



Cairo University



Faculty of Computers  
and Artificial Intelligence

# **The Final Project Task**

## **Maximum Flow and Dijkstra Algorithm**

**Third Year- Computer Science**  
**Spring Semester**

**PID23867613**

**Under the supervision of**  
**Dr. Mostafa Reda El-Tantawi**

<b>Name</b>	<b>ID</b>	<b>Email</b>
Ahmed Sayed Mansour	20170022	ahmed.mans20719@gmail.com
Abdelrahman Bahig	20170143	abdelrahmanbahig04@gmail.com
Ibrahim Ramadan Abdou	20170002	ibrahemramadan130@gmail.com
Atef Magdy Metwally	20170136	atefmagdy12@gmail.com
Hatem Sayed Ali	20170084	hatemgad98@gmail.com

## Contents

<b>Introduction.....</b>	<b>1</b>
<b>Flowcharts.....</b>	<b>2</b>
<b>Maximum flow for ford-Fulkerson's algorithm.....</b>	<b>2</b>
<b>Dijkstra's algorithm.....</b>	<b>3</b>
<b>Snap Shot.....</b>	<b>4</b>
<b>Output Discussion.....</b>	<b>5</b>
<b>Program Step-By-Step.....</b>	<b>5</b>
<b>Pseudocodes.....</b>	<b>7</b>
<b>Dijkstra's algorithm.....</b>	<b>7</b>
<b>Maximum flow for ford-Fulkerson's algorithm.....</b>	<b>7</b>
<b>Implementation.....</b>	<b>8</b>
<b>Ford-Fulkerson Java Implementation.....</b>	<b>8</b>
<b>Dijkstra Java Implementation.....</b>	<b>10</b>
<b>Conclusion.....</b>	<b>12</b>
<b>Suggestions for related future work.....</b>	<b>12</b>
<b>References.....</b>	<b>13</b>

## ❖ Introduction :

**Dijkstra's algorithm** is one algorithm for determining the shortest path from a starting node to a goal node in a weighted graph. The algorithm generates a tree of shortest paths to all other points of the graph from the beginning edge to the root.

A weighted graph can be added to the **Dijkstra's algorithm**, written in 1959 and named after its founder Dutch computer scientist Edsger Dijkstra. The line may be either guided or undirected. One stipulation to use the method is that any edge of the graph must have a non-negative weight.

**Maximum flow problems** include having a viable flow via a flow network that gets the maximum flow rate possible.

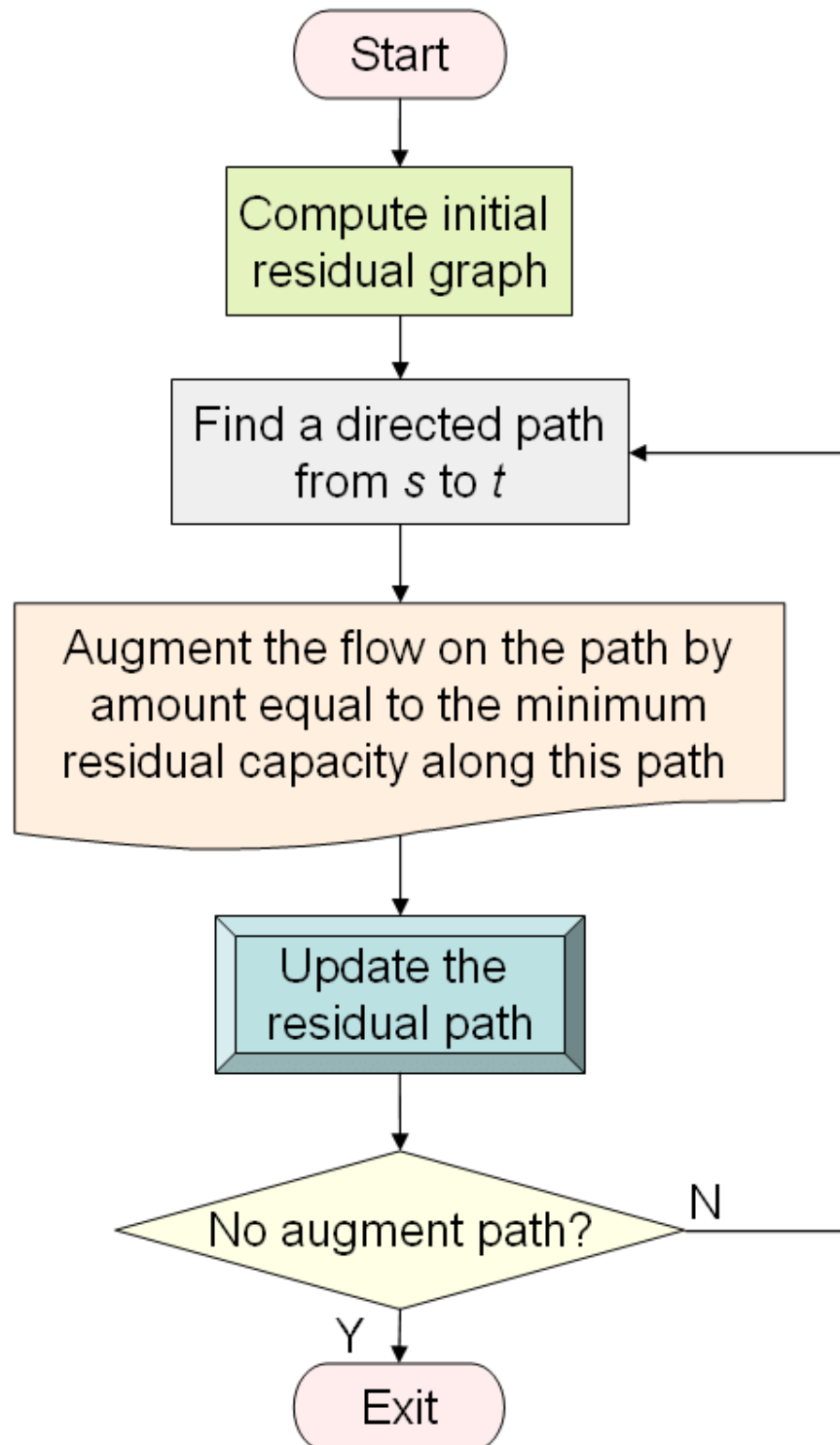
The problem of **maximal flow** can be used as a special case of more general issues of network traffic, including the problem of diffusion. The maximum s-t flow value (i.e. flow from source s to sink t) is equivalent to the minimum s-t cut capability (i.e. cutting s from t) in the network, as described in the min-cut theorem for max flow.

**The Ford-Fulkerson** process or algorithm Ford – Fulkerson (FFA) is a greedy algorithm that determines the optimal flow in a flow network. It is often named a "process" instead of a "algorithm" since it does not completely define the solution to seeking augmenting paths in a residual graph or it is defined in many implementations with different running times. **Ford & Fulkerson** In 1956, made the algorithm available. For the Edmonds – Karp algorithm, which is a specifically specified version of the Ford – Fulkerson system, the term "Ford – Fulkerson" is also used too.

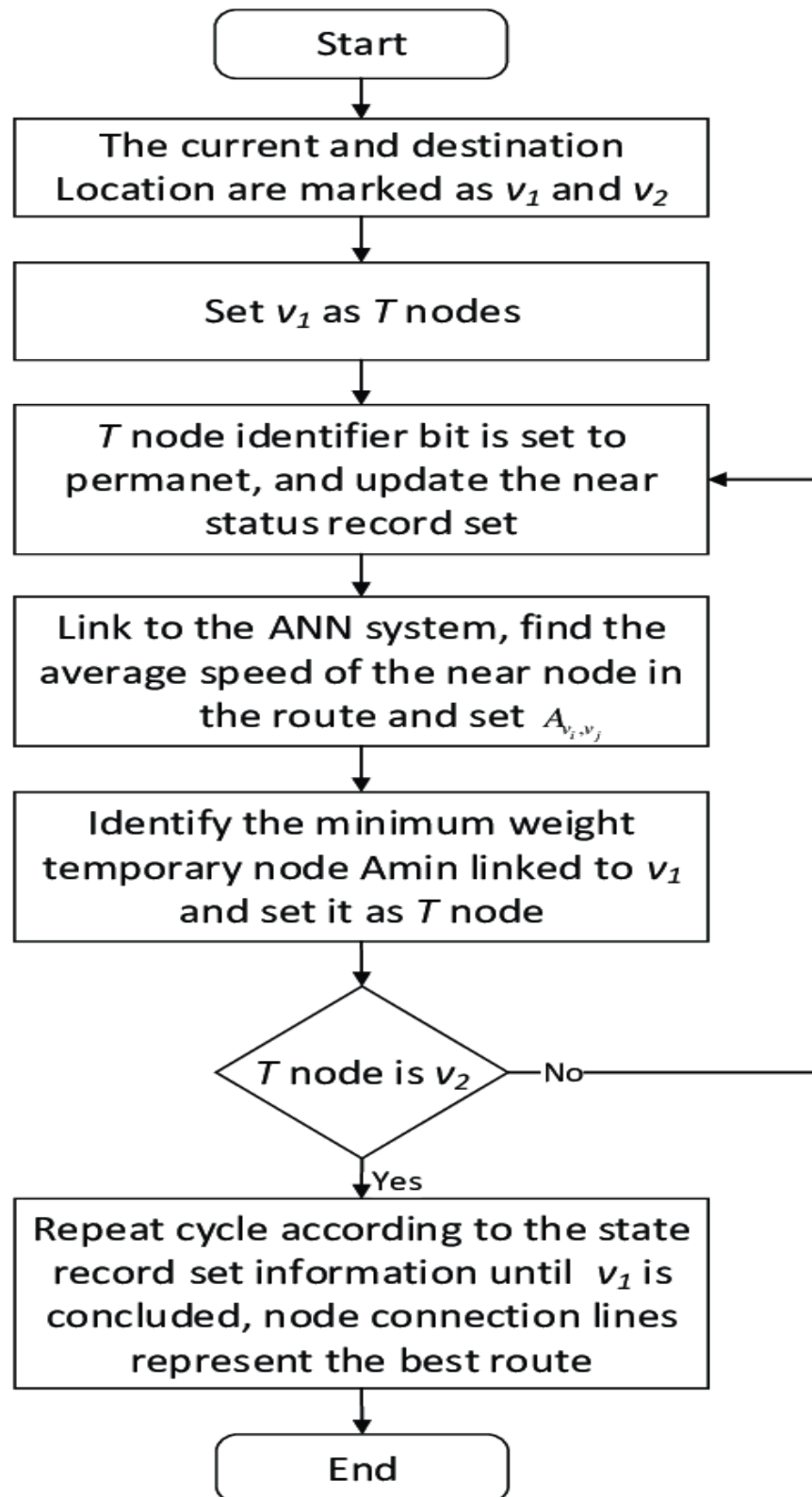
The idea behind the algorithm is as follows: as long as there is a path from the source (start node) to the sink (end node), we send flow along one of the paths with the capacity available on all edges in the path. Instead, we try that route, and so forth. An Augmenting Path is called a path with available capacity.

❖ **flow charts :**

➤ **Maximum flow for ford-Fulkerson's algorithm :**



➤ **Dijkstra's algorithm :**



## ❖ Snap Shots :

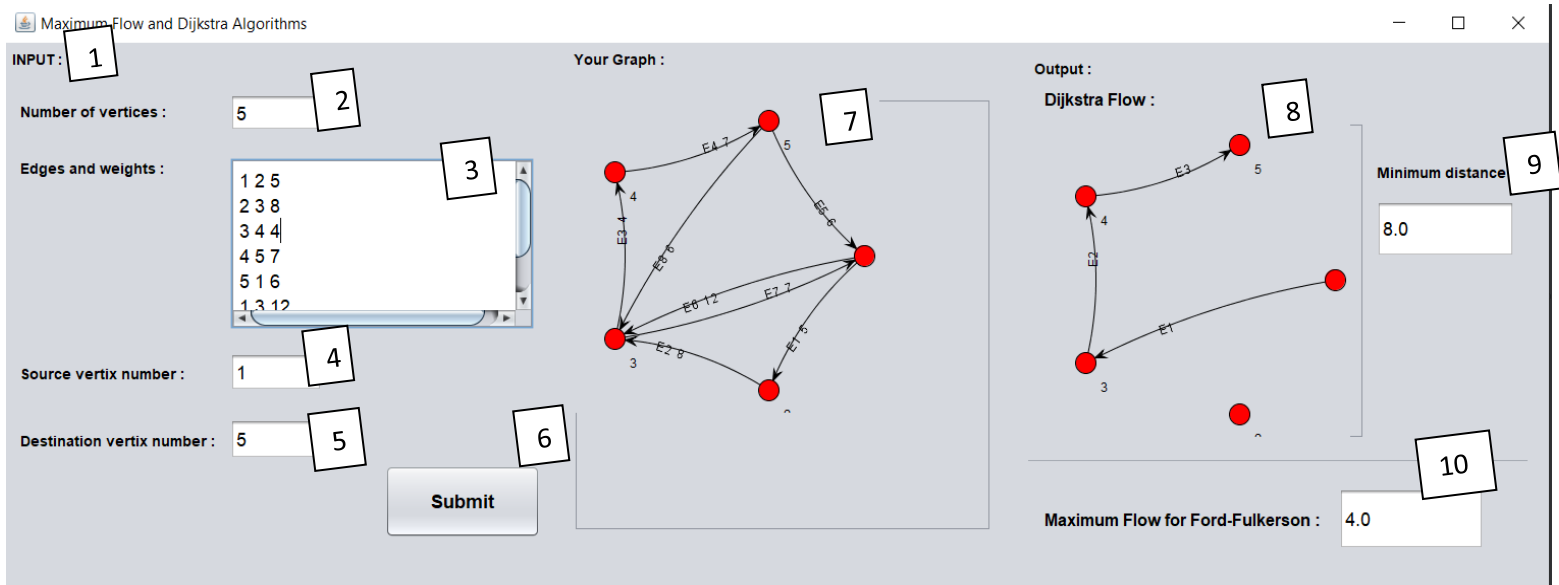


Figure 1-GUI

Number	Description
1	Input menu
2	Number of vertices
3	Edges : (Source – Destination – Weight)
4	Source Vertex
5	Destination Vertex
6	Submit button after finishing input data
7	Your input Graph
8	Dijkstra flow output
9	Minimum distance for Dijkstra algorithm
10	Maximum flow for Ford-Fulkerson Algorithm

## ❖ Output Discussion :

In this photo (Figure 1-GUI), you will see an entire Snap Shot for the program where you can see an input area on the left to input the information of the graph.

At Number seven in (Figure 1-GUI), you can see the graph that you have entered.

On the right the Output area, which contains the Dijkstra flow showing the flow from the source to destination also the minimum distance calculated.

In addition, the Maximum flow calculated using Ford-Fulkerson algorithm.

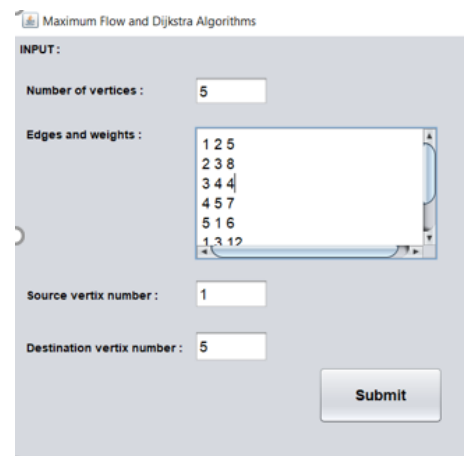
## ❖ Program Step-By-Step :

### 1. Input :

In the input, we find that the vertices are taken as number.

In addition, the edges taken formatted (source - destination - weight).

Submit button clicked after inserting all data.



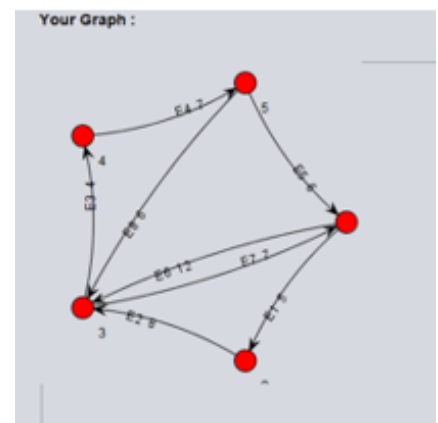
The screenshot shows a window titled "Maximum Flow and Dijkstra Algorithms". The "INPUT:" section contains the following fields:

- Number of vertices : 5
- Edges and weights : A text area containing the following text:

```
1 2 5
2 3 8
3 4 4
4 5 7
5 1 6
1 3 12
```
- Source vertex number : 1
- Destination vertex number : 5
- A "Submit" button.

### 2. Your graph :

This layout shows your graph that you entered its data.

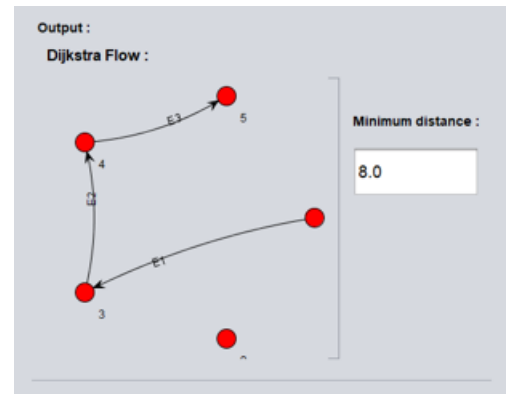


### 3. Output :

#### a. Dijkstra's Algorithm :

For Dijkstra's algorithm we shows its output flow that show the minimum distance can be calculated throw which nodes.

In addition, the minimum distance is calculated.



#### b. Maximum flow for ford-Fulkerson's algorithm :

For Maximum Flow for ford-Fulkerson's algorithm we shows the value also in a box.





## ❖ Pseudocodes :

### ➤ Dijkstra's algorithm :

```
1 function Dijkstra_Algorithm (graph, sourceNode):
2
3   create vertex array A
4
5   for each vertex v in Graph:
6     destination[v] ← INFINITY
7     previous[v] ← UNDEFINED
8     add v to A
9
10  destination [source] ← 0
11
12  while A is not empty:
13    currentNode ← vertex in A with min destination [currentNode]
14
15    remove currentNode from A
16
17    for each neighbor n of currentNode:      // only v that are still in A
18      alt ← destination [currentNode] + length (currentNode, n)
19      if alt < destination [v]:
20        destination [n] ← alt
21        previous [n] ← currentNode
22
23  return destination array, previous array
```

### ➤ Maximum flow for ford-Fulkerson's algorithm :

- **function:** FordFulkerson(Graph G,Node S,Node T):
- **Initialise** flow in all edges to 0
- **while** (there exists an augmenting path(P) between S and T in residual network graph ):
- **Augment** flow between S to T along the path P
- **Update** residual network graph
- **return**

## ❖ Implementation :

### ➤ Ford-Fulkerson Java Implementation:

```
public class Graph {
    private int vCount;
    private float[][] adj;

    public int getvCount() {
        return vCount;
    }

    public float[][] getAdj() {
        return adj;
    }

    public Graph(int vCount) {
        this.vCount = vCount;
        adj = new float[vCount][vCount];
        for (int i = 0; i < vCount; i++) {
            for (int j = 0; j < vCount; j++) {
                adj[i][j] = 0;
            }
        }
    }

    public void addEdge(int i, int j, float weight) {
        adj[i][j] = weight;
    }

    public void removeEdge(int i, int j) {
        adj[i][j] = 0;
    }

    public boolean hasEdge(int i, int j) {
        if (adj[i][j] != 0) {
            return true;
        }
        return false;
    }

    public List<Integer> neighbours(int vertex) {
        List<Integer> edges = new ArrayList<Integer>();
        for (int i = 0; i < vCount; i++)
            if (hasEdge(vertex, i))
                edges.add(i);
        return edges;
    }
}
```

```

public static boolean bfs(Graph rg, int source, int dest, int parent[]) {
    boolean[] seen = new boolean[rg.getvCount()];
    for (int i = 0; i < rg.getvCount(); i++)
        seen[i] = false;

    LinkedList<Integer> q = new LinkedList<Integer>();
    q.add(source);
    seen[source] = true;
    parent[source] = -1;

    while (!q.isEmpty()) {
        int i = q.poll();
        for (Integer j : rg.neighbours(i)) {
            if ((seen[j] == false) && (rg.getAdj()[i][j] > 0)) {
                q.add(j);
                seen[j] = true;
                parent[j] = i;
            }
        }
    }
    return seen[dest];
}

public static float FordFulkerson(Graph g, int source, int dest) {
    if (source == dest) {
        return 0;
    }
    int V = g.getvCount();
    Graph rg = new Graph(V);
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            rg.getAdj()[i][j] = g.getAdj()[i][j];
        }
    }
    int parent[] = new int[V];
    float max_flow = 0; // max flow value
    while (bfs(rg, source, dest, parent)) {
        float path_flow = Float.MAX_VALUE;
        for (int i = dest; i != source; i = parent[i]) {
            int j = parent[i];
            path_flow = Math.min(path_flow, rg.getAdj()[j][i]);
        }
        for (int i = dest; i != source; i = parent[i]) {
            int j = parent[i];
            rg.getAdj()[j][i] -= path_flow;
            rg.getAdj()[i][j] += path_flow;
        }
        max_flow += path_flow;
    }
    return max_flow;
}

```

## ➤ Dijkstra Java Implementation:

```
public class Vertex implements Comparable<Vertex> {

    public String name;
    public List<Edge> adjacenciesList;
    public boolean visited;
    public Vertex predecessor;
    public double distance = Double.MAX_VALUE;

    public Vertex(String name) {
        this.name = name;
        this.adjacenciesList = new ArrayList<>();
    }

    public void addNeighbour(Edge edge) {
        this.adjacenciesList.add(edge);
    }

    @Override
    public String toString() {
        return this.name;
    }

    @Override
    public int compareTo(Vertex otherVertex) {
        return Double.compare(this.distance, otherVertex.getDistance());
    }
}

public class Edge {

    public double weight;
    public Vertex startVertex;
    public Vertex targetVertex;

    public Edge(double weight, Vertex startVertex,
                Vertex targetVertex) {
        this.weight = weight;
        this.startVertex = startVertex;
        this.targetVertex = targetVertex;
    }
}
```

```

public class DijkstraShortestPath {

    public void computeShortestPaths(Vertex sourceVertex) {

        sourceVertex.setDistance(0);
        PriorityQueue<Vertex> priorityQueue = new PriorityQueue<>();
        priorityQueue.add(sourceVertex);
        sourceVertex.setVisited(true);

        while( !priorityQueue.isEmpty() ){
            // Getting the minimum distance vertex from priority queue
            Vertex actualVertex = priorityQueue.poll();

            for(Edge edge : actualVertex.getAdjacenciesList()){

                Vertex v = edge.getTargetVertex();
                if(!v.isVisited())
                {
                    double newDistance = actualVertex.getDistance() + edge.getWeight();

                    if( newDistance < v.getDistance() ){
                        priorityQueue.remove(v);
                        v.setDistance(newDistance);
                        v.setPredecessor(actualVertex);
                        priorityQueue.add(v);
                    }
                }
            }
            actualVertex.setVisited(true);
        }
    }

    public List<Vertex> getShortestPathTo(Vertex targetVertex) {
        List<Vertex> path = new ArrayList<>();

        for(Vertex vertex=targetVertex;vertex!=null;vertex=vertex.getPredecessor()){
            path.add(vertex);
        }
        Collections.reverse(path);
        return path;
    }
}

```

## ❖ Conclusion :

The computed time complexity for both algorithms are acceptable as we get the shortest path between two vertices and the maximum flow diagram.

Time Complexity of Dijkstra's Algorithm is  $O(V^2)$  but with min-priority queue it drops down to  $O(V + E \log V)$ .

### ➤ Pros :

- Learning better algorithms to get shortest.
- Writing Software life cycle helps to enhance our documentation skills describing our projects.

### ➤ Cons :

- Both algorithms are pure without applying in a real life problems.

## ❖ Suggestions for related future work :

I suggest for a related future work with Dijkstra Algorithm and Maximum Flow algorithms is to apply those algorithms on a real problems in real life like:

- ◆ Geographical Maps.
- ◆ Find locations of Map, which refers to vertices of graph.
- ◆ IP routing to find Open shortest Path First.
- ◆ The telephone network.
- ◆ Airline Scheduling.

Applying our algorithms will also help us to understand those algorithms efficiently.

In addition working on such a big Project with better algorithms will increase the efficiency and performance of our applications.

## ❖ References :

1. Controversial, see Moshe Sniedovich (2006). "Dijkstra's algorithm revisited: the dynamic programming connexion".
2. Frana, Phil (August 2010). "An Interview with Edsger W. Dijkstra". Communications of the ACM.
3. Hall, A. How to use Dijkstra's algorithm. Retrieved April 27, 2016, from <https://www.youtube.com/watch?v=Cjzzx3MvOcU>
4. Schrijver, A. (2002). "On the history of the transportation and maximum flow problems".
5. Felner, Ariel (2011). Position Paper: Dijkstra's Algorithm versus Uniform Cost Search or a Case against Dijkstra's Algorithm. Proc. 4th Int'l Symp.
6. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "Section 26.2: The Ford–Fulkerson method". Introduction to Algorithms (Second Ed.). MIT Press and McGraw–Hill.
7. George T. Heineman; Gary Pollice; Stanley Selkow (2008). "Chapter 8: Network Flow Algorithms". Algorithms in a Nutshell. Oreilly Media.
8. Backman, Spencer; Huynh, Tony (2018). "Transfinite Ford–Fulkerson on a finite network".