Name	Balla Mahadev Shrikrishna
UID no.	2023300010
Experiment No.	7

AIM:	Experiment based on maximum flow (Ford-Fulkerson Algorithm)
Program 1	
PROBLEM STATEMENT :	**Maximum Flow Graph –** A Flow Graph is a directed graph where each edge has a capacity, representing the maximum amount of flow that can pass through it. It is commonly used in network flow problems, such as transportation networks, communication networks, and maximum flow algorithms. **Flow Graph** is a graph with following Key Components: 1. Nodes (Vertices): Represent different points in the network. 2. Edges (Arcs): Represent the connections between nodes, with a given capacity. 3. Source (s): The starting point/node where flow originates. 4. Sink (t): The endpoint/node where flow is collected. 5. Capacity (c(u, v)): The maximum amount of flow that an edge (u, v) can handle. 6. Flow (f(u, v)): The actual amount of flow passing through an edge, which must satisfy: ➔ $0 \leq f(u,v) \leq c(u,v)$ (Capacity Constraint) ➔ Flow Conservation: The total flow entering a node (except source and sink) must equal the total flow leaving it. **Residual Graph:** represents the remaining available capacity for each edge after some flow has been assigned. It helps in finding augmenting paths in algorithms like Ford-Fulkerson. Common Algorithms Using Flow Graphs are as follows: • Ford-Fulkerson Algorithm (Maximum Flow) • Edmonds-Karp Algorithm (BFS-based version of Ford-Fulkerson) **Input –** Source, Sink and the Graph, G(V,E) of at least 20 vertices and

	Capacity function values for edges. **Output –** Maximum Flow from source to sink.
graphgenerator.c :	(code below)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MIN_VERTICES 15
#define MAX_VERTICES 20
#define MAX_CAPACITY 50
#define TEST_CASES 5

int randomInt(int low, int high) {
	return low + rand() % (high - low + 1);
}

void generateGraph(FILE* file, int V) {
	int graph[V][V];

	for (int i = 0; i < V; i++)
	for (int j = 0; j < V; j++)
	graph[i][j] = 0;

	for (int i = 0; i < V - 1; i++) {
	int capacity = randomInt(1, MAX_CAPACITY);
	graph[i][i + 1] = capacity;
	}

	int extraEdges = randomInt(V, 2 * V);
	for (int i = 0; i < extraEdges; i++) {
	int u = randomInt(0, V - 1);
	int v = randomInt(0, V - 1);
	if (u != v && graph[u][v] == 0)
	graph[u][v] = randomInt(1, MAX_CAPACITY);
	}

	int src = 0, sink = V - 1;
	fprintf(file, "%d %d %d\n", V, src, sink);
```

	`` for (int i = 0; i < V; i++) { for (int j = 0; j < V; j++) fprintf(file, "%d ", graph[i][j]); fprintf(file, "\n"); } } int main() { srand(time(NULL)); FILE* file = fopen("input.txt", "w"); if (!file) { perror("Error opening file"); return 1; } for (int i = 0; i < TEST_CASES; i++) { int V = randomInt(MIN_VERTICES, MAX_VERTICES); generateGraph(file, V); fprintf(file, "\n"); } fclose(file); printf("Generated input.txt with %d test cases.\n", TEST_CASES); return 0; } ```
flow.c :	``` #include <stdio.h> #include <stdlib.h> #include <limits.h> #include <string.h> #include <time.h> #define MAX 21 // DFS for Ford-Fulkerson int dfs(int rGraph[MAX][MAX], int src, int sink, int parent[], int V, int visited[]) { visited[src] = 1; if (src == sink) return 1; ```

```
        for (int v = 0; v < V; v++) {
        if (!visited[v] && rGraph[src][v] > 0) {
        parent[v] = src;
        if (dfs(rGraph, v, sink, parent, V, visited))
                return 1;
        }
        }
        return 0;
}

int fordFulkerson(int graph[MAX][MAX], int src, int sink, int V) {
        int rGraph[MAX][MAX], parent[MAX], maxFlow = 0;
        for (int u = 0; u < V; u++)
        for (int v = 0; v < V; v++)
        rGraph[u][v] = graph[u][v];

        while (1) {
        int visited[MAX] = { 0 };
        memset(parent, -1, sizeof(parent));
        if (!dfs(rGraph, src, sink, parent, V, visited))
        break;
        int pathFlow = INT_MAX;

        for (int v = sink; v != src; v = parent[v])
        pathFlow = (pathFlow < rGraph[parent[v]][v]) ? pathFlow :
rGraph[parent[v]][v];

        for (int v = sink; v != src; v = parent[v]) {
        int u = parent[v];
        rGraph[u][v] -= pathFlow;
        rGraph[v][u] += pathFlow;
        }
        maxFlow += pathFlow;
        }
        return maxFlow;
}
```

| | ```
// BFS for Edmonds-Karp
int bfs(int rGraph[MAX][MAX], int src, int sink, int parent[], int V) {
	int visited[MAX] = { 0 };
	int queue[MAX], front = 0, rear = 0;

	queue[rear++] = src;
	visited[src] = 1;
	parent[src] = -1;

	while (front < rear) {
	int u = queue[front++];

	for (int v = 0; v < V; v++) {
	if (!visited[v] && rGraph[u][v] > 0) {
		parent[v] = u;
		visited[v] = 1;
		if (v == sink) return 1;
		queue[rear++] = v;
	}
	}
	}
	return 0;
}

int edmondsKarp(int graph[MAX][MAX], int src, int sink, int V) {
	int rGraph[MAX][MAX], parent[MAX], maxFlow = 0;
	for (int u = 0; u < V; u++)
	for (int v = 0; v < V; v++)
	rGraph[u][v] = graph[u][v];

	while (bfs(rGraph, src, sink, parent, V)) {
	int pathFlow = INT_MAX;
	for (int v = sink; v != src; v = parent[v])
	pathFlow = (pathFlow < rGraph[parent[v]][v]) ? pathFlow :
rGraph[parent[v]][v];

	for (int v = sink; v != src; v = parent[v]) {
	int u = parent[v];
``` |
```

```
        rGraph[u][v] -= pathFlow;
        rGraph[v][u] += pathFlow;
        }
        maxFlow += pathFlow;
        }
        return maxFlow;
}

// Computing max-flow and time taken by each algo
void processInputFile(const char* filename) {
        FILE* file = fopen(filename, "r");
        if (!file) {
        perror("Error opening file");
        exit(1);
        }

        int graph[MAX][MAX], V, src, sink, testCase = 1;
        while (fscanf(file, "%d %d %d", &V, &src, &sink) == 3) {
        for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
                fscanf(file, "%d", &graph[i][j]);

        clock_t start, end;
        double cpu_time_used;

        printf("\nTest Case %d:\n", testCase++);

        start = clock();
        int ff_result = fordFulkerson(graph, src, sink, V);
        end = clock();
        cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
        printf("Ford-Fulkerson (DFS): Max Flow = %d, Time = %.6f
seconds\n", ff_result, cpu_time_used);

        start = clock();
        int ek_result = edmondsKarp(graph, src, sink, V);
        end = clock();
        cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
```

```c
        printf("Edmonds-Karp (BFS):   Max Flow = %d, Time = %.6f seconds\n", ek_result, cpu_time_used);
    }
    fclose(file);
}

int main() {
    processInputFile("input.txt");
    return 0;
}
```

```
flow.c U    input.txt U ×
Exp 7 > input.txt
18 0 17
0 18 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 45 11 0 0 0 0 29 0 0 0 0 0 0 0 0 0
0 0 0 11 0 0 23 0 0 0 22 0 0 0 0 0 0 0
0 0 0 0 36 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 28 0 37 0 0 0 0 0 0 0 0 0 0 0 0
23 0 0 0 0 0 41 0 0 0 0 0 0 0 0 0 0 0
11 0 0 0 0 0 0 15 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 26 0 0 4 0 0 0 0 0 0 12 0 0
0 0 0 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 4 0 0 8 0 0 0 0 0 0 0
26 0 0 0 0 0 0 22 0 0 0 5 0 0 0 0 0 0
0 0 0 0 0 30 0 0 0 0 0 0 49 0 0 0 0 0
0 0 0 0 23 0 0 0 0 0 0 0 0 30 0 0 0 0
0 0 0 0 0 36 0 0 0 0 0 0 0 0 43 0 42 0
0 0 0 0 0 0 20 0 0 15 45 0 0 19 0 35 0 0
0 0 0 11 30 0 0 0 0 0 23 0 0 0 0 0 22 0
8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 18 0 0 0 0 0 0 0 0 0 0 24 0 0 0 0 0

16 0 15
0 22 0 0 0 0 0 0 24 0 0 0 49 0 0 0
0 0 50 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 13 0 10 0 0 0 0 0 0 0 0 0 0
0 0 0 0 36 0 0 5 0 0 0 0 0 0 0 0
0 0 0 0 0 12 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 34 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 29 0 0 0 0 0 0 26 5
0 0 0 0 0 0 0 0 7 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 44 0 0 0 23 0 0
0 0 0 0 0 0 0 0 0 0 42 0 0 0 0 0
0 0 0 0 0 0 25 0 0 0 0 19 0 0 0 0
10 0 0 0 0 0 0 37 0 0 0 0 18 0 2 0
0 0 0 0 0 0 0 48 0 0 0 0 0 30 0 0
0 0 0 0 0 0 0 0 0 0 0 0 39 0 31 0
0 23 0 0 0 0 0 0 0 0 0 0 0 0 0 28
0 0 0 0 0 0 0 0 0 0 0 0 0 0 38 0

```

RESULT:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS    GITLENS

mahadev@mahadev-Inspiron-15-3520:~/Desktop/Mahadev/SE/Sem4/DAA/Lab/Lab Sessions/Exp 7$ gcc graphgenerator.c
mahadev@mahadev-Inspiron-15-3520:~/Desktop/Mahadev/SE/Sem4/DAA/Lab/Lab Sessions/Exp 7$ ./a.out
Generated input.txt with 5 test cases.
mahadev@mahadev-Inspiron-15-3520:~/Desktop/Mahadev/SE/Sem4/DAA/Lab/Lab Sessions/Exp 7$ gcc flow.c
mahadev@mahadev-Inspiron-15-3520:~/Desktop/Mahadev/SE/Sem4/DAA/Lab/Lab Sessions/Exp 7$ ./a.out

Test Case 1:
Ford-Fulkerson (DFS): Max Flow = 1, Time = 0.000029 seconds
Edmonds-Karp (BFS):   Max Flow = 1, Time = 0.000013 seconds

Test Case 2:
Ford-Fulkerson (DFS): Max Flow = 33, Time = 0.000031 seconds
Edmonds-Karp (BFS):   Max Flow = 33, Time = 0.000014 seconds

Test Case 3:
Ford-Fulkerson (DFS): Max Flow = 6, Time = 0.000006 seconds
Edmonds-Karp (BFS):   Max Flow = 6, Time = 0.000008 seconds

Test Case 4:
Ford-Fulkerson (DFS): Max Flow = 60, Time = 0.000026 seconds
Edmonds-Karp (BFS):   Max Flow = 60, Time = 0.000017 seconds

Test Case 5:
Ford-Fulkerson (DFS): Max Flow = 35, Time = 0.000038 seconds
Edmonds-Karp (BFS):   Max Flow = 35, Time = 0.000022 seconds
mahadev@mahadev-Inspiron-15-3520:~/Desktop/Mahadev/SE/Sem4/DAA/Lab/Lab Sessions/Exp 7$
```

CONCLUSION:

Algorithm implemented -

Ford-Fulkerson(G, s, t, V):

1. Initialize residual graph rGraph[MAX][MAX]:
   for u = 0 to V-1:
      for v = 0 to V-1:
      rGraph[u][v] = G[u][v]
2. max_flow = 0
3. parent[MAX] = {-1}
4. while DFS_Find_Path(rGraph, s, t, parent, V):
   5. path_flow = ∞
   6. for v = t; v ≠ s; v = parent[v]:
      7. u = parent[v]
      8. path_flow = min(path_flow, rGraph[u][v])
   9. for v = t; v ≠ s; v = parent[v]:
      10. u = parent[v]
      11. rGraph[u][v] -= path_flow
      12. rGraph[v][u] += path_flow
   13. max_flow += path_flow
14. return max_flow

```
DFS_Find_Path(rGraph, src, sink, parent, V):
1. visited[V] = {0}
2. visited[src] = 1
3. if src == sink: return True
4. for v = 0 to V-1:
   5. if !visited[v] and rGraph[src][v] > 0:
         6. parent[v] = src
         7. if DFS_Find_Path(rGraph, v, sink, parent, V):
         8. return True
9. return False
-----------------------------------------------------------------------------------------
```

Edmonds-Karp(G, s, t, V):

```
1. Initialize residual graph rGraph[MAX][MAX]:
   for u = 0 to V-1:
         for v = 0 to V-1:
         rGraph[u][v] = G[u][v]
2. max_flow = 0
3. parent[MAX] = {-1}
4. while BFS_Find_Path(rGraph, s, t, parent, V):
   5. path_flow = ∞
   6. for v = t; v ≠ s; v = parent[v]:
         7. u = parent[v]
         8. path_flow = min(path_flow, rGraph[u][v])
   9. for v = t; v ≠ s; v = parent[v]:
         10. u = parent[v]
         11. rGraph[u][v] -= path_flow
         12. rGraph[v][u] += path_flow
   13. max_flow += path_flow
14. return max_flow

BFS_Find_Path(rGraph, src, sink, parent, V):
1. visited[V] = {0}
2. queue = {src}
3. visited[src] = 1
4. parent[src] = -1
5. while queue not empty:
   6. u = dequeue(queue)
   7. for v = 0 to V-1:
```

8. if !visited[v] and rGraph[u][v] > 0:
9. parent[v] = u
10. if v == sink: return True
11. visited[v] = 1
12. enqueue(queue, v)
13. return False

Time Complexity Analysis -

1. Ford-Fulkerson (DFS):
   - ➔ Worst Case: $O(f^*\cdot E)$...........{ f^* = max flow value }
   - ➔ Best Case: $O(E)$

2. Edmonds-Karp (BFS):
   - ➔ Worst Case: $O(V\cdot E^2)$
   - ➔ Best Case: $O(E)$

Key Differences:

DFS version depends on max flow value (f^*)
BFS version depends only on network size (V,E)
In unit-capacity graphs: BFS version becomes $O(E\sqrt{E})$

Applications of Max-Flow Algorithms:

- ➔ Network Routing
- ➔ Scheduling Problems
- ➔ Computer Vision