Experiment **#13**		Student ID	
Date		Student Name	[@KLWKS_BOT] THANOS

Experiment Title: To implement programs on problem solving using Network Flow Algorithms.

Aim/Objective: To understand the concept and implementation of programs on Network Flow Algorithms

Description: The students will understand the programs on Ford-Fulkerson Method, Edmonds-Karp Algorithm, Max-Flow Min-Cut Theorem, applying them to solve real-world problems.

Pre-Requisites:

Knowledge: Ford-Fulkerson Method, Edmonds-Karp Algorithm, Max-Flow Min-Cut Theorem

Tools: Code Blocks/Eclipse IDE.

Pre-Lab:

You are given a directed graph representing a flow network, where each edge has a capacity. Your task is to compute the maximum flow from a source node s to a sink node t using the Ford-Fulkerson method.

Input Format:

- An integer n, the number of nodes in the graph.
- An integer m, the number of edges in the graph.
- The next m lines each contain three integers u, v, and c, representing a directed edge from node u to node v with capacity c.
- Two integers s and t, the source and sink nodes.

Output Format:

- A single integer representing the maximum flow from s to t.

Sample Input:

- There are 5 cities: A, B, C, D, and E. The possible railway connections and their costs are:

Constraints:

- $2 \leq n \leq 100$
- $1 \leq m \leq 10^4$
- $1 \leq u,v \leq n$
- $0 \leq c \leq 10^9$
- There is at least one path from s to t.

Sample Input:

4 5

1 2 100

1 3 100

2 3 1

2 4 100

3 4 100

1 4

Sample Output:

200

- **Procedure/Program:**

```
import java.util.LinkedList;
import java.util.Queue;

public class Main {
    static final int MAX_NODES = 100;
    static int[][] capacity = new int[MAX_NODES][MAX_NODES];
    static int[][] flow = new int[MAX_NODES][MAX_NODES];
    static int[] parent = new int[MAX_NODES];
    static int n, m;

    static boolean bfs(int s, int t) {
        for (int i = 0; i < MAX_NODES; i++) {
            parent[i] = -1;
        }
```

```
    parent[s] = -2;
    Queue<Integer> queue = new LinkedList<>();
    queue.add(s);

    while (!queue.isEmpty()) {
        int current = queue.poll();
        for (int next = 1; next <= n; next++) {
            if (parent[next] == -1 && capacity[current][next] > flow[current][next]) {
                parent[next] = current;
                if (next == t) return true;
                queue.add(next);
            }
        }
    }
    return false;
}

static int fordFulkerson(int s, int t) {
    int maxFlow = 0;

    while (bfs(s, t)) {
        int pathFlow = Integer.MAX_VALUE;
        for (int v = t; v != s; v = parent[v]) {
            int u = parent[v];
            pathFlow = Math.min(pathFlow, capacity[u][v] - flow[u][v]);
        }

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

```
            int u = parent[v];
            flow[u][v] += pathFlow;
            flow[v][u] -= pathFlow;
        }
        maxFlow += pathFlow;
    }
    return maxFlow;
}

public static void main(String[] args) {
    n = 4; m = 5;
    capacity[1][2] = 100;
    capacity[1][3] = 100;
    capacity[2][3] = 1;
    capacity[2][4] = 100;
    capacity[3][4] = 100;

    int s = 1, t = 4;
    System.out.println(fordFulkerson(s, t));
}
}
```

- **Data and Results:**

Data

Graph with 4 nodes, 5 edges, and given capacities.

Result

Maximum flow from source to sink is calculated as 200.

- **Analysis and Inferences:**

Analysis

Ford-Fulkerson algorithm finds augmenting paths using BFS traversal.

Inferences

Graph flow increases with available paths and higher capacity edges.

In-Lab:

1. Consider a network consisting of n computers and m connections. Each connection specifies how fast a computer can send data to another computer. Kotivalo wants to download some data from a server. What is the maximum speed he can do this, using the connections in the network?

Input Format:

- The first input line has two integers n and m: the number of computers and connections. The computers are numbered 1,2,…,n. Computer 1 is the server and computer n is Kotivalo's computer.
- After this, there are m lines describing the connections. Each line has three integers a, b and c: computer a can send data to computer b at speed c.

Output Format:

- Print one integer: the maximum speed Kotivalo can download data.

Constraints :

- $1 \leq n \leq 500$
- $1 \leq m \leq 1000$
- $1 \leq a,b \leq n$
- $1 \leq c \leq 109$

Example:

Input:

4 5
1 2 3
2 4 2
1 3 4
3 4 5
4 1 3

Output:

6

- **Procedure/Program**:

```
import java.util.LinkedList;
import java.util.Queue;

public class Main {
   static final int MAX_N = 501;
   static int[][] maxFlow = new int[MAX_N][MAX_N];
   static int[] parent = new int[MAX_N];
   static int n = 4, m = 5;

   static boolean bfs(int source, int sink) {
      boolean[] visited = new boolean[MAX_N];
      Queue<Integer> queue = new LinkedList<>();
      queue.add(source);
      visited[source] = true;
      parent[source] = -1;

      while (!queue.isEmpty()) {
         int u = queue.poll();
         for (int v = 1; v <= n; v++) {
            if (!visited[v] && maxFlow[u][v] > 0) {
               queue.add(v);
               visited[v] = true;
               parent[v] = u;
               if (v == sink) return true;
            }
         }
      }
      return false;
   }
```

```
 static int edmondsKarp(int source, int sink) {
   int totalFlow = 0;
   while (bfs(source, sink)) {
      int pathFlow = Integer.MAX_VALUE;
      for (int v = sink; v != source; v = parent[v]) {
         int u = parent[v];
         if (maxFlow[u][v] < pathFlow) pathFlow = maxFlow[u][v];
      }
      for (int v = sink; v != source; v = parent[v]) {
         int u = parent[v];
         maxFlow[u][v] -= pathFlow;
         maxFlow[v][u] += pathFlow;
      }
      totalFlow += pathFlow;
   }
   return totalFlow;
 }

 public static void main(String[] args) {
   int[][] edges = {
      {1, 2, 3},
      {2, 4, 2},
      {1, 3, 4},
      {3, 4, 5},
      {4, 1, 3}
   };

   for (int i = 0; i < m; i++) {
      int a = edges[i][0], b = edges[i][1], c = edges[i][2];
      maxFlow[a][b] += c;
   }
   System.out.println(edmondsKarp(1, n));
 }
}
```

- **Data and Results:**

Data

Graph with 4 nodes, 5 edges, and assigned capacities provided.

Result

Maximum flow from source to sink is calculated as 6.

- **Analysis and Inferences:**

Analysis

Algorithm applies Edmonds-Karp method using BFS for augmenting paths.

Inferences

Flow network optimization helps determine maximum capacity between nodes efficiently.

2. A game consists of n rooms and m teleporters. At the beginning of each day, you start in room 1 and you have to reach room n. You can use each teleporter at most once during the game. How many days can you play if you choose your routes optimally?

Input Format:

- The first input line has two integers n and m: the number of rooms and teleporters. The rooms are numbered 1,2,…,n.
- After this, there are m lines describing the teleporters. Each line has two integers a and b: there is a teleporter from room a to room b.
- There are no two teleporters whose starting and ending room are the same.

Output Format:

First print an integer k: the maximum number of days you can play the game. Then, print k route descriptions according to the example. You can print any valid solution.

Constraints :

- $2 \leq n \leq 500$
- $1 \leq m \leq 1000$
- $1 \leq a,b \leq n$

Example:

Input:

6 7
1 2
1 3
2 6
3 4
3 5
4 6
5 6

Output:

2

3

1 2 6

4

1 3 4 6

- **Procedure/Program**:

```
import java.util.ArrayList;

import java.util.Arrays;


public class Main {

   static final int MAX_N = 500;

   static final int MAX_M = 1000;


   static int n = 6, m = 7;

   static int[][] adj = new int[MAX_N + 1][MAX_N + 1];

   static int[] path = new int[MAX_N];

   static int pathSize;

   static int[][] routes = new int[MAX_M][MAX_N];

   static int[] routeSizes = new int[MAX_M];

   static int routeCount = 0;
```

```
static int[][] input = {
   {1, 2}, {1, 3}, {2, 6}, {3, 4}, {3, 5}, {4, 6}, {5, 6}
};


static boolean findPath(int node) {
   if (node == n) {
      System.arraycopy(path, 0, routes[routeCount], 0, pathSize);
      routeSizes[routeCount++] = pathSize;
      return true;
   }


   for (int i = 1; i <= n; i++) {
      if (adj[node][i] == 1) {
         adj[node][i] = 0;
         path[pathSize++] = i;
         if (findPath(i)) return true;
         pathSize--;
         adj[node][i] = 1;
      }
   }
   return false;
}
```

```
public static void main(String[] args) {
    for (int[] row : adj) {
        Arrays.fill(row, 0);
    }

    for (int i = 0; i < m; i++) {
        adj[input[i][0]][input[i][1]] = 1;
    }

    while (true) {
        pathSize = 1;
        path[0] = 1;
        if (!findPath(1)) break;
    }

    System.out.println(routeCount);
    for (int i = 0; i < routeCount; i++) {
        System.out.println(routeSizes[i]);
        for (int j = 0; j < routeSizes[i]; j++) {
            System.out.print(routes[i][j] + (j == routeSizes[i] - 1 ? "\n" : " "));
        }
```

```
    }
  }
}
```

- **Data and Results:**

Data:

The game has rooms, teleporters, and a goal to reach.

Result:

Optimal routes maximize gameplay days by selecting unique paths.

- **Analysis and Inferences:**

Analysis:

Graph traversal finds distinct paths from room one to n.

Inferences:

More teleporters increase possible routes and gameplay longevity.

Post-Lab:

There are n boys and m girls in a school. Next week a school dance will be organized. A dance pair consists of a boy and a girl, and there are k potential pairs. Your task is to find out the maximum number of dance pairs and show how this number can be achieved.

Input Format:

- The first input line has three integers n, m and k: the number of boys, girls, and potential pairs. The boys are numbered 1,2,….,n, and the girls are numbered 1,2,…,m.
- After this, there are k lines describing the potential pairs. Each line has two integers a and b: boy a and girl b are willing to dance together.

Output Format:

- First print one integer r: the maximum number of dance pairs. After this, print r lines describing the pairs. You can print any valid solution.

Constraints:

- $1 \leq n,m \leq 500$
- $1 \leq k \leq 1000$
- $1 \leq a \leq n$
- $1 \leq b \leq m$

Sample Input:

```
3 2 4
1 1
1 2
2 1
3 1
```

Sample Output:

```
2
1 2
3 1
```

- **Procedure/Program**:

```java
import java.util.Arrays;

public class Main {
    static final int MAX_N = 500;
    static final int MAX_M = 500;
    static int[][] graph = new int[MAX_N + 1][MAX_M + 1];
    static int[] paired = new int[MAX_M + 1];
    static boolean[] visited = new boolean[MAX_M + 1];

    static boolean canMatch(int boy, int m) {
        for (int girl = 1; girl <= m; girl++) {
            if (graph[boy][girl] == 1 && !visited[girl]) {
                visited[girl] = true;
                if (paired[girl] == -1 || canMatch(paired[girl], m)) {
                    paired[girl] = boy;
                    return true;
                }
            }
        }
        return false;
    }

    public static void main(String[] args) {
        int n = 3, m = 2, k = 4;
        int[][] input = {{1, 1}, {1, 2}, {2, 1}, {3, 1}};

        for (int[] row : graph) {
            Arrays.fill(row, 0);
```

```
        }
        Arrays.fill(paired, -1);

        for (int i = 0; i < k; i++) {
            int a = input[i][0], b = input[i][1];
            graph[a][b] = 1;
        }

        int maxPairs = 0;
        for (int boy = 1; boy <= n; boy++) {
            Arrays.fill(visited, false);
            if (canMatch(boy, m)) maxPairs++;
        }

        System.out.println(maxPairs);
        for (int girl = 1; girl <= m; girl++) {
            if (paired[girl] != -1) {
                System.out.println(paired[girl] + " " + girl);
            }
        }
    }
}
```

- **Data and Results:**

Data

The dataset contains boys, girls, and their potential dance pairs.

Result

The maximum number of dance pairs and their valid pairings.

- **Analysis and Inferences :**

Analysis

A bipartite matching approach ensures optimal boy-girl dance pairings.

Inferences

Matching efficiency depends on available pairs and compatibility constraints.

- **Sample VIVA-VOCE Questions (In-Lab):**

1. How do you construct the minimum cut from the final residual graph after finding the maximum flow??

Nodes reachable from the source form one set; edges crossing to unreachable nodes form the cut.

2. What is the time complexity of the Ford-Fulkerson method, and on what factors does it depend?

$O(E \cdot \text{max flow})$, depends on edge capacities and augmenting paths.

3. Describe the significance of the bottleneck capacity in an augmenting path.

Limits maximum flow increase per iteration.

4. How does the Ford-Fulkerson method ensure that flow conservation and capacity constraints are maintained?

Maintains inflow = outflow at nodes, updates respect capacities.

5. Compare and contrast the space complexity of the Edmonds-Karp algorithm with the standard Ford-Fulkerson implementation.

Both $O(V + E)$; Edmonds-Karp uses BFS, Ford-Fulkerson may use DFS.

Evaluator Remark (if Any):	**Marks Secured ___out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.