Cairo University Faculty of Computers and Information



# **CS316**

Algorithms

2nd Semester 2020 Research

**Maximum Flow and Dijkstra Algorithms**

|  |  |  |  |
| --- | --- | --- | --- |
| ID | National-Id | Name | Email |
| 20170111 | 29805100100506 | Sara Saeed Ibrahim | sara.said1998@gmail.com |
| 20170113 | 29907220100817 | Sameh Serag El-Den Ramadan | samehserag1@gmail.com |
| 20170126 | 29905182102743 | Shahesta Mohamed Ahmed | shahesta.alkady@gmail.com |
| 20170127 | 29901170105024 | Shorouq Hessuein Hessuein | shrouqhussein17@gmail.com |

**The Project ID : PID23867952**

**Introduction**

The purpose of this article is to discuss two of the algorithms much use which are Dijkstra and Maximum Flow algorithms. So, let us begin with a brief description of Dijkstra's algorithm.

Dijkstra's algorithm is an algorithm for finding the shortest paths in a graph between nodes which, for example, can represent road networks.

The original algorithm from Dijkstra found the shortest path between two given nodes, but a more common version fixes a single node as the "source" node and finds the shortest paths from the source to all other nodes.

On the other hand, we have the Maximum Flow problem.

Suppose a flow network has a maximum flow capability at each edge. The max flow problem is all about finding the maximum flow we can get when going from a vertex of one source to a vertex of destination. An edge cannot outstrip its capacity. Many algorithms exist to solve the problem of maximum flow. Ford-Fulkerson algorithm and Dinic's Algorithm are two major algorithms to solve these kinds of problems. And here we have used the Ford-Fulkerson algorithm.

## 

## Software Purpose

The purpose of this project is to enable users to create directed graphs using our Graphic User Interface (GUI) and then apply Maximum flow or Dijkstra’s algorithms.

## Software Scope

This project aims to create graph then find maximum flow between two vertices or finds shortest paths from source to all vertices in the entered graph and shows them step by step in the graph.

Requirements

1. This program allow user enter vertices name and edges (initial, terminal, weight) of the graph

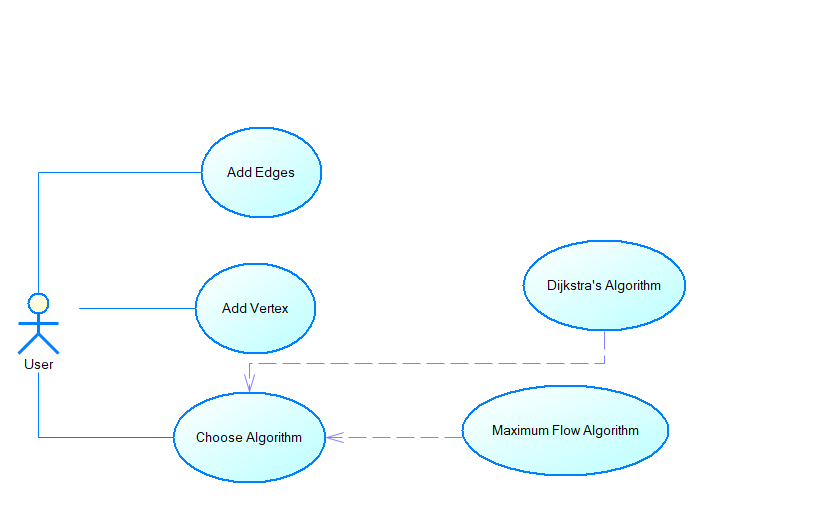
2. This program displays the graph that user has entered.

3. User has the option to choose maximum flow or Dijkstra’s algorithm to be applied.

4. The program can find maximum flow between two vertices that user entered and shows step by step in the graph

5. The program can find shortest paths from source to all vertices in the entered graph and shows step by step in the graph

Use Case Model:



## Use Case Tables:

|  |  |  |
| --- | --- | --- |
| Use Case ID: | 1 | |
| Use Case Name: | Enter vertices | |
| Actors: | User | |
| Pre-conditions: |  | |
| Post-conditions: |  | |
| Flow of events: | **User Action** | **System Action** |
| 1- User enter vertex name then press enter |  |
|  | 2- system save vertex name |
| And so on |  |
| When finish press finish |  |
|  | System send vertices name to the graph |
| Exceptions: | **User Action** | **System Action** |
| 1- User enter same name of existing vertex or empty cell |  |
|  | Exception massage and let user enter gain |
| Includes: |  | |
| Notes and Issues: |  | |

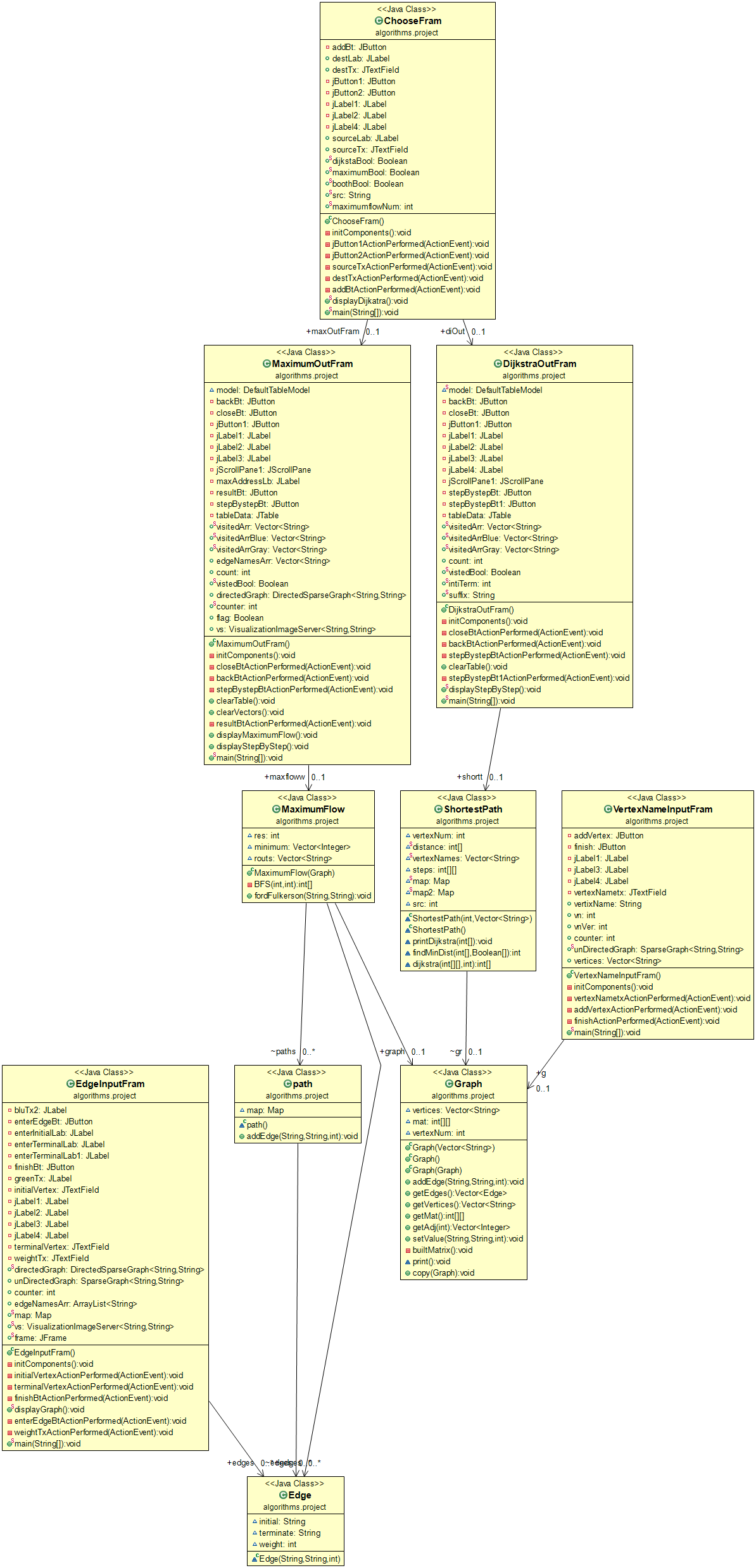
|  |  |  |
| --- | --- | --- |
| Use Case ID: | 2 | |
| Use Case Name: | Enter edges | |
| Actors: | User | |
| Pre-conditions: | Vertices are entered | |
| Post-conditions: | The graph will be created | |
| Flow of events: | **User Action** | **System Action** |
| 1- User Enter start vertex, end vertex and weight of edge |  |
|  | 2- system send edge information to graph and tell the user |
| And so on |  |
| When finish press finish |  |
| Exceptions: | **User Action** | **System Action** |
| 1- User Enter start vertex, end vertex and weight of edge or any cell being empty |  |
|  | 2- system can’t found vertex in such name, show massage to user and then let user enter correct vertex |
| Includes: |  | |
| Notes and Issues: |  | |

|  |  |  |
| --- | --- | --- |
| Use Case ID: | 3 | |
| Use Case Name: | Get maximum flow | |
| Actors: | User | |
| Pre-conditions: | Graph is entered | |
| Post-conditions: | Know maximum flow between two vertices | |
| Flow of events: | **User Action** | **System Action** |
| 1- User Enter start vertex and end vertex |  |
|  | 2- system show steps in the graph and table to calculate maximum flow |
|  |  |
| Exceptions: | **User Action** | **System Action** |
| 1. User Enter start vertex and end vertex 2. Or enter empty cell |  |
|  | 2- system can’t found vertex in such name, show massage to user and then let user enter correct vertex |
| Includes: | Graph | |
| Notes and Issues: |  | |

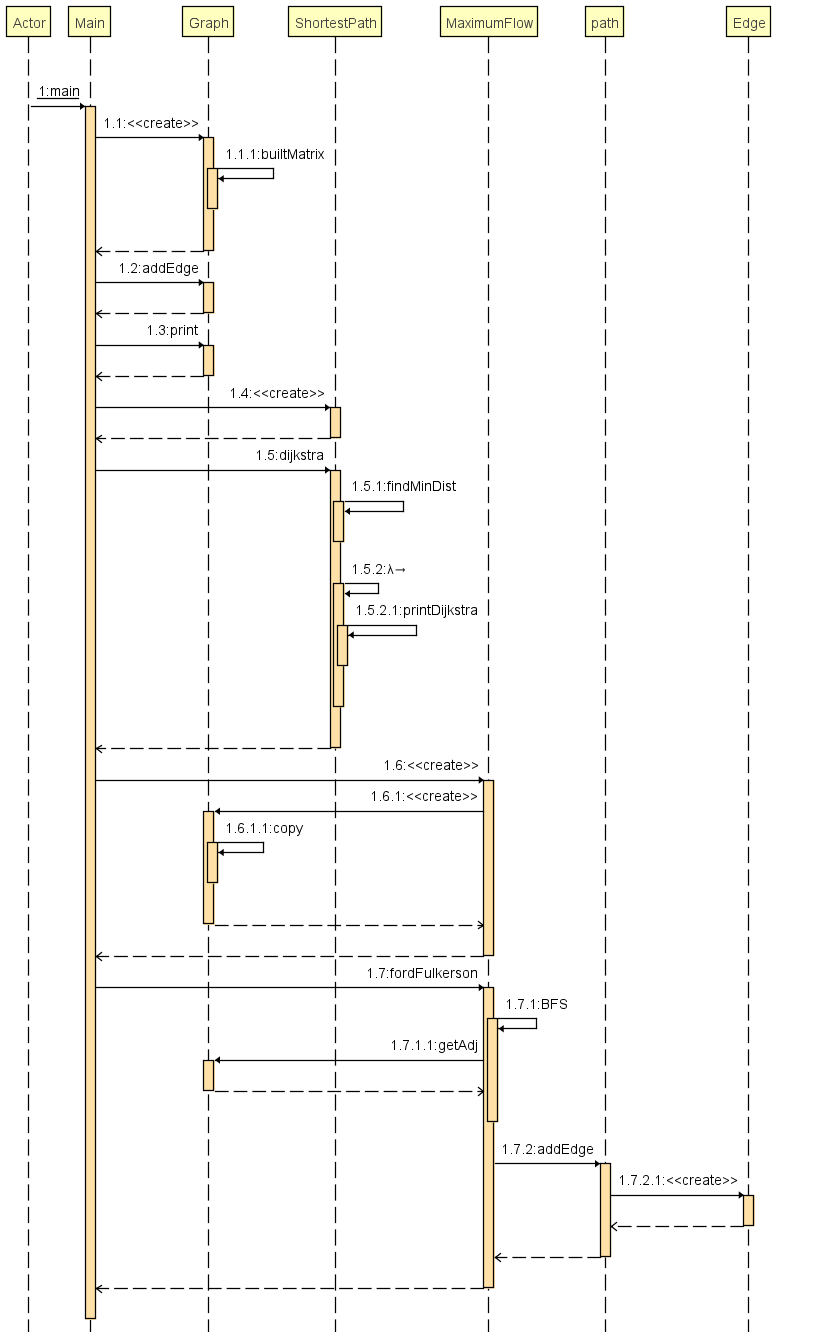
|  |  |  |
| --- | --- | --- |
| Use Case ID: | 4 | |
| Use Case Name: | Dijkstra’s Algorithms | |
| Actors: | User | |
| Pre-conditions: | Graph is entered | |
| Post-conditions: | Know shortest path from start vertex to other vertex | |
| Flow of events: | **User Action** | **System Action** |
| 1- User enter source vertex |  |
|  | 2- system show steps in the graph and table to show shortest path to other vertices |
| Exceptions: | **User Action** | **System Action** |
| 1. User Enter source vertex 2. Or empty cell |  |
|  | 2- system can’t found vertex in such name, show massage to user and then let user enter correct vertex |
| Includes: | Graph | |
| Notes and Issues: |  | |

|  |  |  |
| --- | --- | --- |
| Use Case ID: | 5 | |
| Use Case Name: | Choose | |
| Actors: | User | |
| Pre-conditions: | Graph is entered | |
| Post-conditions: | Call maximum flow or Dijkstra’s Algorithm | |
| Flow of events: | **User Action** | **System Action** |
| 1- User choose which algorithm want to apply |  |
|  | 2- system calls the algorithm and show the graph |
| Exceptions: | **User Action** | **System Action** |
|  |  |
| Includes: | Graph | |
| Notes and Issues: |  | |

Class diagram:



Sequence diagram:



# GitHub repository link (Implementation Code)

**On GitHub**

* <https://github.com/samehserageldine/Algorithmes-Project-GUI.git>

**On Google Drive**

* <https://drive.google.com/drive/folders/16IFVHr74rUr1xLb852k31jaRpZCUpvOM>

# Pseudocode

**Dijkstra's Algorithm**

**Dijkstra's Algorithm to Find Shortest-Path:**

1. Initialization of all nodes with "infinite" distance; launch node initialization with 0.
2. Marking of the start node distance as permanent, all the other distances as temporary. The setting node immediately begins as active.
3. Setting node starts as active.
4. Calculation of the temporary distances of all of the active node's neighbor nodes by summarizing its distance with both the edge weights.
5. If such a measured node distance is smaller than the current one, then change the distance and assign the current node as the predecessor. This move is also called an update and is the core idea of Dijkstra.
6. The setting of the node as active with minimum temporary distance. Label its permanent reach.
7. Repeat steps 4 to 7 until there are no nodes left with a fixed distance that are only temporary distances for neighbors.

**Dijkstra's Algorithm Pseudocode:**

Function dijkstra (Graph, src):

       distance[src] = 0 // Distance from source to source is set to 0

       for each vertex vert in Graph: // Initialize

           if vert ≠ src

               distance[vert] = infinity //source to each node set to infinity

           add vert to queue // All nodes initially in queue

      while queue is not empty: // it is main loop

          vert = vertex in queue with min distance[vert] // the first run-through, this vertex is the source node

          remove vert from queue

          for each neighbor n of vert: // where neighbor n has not yet been removed from queue.

              alt = distance[vert] + length (vert, n)

              if alt < distance[n]: // A shorter path to n is found

                  distance[n] = alt // Update distance of n

      return distance []

  end function

**Maximum flow**

**Using Ford-Fulkerson Algorithm:**

1. It gets a graph and source and destination vertices as Input
2. It creates an residual graph initialized equal to the input graph and initializes maxflow=0
3. It loops as long as BFS (or DFS) returns a path from source to vertex and

* Finds the maximum path flow
* Adds the path flow to the maxflow

1. Lastly returns the max flow

**Pseudocode function Ford-Fulkerson Algorithm:**

**Ford-Fulkerson (Graph graph, String start, String end):**

**If start = end return 0**

**Initial maximumFlow = 0**

**While there exist paths between start and end:**

* 1. **Get path using BFS or DFS**
  2. **Initialize flow a maximum number**
  3. **Loop from start to end and reduce from this path flow (For each edge from start to end in this path):**

**If flow > weight of current edge in this path:**

**flow = weight of current edge in this path**

* 1. **Loop from start to end and reduce from this path flow (For each edge from start to end in this path):**

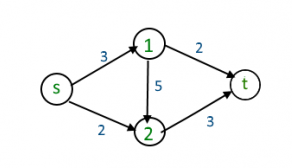
**Weight of current edge in this path =**

**Weight of current edge in this path - flow**

* 1. **Add this flow to maximumFlow (maximumFlow = maximumFlow + flow)**

**return maximumFlow**

**Test case**

* Inputs:
* Vertices name : s , 1 , 2 , t
* Edges ( Initial vertex, terminal vertex, weight ) :
* (s , 1 , 3)
* (s , 2 , 2)
* (1 , 2 , 5)
* (1 , t , 2)
* (2 , t , 3)
* Outputs:

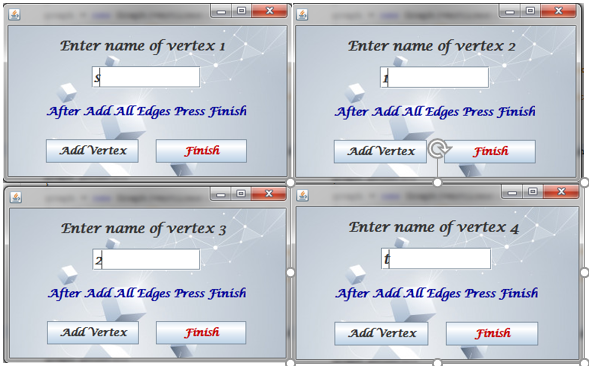
1. For maximum flow choice from s to t:

|  |  |  |
| --- | --- | --- |
|  | Path | flow |
| 1 | s, 1, t | 2 |
| 2 | s, 2, t | 2 |
| 3 | s, 1, 2 | 1 |

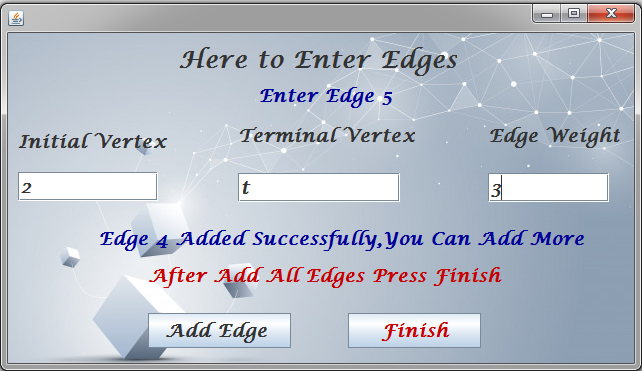
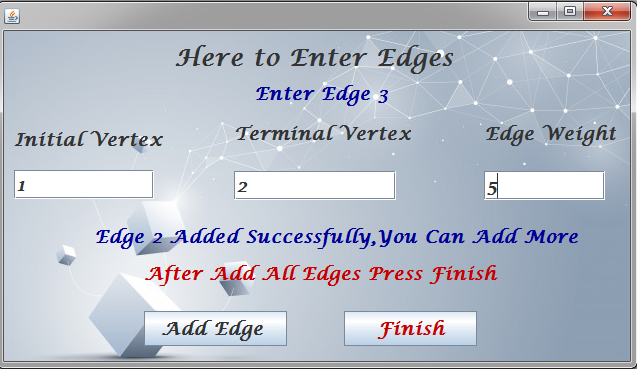
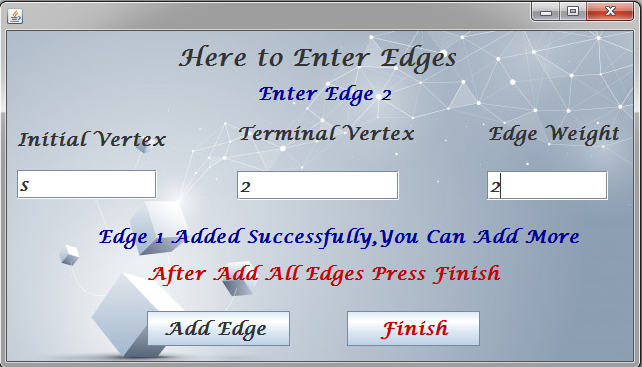
Then the maximum flow from s to t is 5 (the summation of above paths).

1. For shortest path from s:

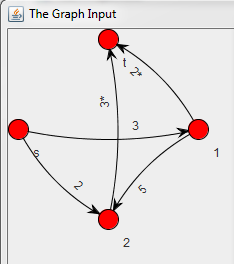
|  |  |  |
| --- | --- | --- |
| To | path | Cost |
| S |  | 0 |
| 1 | s , 1 | 3 |
| 2 | s , 2 | 2 |
| T | s , 2 , t | 5 |

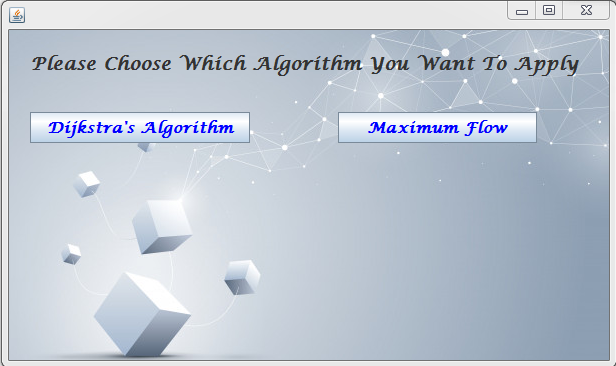
Screen shots for the program for this test case:

Note: in each time enter vertex name press Add Vertex after enter the vertex name and when finish press finish



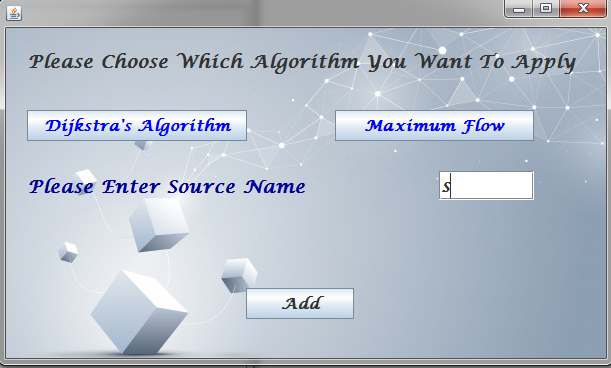
Note: in each time after you enter edge information press Add Edge and when you finish enter Finish.

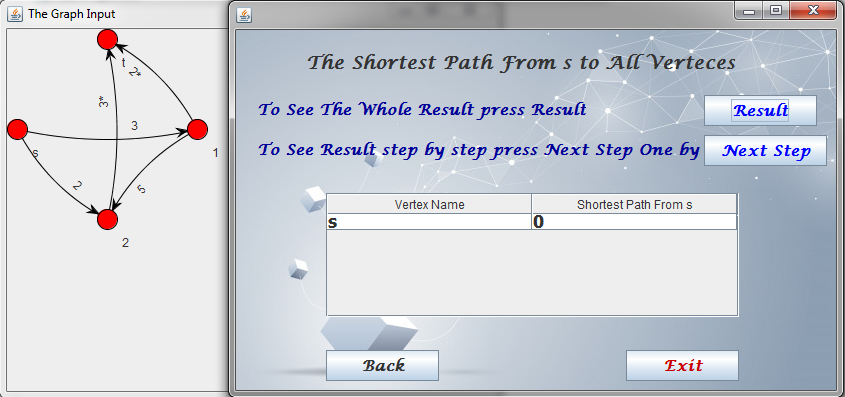
**Your graph**

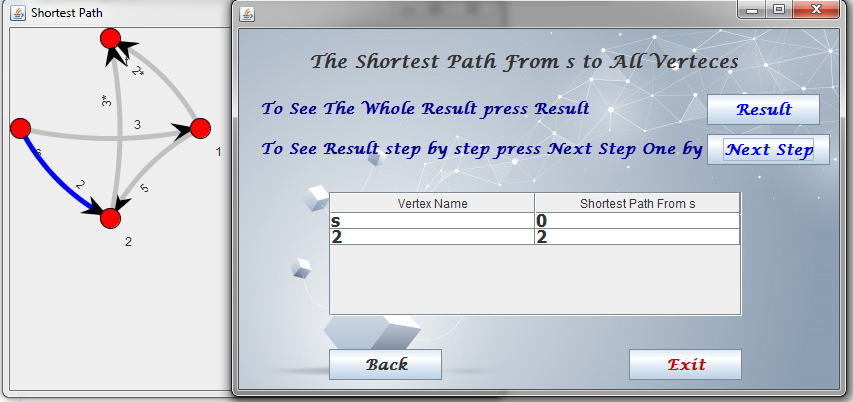


Here we will choose the algorithm to run first

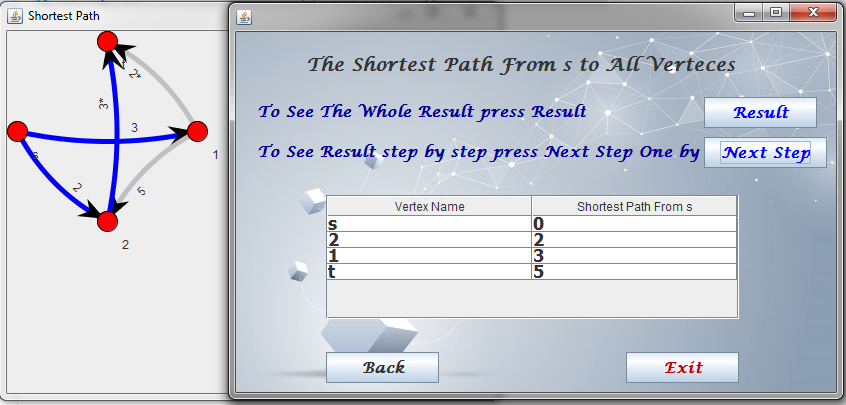
1. For Dijkstra’s algorithm:

After enter source name press Add after that will see next window





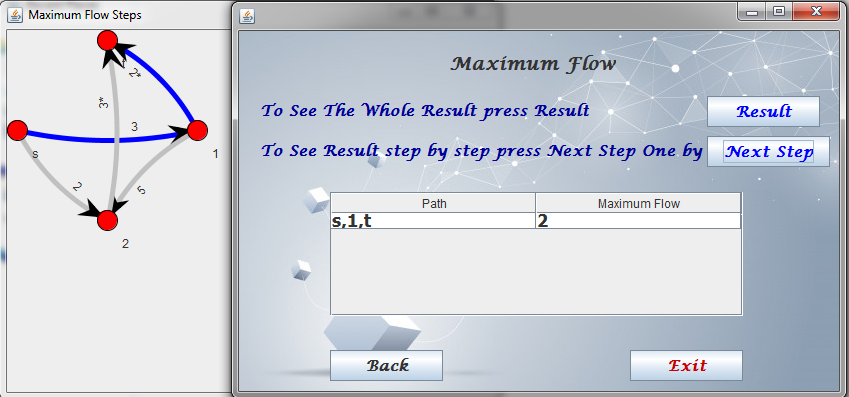
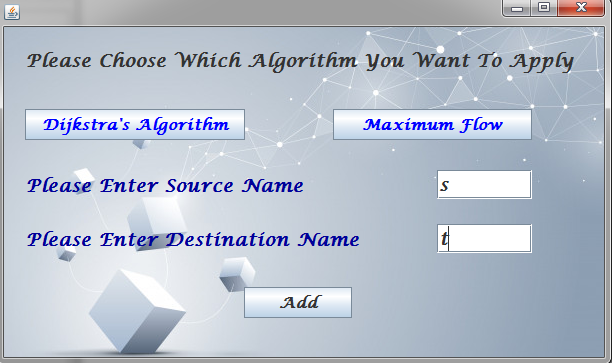


 Note: to see next step press on Next Step and you will see the step in the table and the graph

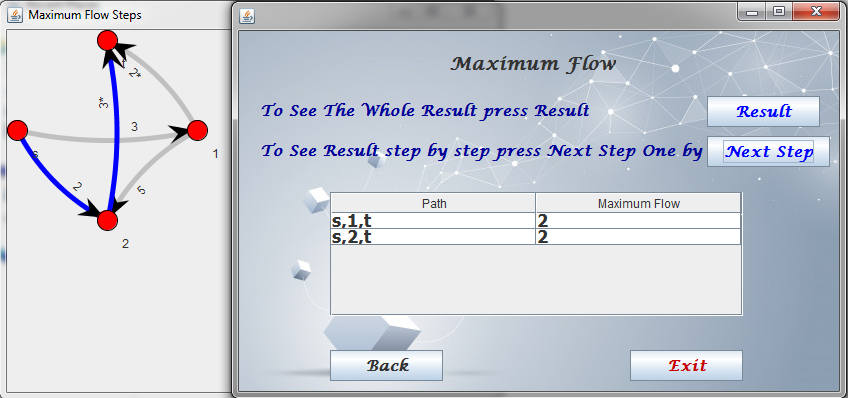
If you want to see whole result without steps press Result

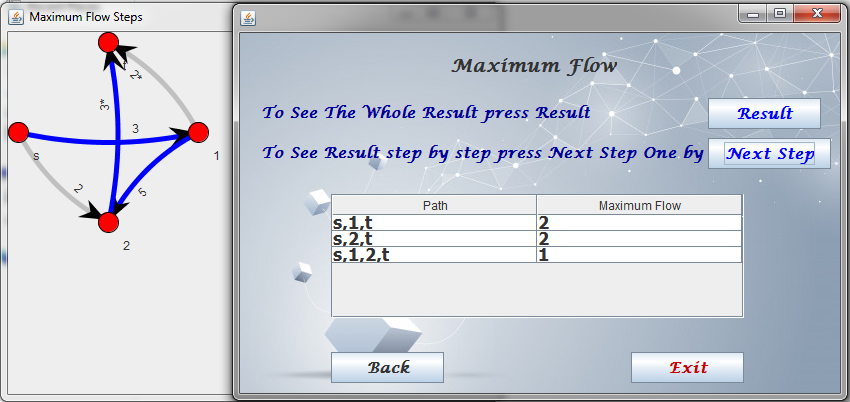
Here we will show step by step

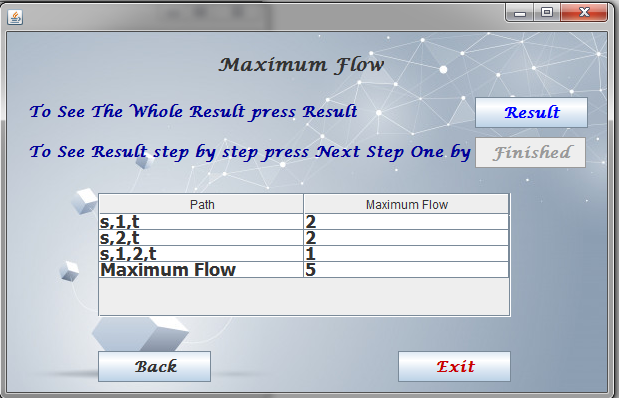
Note: if you want to see another choice press back

1. For maximum flow choice:

After enter source and destination press Add







**Suggestions for related future work**

* We can using Floyd-Warshall algorithm in shortest path instead of Dijkstra's Algorithm because it gets shortest path for many sources not only for one source.
* We can use Dinic's Algorithm because it is faster algorithm for calculating maximum flow over the networks

**Conclusion**

**We first analyzed and design the project, in which we determined the Purpose, Scope, and requirement of the project. then we built the Use Case Model, Use Case Tables, Class diagram, and Sequence diagram.**

**Second, we implemented in the java programming language the Ford-Fulkerson Algorithm to get the Maximum Flow from source to destination, also we implemented Dijkstra Algorithm to get the shortest path from a source. Then we turned to visibility graphs in our GUI in which the user has the choice to show the graph implements step by step.**

**Pros:**

1. **The program is easy to use by some instructions.**
2. **The program lets the user implements the two algorithms on the same graph more than one time, and also show it step-by-step.**
3. **The program handles any kind of mistakes are made by the user**

**Cons:**

1. **sometimes it takes 2 sec to display the first output graph**

**References**

1. J. O’Rourke, *Computation Geometry in C*, Cambridge University Press, 1994.
2. M. deBerg et al, *Computational Geometry: algorithms and applications*, Springer Verlag, 1997.