;++j){
                if(i==count || j==count || temp[i][count]==INT_MAX ||
temp[count][j]==INT_MAX)
                    updated[i][j] = temp[i][j];
                else{
```

```
                    if(temp[i][count] + temp[count][j] < temp[i][j])
                        updated[i][j] = temp[i][count] + temp[count][j];
                    else
                        updated[i][j] = temp[i][j];
                }
            }
        temp = updated;
    }

    cout<<"Final Minimum Distance Matrix: "<<endl;
    for(int i=0;i<len;i++){
        for(int j=0;j<len;j++)
            if (temp[i][j]==INT_MAX)
                cout<<"inf"<<" ";
            else
                cout<<temp[i][j]<<" ";
    }
}

int main(){
    vector<vector<int>> adjacency =
{{0,5,INT_MAX,10},{INT_MAX,0,3,INT_MAX},{INT_MAX,INT_MAX,0,1},{INT_MAX,INT_MAX,INT_MAX,0}
};
    cout<<endl<<"Initial Adjacency Matrix: "<<endl;
    for(int i=0;i<adjacency.size();i++){
        for(int j=0;j<adjacency.size();j++){
            if(adjacency[i][j]==INT_MAX)
                cout<<"inf"<<" ";
            else
                cout<<adjacency[i][j]<<" ";
        }
        cout<<endl;
    }
    cout<<endl;
    FloydWarshall(adjacency);
    return 0;
}
```

Output:

```
PS E:\Codes\MC405 Graph Theory> cd "e:\Codes\MC405 Graph Theory\" ; if ($?) { g++ Exp_4.cpp -o Exp_4 } ; if ($?) { .\Exp_4 }

Initial Adjacency Matrix:
0 5 inf 10
inf 0 3 inf
inf inf 0 1
inf inf inf 0

Final Minimum Distance Matrix:
0 5 8 9
inf 0 3 4
inf inf 0 1
inf inf inf 0

PS E:\Codes\MC405 Graph Theory>
```

Experiment 5

Aim:

Program to find shortest path between two vertices in a graph using Bellman-Ford's algorithm.

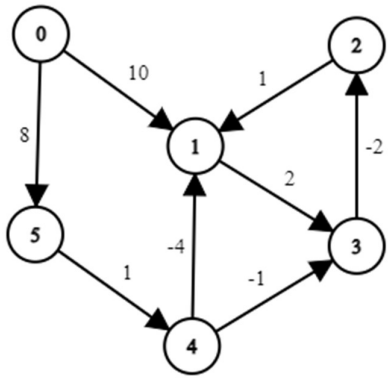
Theory:

The Bellman-Ford algorithm solves the single-source shortest path problem: determining the shortest path from a single source vertex to all other vertices in a weighted graph.

Properties: It is slower than Dijkstra's but it allows negative edges.

Limitations: It will not work if negative weights cycles exist in graph.

Graph Considered:



Code:

```
// Aneesh Panchal 2K20/MC/21

#include<iostream>
#include<vector>
#include<limits.h>
using namespace std;

void BellmanFord(vector<vector<int>> adjacency, int source){
    int len = adjacency.size();
    vector<int> distance(len,INT_MAX);
    distance[source] = 0;
    for(int i=0;i<len-1;i++)
        for(int j=0;j<len;j++)
            for(int k=0;k<len;k++)
                if(adjacency[j][k]!=0 && distance[j]!=INT_MAX &&
distance[j]+adjacency[j][k]<distance[k])
                    distance[k]=distance[j]+adjacency[j][k];
    cout<<"Final Distance Matrix: "<<endl;
    for(int i=0;i<len;i++)
        cout<<"To vertex: "<<i<<" with Distance: "<<distance[i]<<endl;
}
```

```
int main(){
    vector<vector<int>> adjacency = {{0,10,0,0,0,8}, {0,0,0,2,0,0},
                                     {0,1,0,0,0,0}, {0,0,-2,0,0,0},
                                     {0,-4,0,-1,0,0}, {0,0,0,0,1,0}};
    cout<<endl<<"Initial Adjacency Matrix: "<<endl;
    for(int i=0;i<adjacency.size();i++){
        for(int j=0;j<adjacency.size();j++)
            cout<<adjacency[i][j]<<" ";
        cout<<endl;
    }
    cout<<endl;
    int source = 0;
    BellmanFord(adjacency, source);
    cout<<endl;
    return 0;
}
```

Output:

```
PS E:\Codes\MC405 Graph Theory> cd "e:\Codes\MC405 Graph Theory\" ; if ($?) { g++ Exp_5.cpp -o Exp_5 } ; if ($?) { .\Exp_5 }

Initial Adjacency Matrix:
0 10 0 0 0 8
0 0 0 2 0 0
0 1 0 0 0 0
0 0 -2 0 0 0
0 -4 0 -1 0 0
0 0 0 0 1 0

Final Distance Matrix:
To vertex: 0 with Distance: 0
To vertex: 1 with Distance: 5
To vertex: 2 with Distance: 5
To vertex: 3 with Distance: 7
To vertex: 4 with Distance: 9
To vertex: 5 with Distance: 8

PS E:\Codes\MC405 Graph Theory>
```

Experiment 6

Aim:

Program to find minimal spanning tree of a graph using Prim's algorithm.

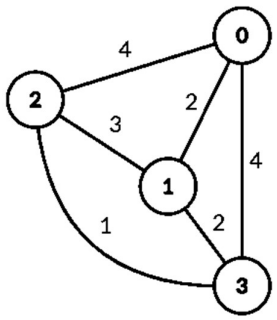
Theory:

Prim's algorithm is a greedy algorithm that finds the MST for a weighted undirected graph. The algorithm works by starting from an arbitrary node, then repeatedly adding the shortest edge that connects a vertex in the MST to one outside it.

Properties: It follows greedy approach.

Limitations: It will not work if negative weights cycles exist in graph or, if graph is disconnected.

Graph Considered:



Code:

```
// Aneesh Panchal 2K20/MC/21

#include<iostream>
#include<vector>
#include<limits.h>
using namespace std;

vector<int> indices(vector<vector<int>> adjacency, vector<bool> selected, int len){
    vector<int> index(2,len);
    int min = INT_MAX;
    for(int i=0;i<len;i++){
        for(int j=0;j<len;j++){
            if(adjacency[i][j]>0 && adjacency[i][j]<min && selected[i]==true &&
selected[j]==false){
                min = adjacency[i][j];
                index[0] = i;
                index[1] = j;
            }
        }
    }
    return index;
}
```

```
vector<int> initial(vector<vector<int>> adjacency, int len){
    vector<int> index(2,len);
    int min = INT_MAX;
    for(int i=0;i<len;i++){
        for(int j=0;j<len;j++){
            if(adjacency[i][j]>0 && adjacency[i][j]<min){
                min = adjacency[i][j];
                index[0] = i;
                index[1] = j;
            }
        }
    }
    return index;
}

void Prims(vector<vector<int>> adjacency){
    int len = adjacency.size();
    vector<int> index(2,INT_MAX);
    vector<bool> selected(len,false);
    vector<vector<int>> selected_edges(len,vector<int>(len,0));
    int x = 0, y = 0, count = 0, iter = 1;
    while(true){
        if(iter == 1)
            index = initial(adjacency,len);
        else
            index = indices(adjacency,selected,len);
        x = index[0];
        y = index[1];
        selected[x] = true;
        selected[y] = true;
        selected_edges[x][y] = adjacency[x][y];
        selected_edges[y][x] = adjacency[y][x];
        ++iter;
        for(int i=0;i<len;++i)
            if(selected[i]==true)
                ++count;
        if(count==len)
            break;
        else
            count = 0;
    }

    cout<<endl<<"MST using Prims Algorithm: "<<endl;
    for(int i=0;i<len;i++){
        for(int j=0;j<selected_edges.size();j++)
            cout<<selected_edges[i][j]<<" ";
    }
    int val = 0;
    for(int i=0;i<selected_edges.size();i++)
        for(int j=0;j<selected_edges.size();j++)
            val = val + selected_edges[i][j];
```

```
    cout<<"Total Cost of MST: "<<val/2<<endl<<endl;
}

int main(){
    vector<vector<int>> adjacency = {{0,2,4,4},{2,0,3,2},{4,3,0,1},{4,2,1,0}};
    cout<<endl<<"Initial Adjacency Matrix: "<<endl;
    for(int i=0;i<adjacency.size();i++){
        for(int j=0;j<adjacency.size();j++)
            cout<<adjacency[i][j]<<" ";
        cout<<endl;
    }
    cout<<endl;
    Prims(adjacency);
    return 0;
}
```

Output:

```
PS E:\Codes\MC405 Graph Theory> cd "e:\Codes\MC405 Graph Theory\" ; if ($?) { g++ Exp_6.cpp -o Exp_6 } ; if ($?) { .\Exp_6 }

Initial Adjacency Matrix:
0 2 4 4
2 0 3 2
4 3 0 1
4 2 1 0


MST using Prims Algorithm:
0 2 0 0
2 0 0 2
0 0 0 1
0 2 1 0

Total Cost of MST: 5

PS E:\Codes\MC405 Graph Theory>
```

Experiment 7

Aim:

Program to find minimal spanning tree of a graph using Kruskal's algorithm.

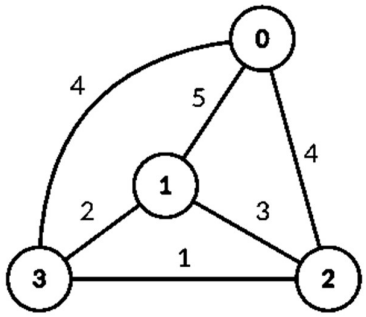
Theory:

Kruskal's algorithm is a greedy method used for finding the MST of a weighted, undirected graph. The core idea behind Kruskal's algorithm is to sort all the edges in non-decreasing order of their weight. Then, pick the smallest edge, and ensure that it doesn't form a cycle with the MST formed so far. Repeat the process until there are V−1 edges in the MST, where V is the number of vertices.

Properties: It follows greedy approach.

Limitations: It will not work if negative weights cycles exist in graph or, if graph is disconnected.

Graph Considered:



Code:

```
// Aneesh Panchal 2K20/MC/21

#include<iostream>
#include<vector>
#include<limits.h>
using namespace std;

vector<int> indices(vector<vector<int>> adjacency, vector<bool> selected, int len){
    vector<int> index(2,len);
    int min = INT_MAX;
    for(int i=0;i<len;i++){
        for(int j=0;j<len;j++){
            if(adjacency[i][j]>0 && adjacency[i][j]<min && (selected[i]==false ||
selected[j]==false)){
                min = adjacency[i][j];
                index[0] = i;
                index[1] = j;
            }
        }
    }
    return index;
}
```

```
void Kruskal(vector<vector<int>> adjacency){
    int len = adjacency.size();
    vector<int> index(2,INT_MAX);
    vector<bool> selected(len,false);
    vector<vector<int>> selected_edges(len,vector<int>(len,0));
    int x = 0, y = 0, count = 0;
    while(true){
        index = indices(adjacency,selected,len);
        x = index[0];
        y = index[1];
        selected[x] = true;
        selected[y] = true;
        selected_edges[x][y] = adjacency[x][y];
        selected_edges[y][x] = adjacency[y][x];

        for(int i=0;i<len;++i)
            if(selected[i]==true)
                ++count;
        if(count==len)
            break;
        else
            count = 0;
    }

    cout<<endl<<"MST using Kruskal Algorithm: "<<endl;
    for(int i=0;i<len;i++){
        for(int j=0;j<selected_edges.size();j++)
            cout<<selected_edges[i][j]<<" ";
        cout<<endl;
    }
    cout<<endl;

    int val = 0;
    for(int i=0;i<selected_edges.size();i++)
        for(int j=0;j<selected_edges.size();j++)
            val = val + selected_edges[i][j];
    cout<<"Total Cost of MST: "<<val/2<<endl<<endl;
}

int main(){
    vector<vector<int>> adjacency = {{0,5,4,4},{5,0,3,2},{4,3,0,1},{4,2,1,0}};
    cout<<endl<<"Initial Adjacency Matrix: "<<endl;
    for(int i=0;i<adjacency.size();i++){
        for(int j=0;j<adjacency.size();j++)
            cout<<adjacency[i][j]<<" ";
        cout<<endl;
    }
    cout<<endl;
    Kruskal(adjacency);
    return 0;
}
```

Output:

```
PS E:\Codes\MC405 Graph Theory> cd "e:\Codes\MC405 Graph Theory\" ; if ($?) { g++ Exp_7.cpp -o Exp_7 } ; if ($?) { .\Exp_7 }

Initial Adjacency Matrix:
0 5 4 4
5 0 3 2
4 3 0 1
4 2 1 0


MST using Kruskal Algorithm:
0 0 4 0
0 0 0 2
4 0 0 1
0 2 1 0

Total Cost of MST: 7

PS E:\Codes\MC405 Graph Theory>
```

Experiment 8

Aim:

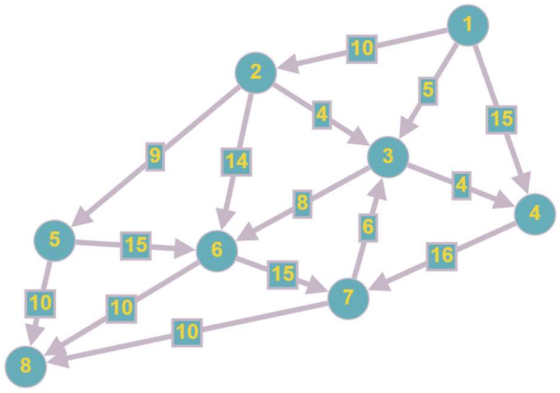
Program to maximum flow from source node to sink node using Ford-Fulkerson Algorithm.

Theory:

Ford-Fulkerson Algorithm is used to find the maximum flow of the s-t network. A network which can be solved by this algorithm is a weighted directed graph. Ford-Fulkerson Algorithm is an iterative algorithm which updates until no further update is possible. An s-t flow is assignment of values to edges of directed graph such that the following 2 conditions hold,

1. ***Capacity Constraint:***
   $0 \leq$ edge flow
2. ***Local Equilibrium:***
   Inflow = Outflow (at every vertex)

Graph Considered:



Code:

```
// Aneesh Panchal 2K20/MC/21

#include<iostream>
#include<vector>
#include<algorithm>
#include<limits.h>
using namespace std;

vector<int> find_Path(vector<vector<int>> adjacency, int source, int sink, vector<bool>
visited){
    if(source==sink){
        vector<int> path;
        path.push_back(source);
        return path;
    }

    for(int i=0;i<adjacency[source].size();i++){
```

```
        if(adjacency[source][i] && !visited[i]){
            visited[i] = true;
            vector<int> temp = find_Path(adjacency, i, sink, visited);
            if(temp.size()){
                temp.push_back(source);
                return temp;
            }
            visited[i] = false;
        }
    }
    return vector<int>();
}

void FordFulkerson(vector<vector<int>> adjacency){
    int len = adjacency.size();
    vector<vector<int>> changed = adjacency;
    vector<vector<int>> network(len,vector<int>(len,0));
    while(true){
        vector<bool> visited(len,false);
        vector<int> path = find_Path(changed,0,7,visited);
        if(path.size()==0)
            break;
        int min = INT_MAX;
        reverse(path.begin(), path.end());

        for(int i=0;i<path.size()-1;++i){
            int val = changed[path[i]][path[i+1]];
            if(val<min)
                min = val;
        }

        for(int i=0;i<path.size()-1;++i){
            changed[path[i]][path[i+1]] = changed[path[i]][path[i+1]] - min;
            changed[path[i+1]][path[i]] = changed[path[i+1]][path[i]] + min;
            network[path[i]][path[i+1]] = network[path[i]][path[i+1]] + min;
        }
    }

    cout<<"Flow Network"<<endl;
    for(int i=0;i<network.size();i++){
        for(int j=0;j<network.size();j++)
            cout<<network[i][j]<<" ";
        cout<<endl;
    }

    int max_flow=0;
    for(int i=0;i<len;++i){
        max_flow = max_flow + network[i][len-1];
    }
    cout<<"Maximum Flow: "<<max_flow<<endl<<endl;
}
```

```
int main(){
    vector<vector<int>> adjacency = {{0,10,5,15,0,0,0,0},{0,0,4,0,9,14,0,0},
                                     {0,0,0,4,0,8,0,0},{0,0,0,0,0,0,16,0},
                                     {0,0,0,0,0,15,0,10},{0,0,0,0,0,0,15,10},
                                     {0,0,6,0,0,0,0,10},{0,0,0,0,0,0,0,0}};
    cout<<endl<<"Initial Adjacency Matrix: "<<endl;
    for(int i=0;i<adjacency.size();i++){
        for(int j=0;j<adjacency.size();j++)
            cout<<adjacency[i][j]<<" ";
        cout<<endl;
    }
    cout<<endl;
    FordFulkerson(adjacency);
    return 0;
}
```

Output:

```
PS E:\Codes\MC405 Graph Theory> cd "e:\Codes\MC405 Graph Theory\" ; if ($?) { g++ Exp_8.cpp -o Exp_8 } ; if ($?) { .\Exp_8 }

Initial Adjacency Matrix:
0 10 5 15 0 0 0 0
0 0 4 0 9 14 0 0
0 0 0 4 0 8 0 0
0 0 0 0 0 0 16 0
0 0 0 0 0 15 0 10
0 0 0 0 0 0 15 10
0 0 6 0 0 0 0 10
0 0 0 0 0 0 0 0

Flow Network
0 24 5 13 0 0 0 0
14 0 4 0 9 1 0 0
0 4 0 4 0 8 0 0
0 0 4 0 0 0 13 0
0 0 0 0 0 2 0 8
0 0 0 0 1 0 6 10
0 0 3 0 0 6 0 10
0 0 0 0 0 0 0 0

Maximum Flow: 28

PS E:\Codes\MC405 Graph Theory>
```

Experiment 9

Aim:

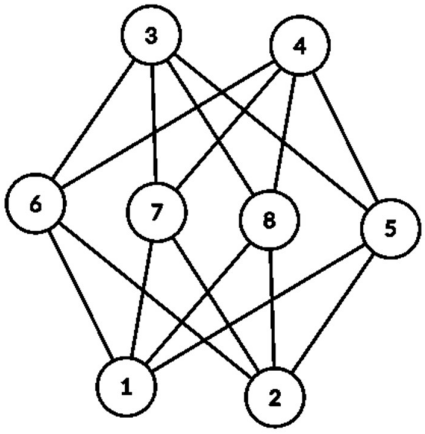
Program to find maximum matching for bipartite graph.

Theory:

Matching in a graph is a subset of edges in which no 2 edges are adjacent that is, it is a set of independent edges. Matching in which no more edges can be added is known as ***Maximal Matching***. Largest maximal matching is known as ***Maximum Matching***.

Bipartite graph is a graph in which set of graph vertices decomposed into two disjoint sets such that no two graph vertices within the same set are adjacent.

Graph Considered:



Code:

```
// Aneesh Panchal 2K20/MC/21

#include<iostream>
#include<vector>
using namespace std;

void max_match(vector<vector<int>> &adjacency, int m){
    int len = adjacency.size();
    vector<bool> visited(len);
    vector<int> match(len, -1);
    int count = 0;

    for(int i=0; i<m; i++){
        visited.assign(len, false);
        bool matched = false;
        for (int adjacentNode=0; adjacentNode<len && !matched; adjacentNode++){
            if(adjacency[i][adjacentNode] && !visited[adjacentNode]){
                visited[adjacentNode] = true;
```

```
                if(match[adjacentNode]==-1){
                    match[adjacentNode]=i;
                    matched = true;
                }
                else{
                    for(int otherNode=0; otherNode<len; otherNode++){
                        if(adjacency[match[adjacentNode]][otherNode] &&
!visited[otherNode]){
                            visited[otherNode] = true;
                            if(match[otherNode]==-1 || match[otherNode]==i){
                                match[otherNode] = match[adjacentNode];
                                match[adjacentNode] = i;
                                matched = true;
                                break;
                            }
                        }
                    }
                }
            }
        }
        if(matched)
            count++;
    }

    vector<bool> checked(len,false);
    cout<<endl;
    cout<<"Maximum matching is "<<count<<endl<<endl;
    cout<<"Edges are: "<<endl;
    for(int i=0;i<len;i++)
        if(match[i] != -1 && !checked[i]){
            cout<<match[i]+1<<" "<<i+1<<endl;
            checked[i] = true;
            checked[match[i]] = true;
    }
}

int main(){
    vector<vector<int>> adjacency = {{0,0,0,0,1,1,1,1},{0,0,0,0,1,1,1,1},
                                     {0,0,0,0,1,1,1,1},{0,0,0,0,1,1,1,1},
                                     {1,1,1,1,0,0,0,0},{1,1,1,1,0,0,0,0},
                                     {1,1,1,1,0,0,0,0},{1,1,1,1,0,0,0,0}};
    cout<<endl;
    cout<<"Adjacency matrix is: "<<endl;
    for(int i=0; i<adjacency.size(); i++){
        for(int j=0; j<adjacency.size(); j++)
            cout<<adjacency[i][j]<<" ";
        cout<<endl;
    }
    max_match(adjacency, 4);
    return 0;
}
```

Output:

```
PS E:\Codes\MC405 Graph Theory> cd "e:\Codes\MC405 Graph Theory\" ; if ($?) { g++ Exp_9.cpp -o Exp_9 } ; if ($?) { .\Exp_9 }

Adjacency matrix is:
0 0 0 0 1 1 1 1
0 0 0 0 1 1 1 1
0 0 0 0 1 1 1 1
0 0 0 0 1 1 1 1
1 1 1 1 0 0 0 0
1 1 1 1 0 0 0 0
1 1 1 1 0 0 0 0
1 1 1 1 0 0 0 0

Maximum matching is 4

Edges are:
4 5
1 6
2 7
3 8

PS E:\Codes\MC405 Graph Theory>
```

Experiment 10

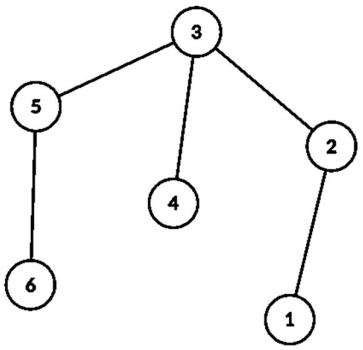
Aim:

Program to find maximum matching for general graph.

Theory:

Matching in a graph is a subset of edges in which no 2 edges are adjacent that is, it is a set of independent edges. Matching in which no more edges can be added is known as ***Maximal Matching***. Largest maximal matching is known as ***Maximum Matching***.

Graph Considered:



Code:

```
// Aneesh Panchal 2K20/MC/21

#include<iostream>
#include<vector>
using namespace std;

void max_match(vector<vector<int>> &adjacency){
    int len = adjacency.size();
    vector<bool> visited(len);
    vector<int> match(len, -1);
    int count = 0;
    auto canMatch = [&](int u, auto &self) -> bool{
        for(int v=0; v<len; v++){
            if(adjacency[u][v] && !visited[v]){
                visited[v] = true;
                if(match[v]==-1 || self(match[v],self)){
                    match[v] = u;
                    return true;
                }
            }
        }
        return false;
    };
```

```
    for(int i=0; i<len; i++){
        visited.assign(len, false);
        if(canMatch(i, canMatch)){
            count++;
        }
    }

    vector<bool> checked(len,false);
    cout<<endl;
    cout<<"Maximum matching is "<<count/2<<endl;
    cout<<endl;
    cout<<"Edges are: "<<endl;
    for(int i=0;i<len;i++)
        if(match[i] != -1 && !checked[i]){
            cout<<match[i]+1<<" "<<i+1<<endl;
            checked[i] = true;
            checked[match[i]] = true;
    }
}

int main(){
    vector<vector<int>> adjacency = {{0,1,0,0,0,0},{1,0,1,0,0,0},
                                     {0,1,0,1,1,0},{0,0,1,0,0,0},
                                     {0,0,1,0,0,1},{0,0,0,0,1,0}};
    cout<<endl;
    cout<<"Adjacency matrix is: "<<endl;
    for(int i=0; i<adjacency.size(); i++){
        for(int j=0; j<adjacency.size(); j++)
            cout<<adjacency[i][j]<<" ";
        cout<<endl;
    }
    max_match(adjacency);
    return 0;
}
```

Output:

```
PS E:\Codes\MC405 Graph Theory> cd "e:\Codes\MC405 Graph Theory\" ; if ($?) { g++ Exp_10.cpp -o Exp_10 } ; if ($?) { .\Exp_10 }

Adjacency matrix is:
0 1 0 0 0 0
1 0 1 0 0 0
0 1 0 1 1 0
0 0 1 0 0 0
0 0 1 0 0 1
0 0 0 0 1 0

Maximum matching is 3

Edges are:
2 1
4 3
6 5

PS E:\Codes\MC405 Graph Theory>
```