CS 542

Computer Networks 1

OPERATIONS MANUAL

Rudra, Ronit
A20379221
rrudra@hawk.iit.edu
Seat Number : 74

Ajmire, Shushupti, Vijay
A20385389
sajmire@hawk.iit.edu
Seat Number : 03

Course Instructor
Prof. Michael Choi

1 Readme

This operations manual is a guide fulfilling the project requirements for CS-542 Section 4 for the Fall 2016 Semester.

The Manual is divided into multiple sections detailing the functionality of the GUI, cases tested on the program, backend designs as well as brief description of the algorithmns used.

This Program is compatible with 64-bit architectures of both Windows OS (.exe) and Linux OS (binary). This was tested on a machines running 64-bit Linux Mint 18, 64-bit Linux Mint 17.3 and 64-bit Windows 7 Ultimate.

2 Navigating The Directory

The zip file contains all the files required for the proper functioning of the program as well as required documentation.

Operations Manual.pdf Manual detailing the program functionality as well as test cases.

CS542_Project_PPT.pptx Powerpoint Presentation for brief explanation.

Folder: Linux Executable Contains the linux executable file *Simulator*

Folder: Windows Executable Contains the windows executable file *Simulator.exe*.

Folder: Source Code Contains the entire source code of the program.

gui.py Source code for entire program, including GUI and backend functions.

pathfind.py Source code for back-end only functions for implementing routing. (not-updated)

helper.py Source code for back-end only functions for reading data, visualization, network modification etc. (not-updated)

Example<x>.txt 5 test cases 1 through 5.

NOTE: See gui.py for a well-commented and up-to-date version of code.

3 Workload Division

The workload was divided between the two members and each one's contribution is described briefly below.

- **Rudra, Ronit :** Responsible for the entire back-end design and debugging of front-end and back-end code. Also responsible for generating executable packages for both Linux and Windows operating systems.
  - Wrote Python Code for performing read operations; graph formation from data; visualization; routing algorithm (Dijkstra's Shortest Path); edge modification; routing table formation; node addition; deletion and recovery.
  - Generated standalone executable files for Linux using *Pyinstaller* and for Windows using *CX-Freeze*.
  - Debugged back-end code over multiple revisions to ensure compatibility with front-end GUI in response to testing.

  - Helped in design of GUI.
  - Front-end/back-end interfacing by modification of back-end functional code.

- **Ajmire, Shushupti :** Responsible for the front-end design, interfacing and testing outputs of final product with test cases.
  - Designed the GUI using *Tkinter* package for python which included file navigation, visualization, user inputs, outputs through embedded widgets.
  - Modified back-end code to accept callbacks from front-end.
  - Performed testing on iterative versions of program to ensure correct operatibility.
  - Performed testing of Final Product using network inputs with 10 or more nodes.

4 Link State Routing Protocol

- Link State Routing Protocol used in packet switching networks for computer communications.
- The link-state protocol is performed by every switching node in the network that means nodes that are prepared to forward packets in the Internet, these are called routers.
- The basic concept of link-state routing is that every node constructs a map of the connectivity to the network, in the form of a graph, showing which nodes are connected to which other nodes.
- Each node then independently calculates the next best logical path from it to every possible destination in the network.
- The collection of best paths will then form the node's routing table.
- This protocol will used to find the shortest paths between nodes in graph.
- It will find the shortest path from source node to every other node.
- It can also be used for finding the shortest paths from a single node to a single destination node by ending the algorithm once the shortest path to the destination node has been determined.
- Examples of link-state routing protocols include Open Shortest Path First (OSPF).

4.1 Dijkstra's Algorithm

- Let the node at which we are starting be called the initial node. Let the distance of node Y be the distance from the initial node to Y. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.
- Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
- Set the initial node as current. Mark all other nodes unvisited. Create a set of all the unvisited nodes called the unvisited set.
- For the current node, consider all of its unvisited neighbors and calculate their tentative distances. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one.

- When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again.
- If the destination node has been marked visited or if the smallest tentative distance among the nodes in the unvisited set is infinity then stop. The algorithm has finished.
- Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and repeat the procedure again.

Using the above, the algorithm was designed to follow the steps outlined below (also see the process_dijkstra() function):

For all nodes in the graph do the following:

- Run Dijkstra's Algorithm
- Find Shortest path to all nodes
- Create Routing Table from interface table and path information

The Dijkstra's Algorithm's implementation for a given source was as follows:

- Create a list of all nodes in graph.
- Create a dictionary to store names and distances of nodes not yet visited.
- Initialize all distances to infinity and the source to zero.
- Keep a track of current node and initialize it with source.
- Create a dictionary to store distances of nodes which have been visited.
- Create another dictionary to store information of parent node.
- Iterate through unvisited set till it's empty.
  - Extract adjacent nodes and distances from current node.
  - Check whether neighbour is already visited.
  - If Unvisited then update distance.
  - If new distance is lower then store this information.
  - Remove the current node from unvisited set.
  - Sort the distances of neighbouring nodes and choose the one which is closest.

5 Quick Start Guide

This program enables simple processing and visualization of router networks and includes:

- Read in data from a text file
- Run a pathfinding algorithm to establish paths between nodes
- Form routing tables for each node
- Add, modify or delete connections between nodes

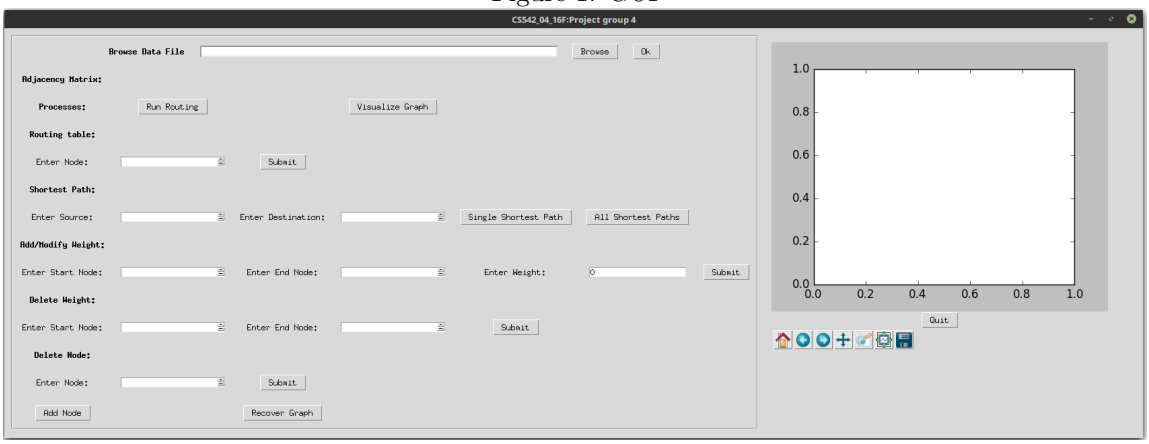
- Add new nodes
- Remove nodes
- Recover deleted nodes
- Visualize Graph

A detailed description of the User Interface is given in the next section.

6 Graphical User Interface

The description of all the features of the GUI are listed below. Clickable buttons are typeset in bold in the description. Please refer to the source code to understand functions and processes linked to the front-end. The GUI is shown in Figure 1.

Figure 1: GUI



BROWSE DATA FILE Contains a text field to specify full path of the file to be read. User can type in the entire path or alternatively use the **Browse** button to bring up a directory window to interactively select a network text file. Press the **OK** button to process the file.

NOTE: Sanity Checks are not included so please make sure you select the right file.

Figure 2: Browse for Data

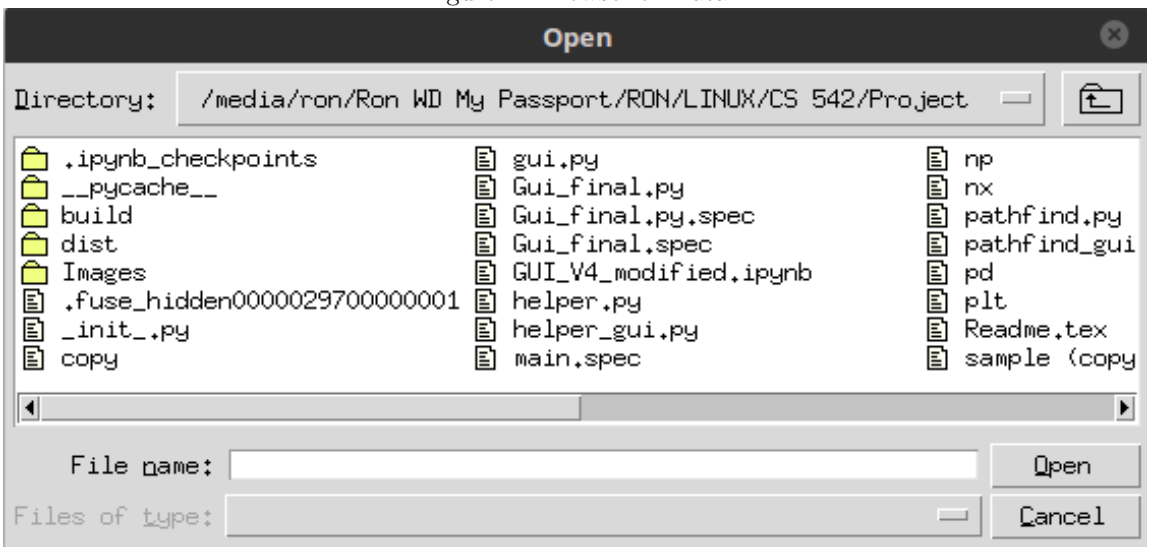
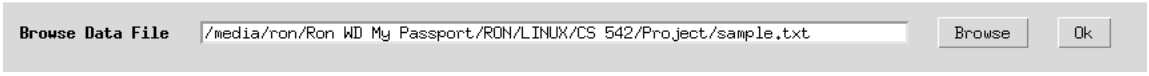


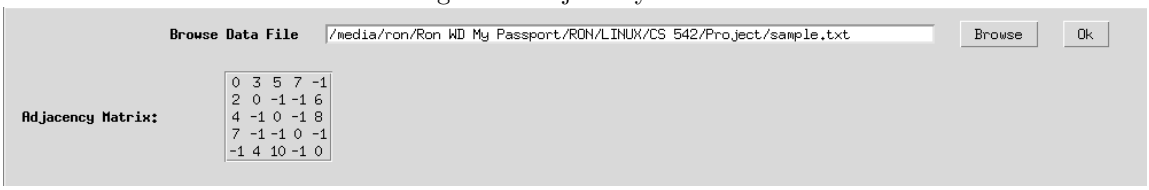
Figure 3: Selected Path to File



ADJACENCY MATRIX Displays the network matrix once the file has been processed.

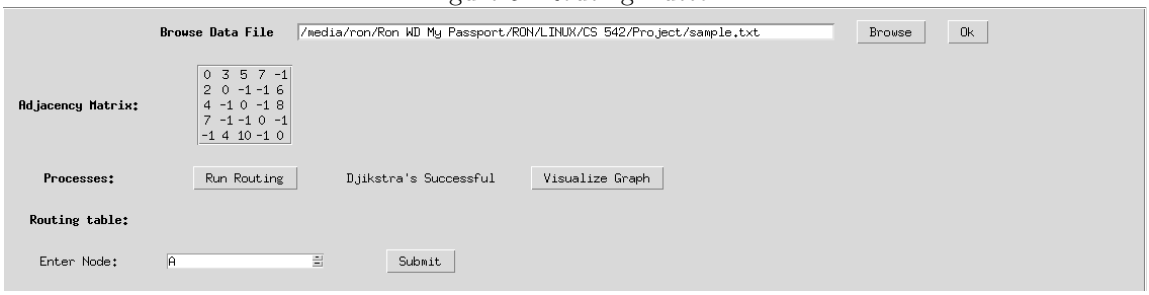
NOTE: The matrix is displayed only once when the file is read and processed.

Figure 4: Adjacency Matrix



RUN ROUTING Clicking on the **Run Routing** starts the routing process on the graph. This includes assigning interfaces to each outgoing edge of a node, runnning the *Dijkstra's Shortest Path* algorithm and creating a *Routing Table* for each node.

Figure 5: Routing Button

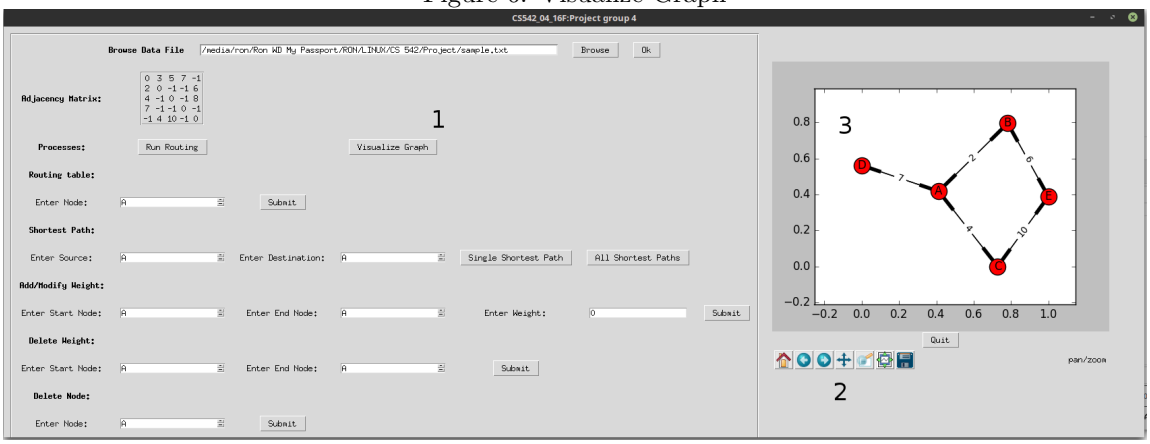


VISUALIZE GRAPH Click on **Visualize Graph** to plot the network in the Plot Widget on the right. One can use the toolbar to change orientation or save the graph.

NOTE 1: The graph plotting module has no option for parallel edges hence only a single edge is displayed. Direction of edges are denoted by thicker lines near the nodes.

NOTE 2: Refering from above, only one weight is displayed instead of two for bi-directional.
NOTE 3: Due to poor interfacing between *matplotlib* and *tkinter*, to update graph in window please first click on **Visualize Graph** (1),then on any toolbox widget (2) and then on the graph (3) (see figure 6 below).

Figure 6: Visualize Graph



ROUTING TABLE This will let you choose the node for which you wish to view the routing table. Select the appropriate node from the *Up* and *Down* arrows and click on **Submit**. The Table will be displayed on the right side of the window.

Figure 7: Routing Table

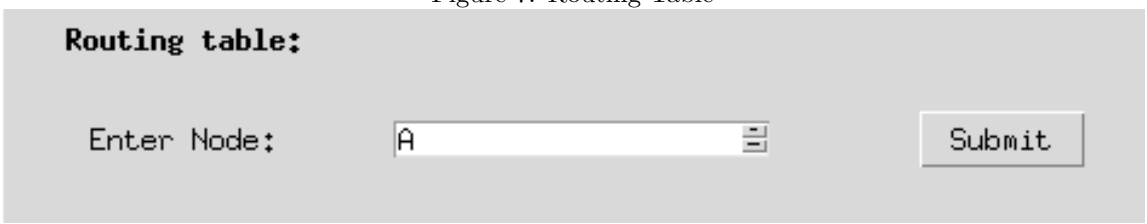
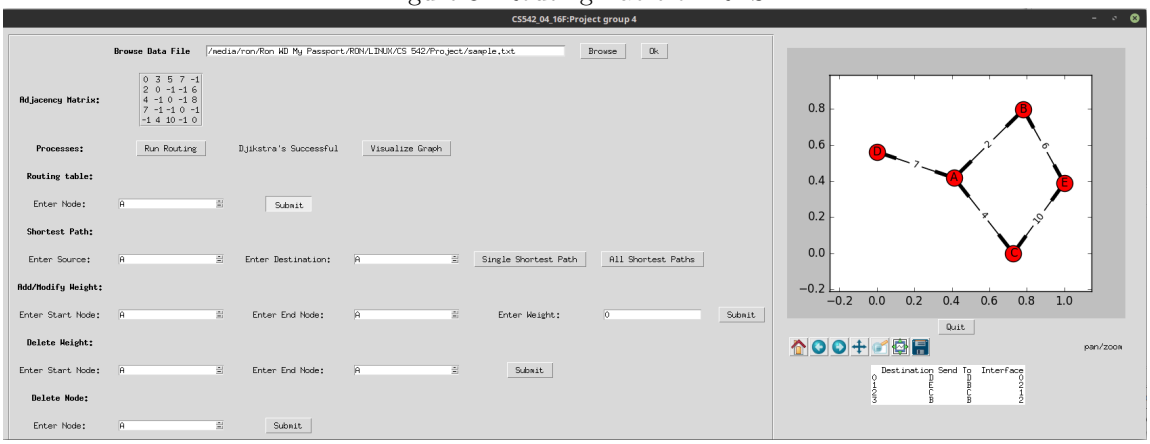
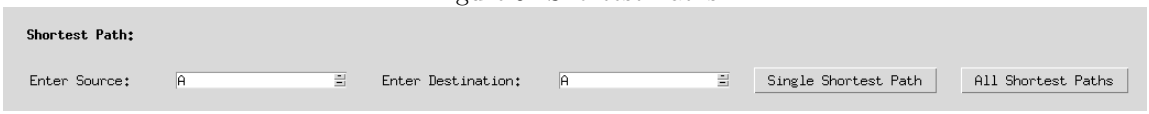


Figure 8: Routing Table on RHS



SHORTEST PATH This lets you choose the *source* and *destination* nodes for whom you want the shortest path displayed. Click on **Single Shortest Path** to display the shortest path and distance or click on **All Shortest Paths** to display multiple shortest paths and distances (if available). Both the outputs are displayed on the right side of the window.

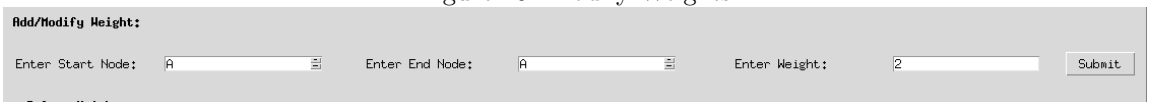
Figure 9: Shortest Paths



ADD/MODIFY WEIGHT This lets you modify the weight between the *start* and *end* node. If an edge already exists then it is modified otherwise a new edge is added. Click on **Submit** to apply changes.

NOTE: Keep in mind that the graph is directed i.e. you need to explicitly modify weights from source to destination and from destination to source if a bidirectional link is required.

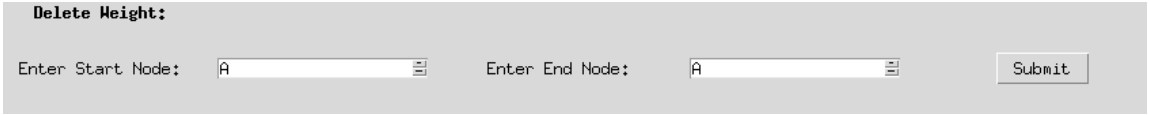
Figure 10: Modify Weights



DELETE WEIGHT This lets you delete an edge between the *start* and *end* node. Click of **Submit** to apply changes.

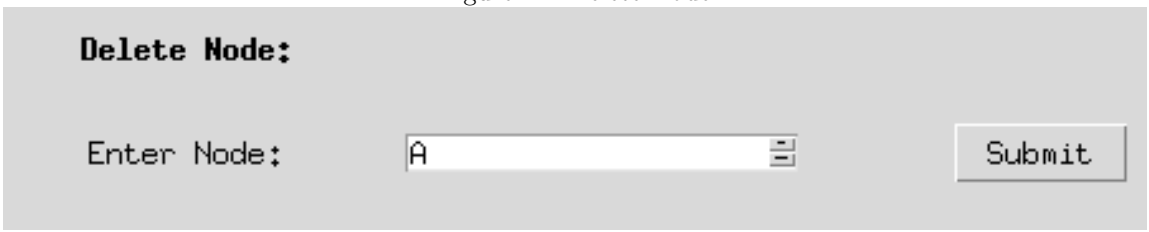
NOTE: Keep in mind that the graph is directed i.e. you need to explicitly delete weights from source to destination and from destination to source if required.

Figure 11: Delete Weights



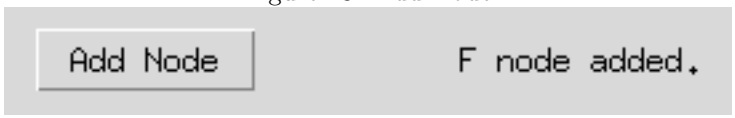
DELETE NODE This lets you delete a particular node you selected. Click on **Submit** to apply changes.

Figure 12: Delete Node



ADD NODE Click on **Add Node** to add a new node to the network. The name of the node is automatically assigned as the next alphabet in sequence. This outputs which node has been added.

Figure 13: Add Node



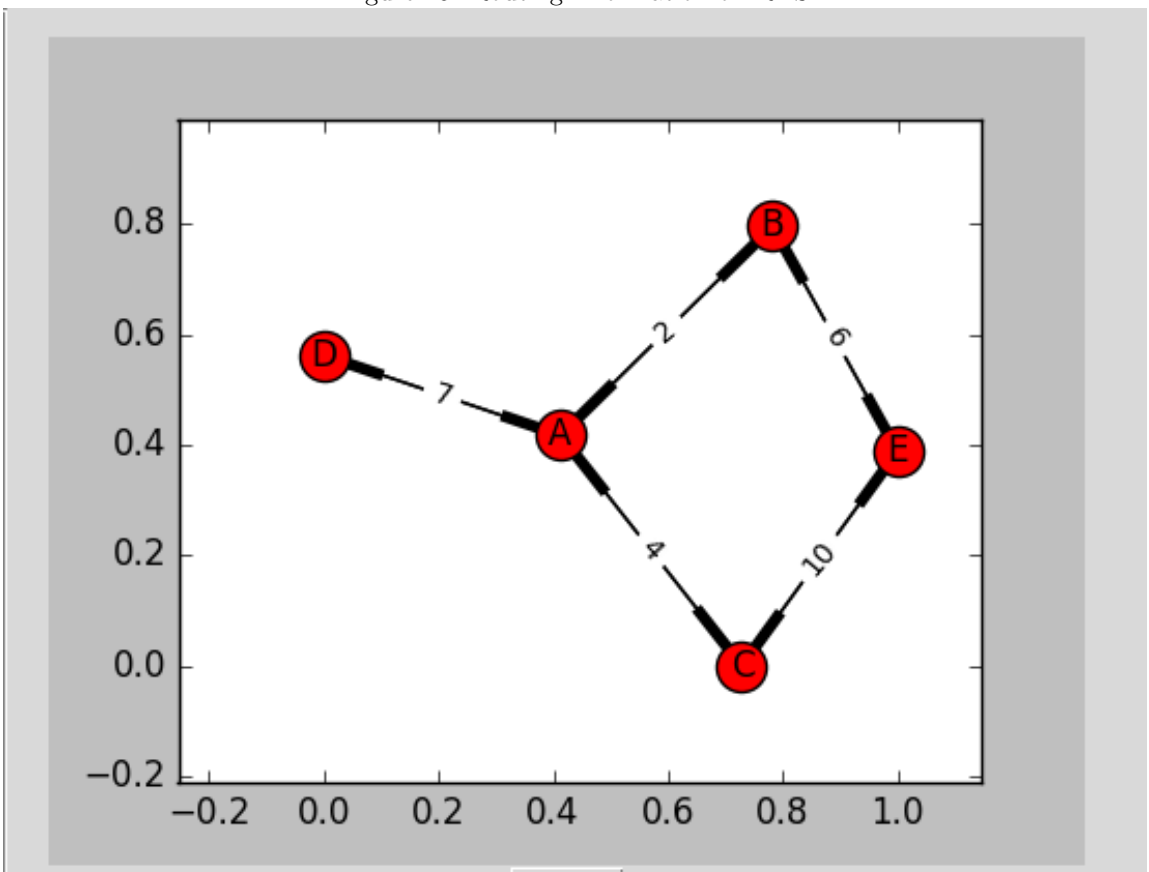
RECOVER GRAPH Click on **Recover Graph** to add back all the nodes that had been previously deleted.

Figure 14: Recover Graph

QUIT click on **Quit** to close the program. Alternatively, one can use the standard close button on the top right corner.

Figure 15: Quit

Figure 16: Routing Information on RHS



Quit

pan/zoom

	Destination	Send To	Interface
0	D	D	0
1	E	B	2
2	C	C	1
3	B	B	2

Shortest path is: ['A', 'B', 'E']

Total Distance is: 9

All Shortest paths are:
[['A', 'C', 'E'], ['A', 'B', 'E']]

Total Distances are: [13, 9]

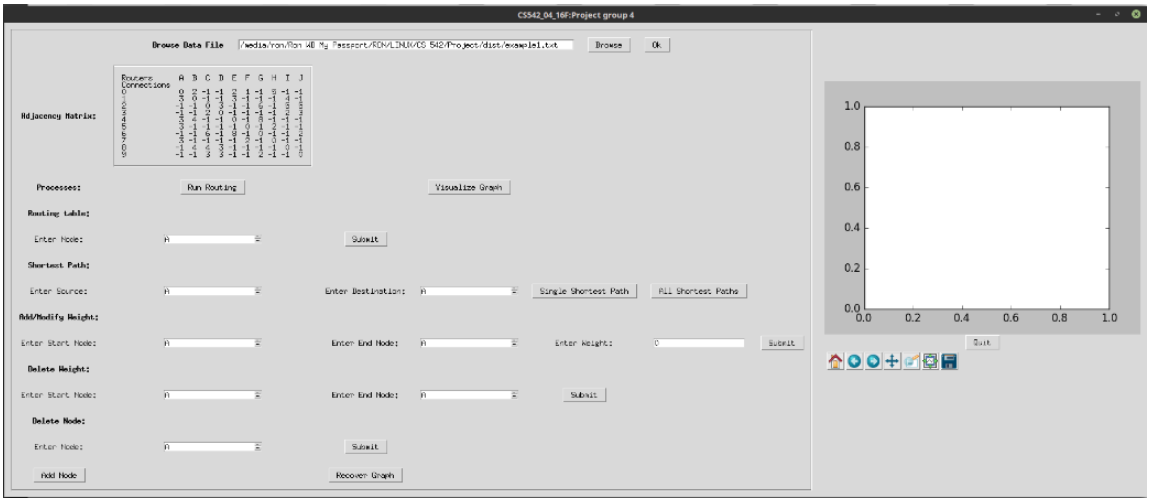
7 Test Cases

For the test cases we used five different test files (available in project package).

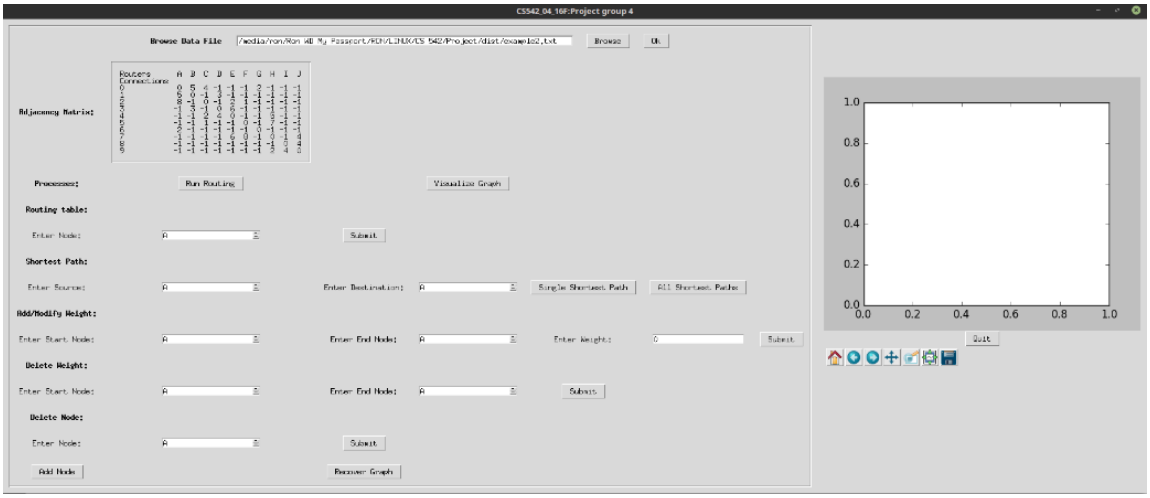
7.1 Test Case 1

Description: Open file and display Adjacency matrix.
Expected output: Adjacency matrix of the network should be displayed.

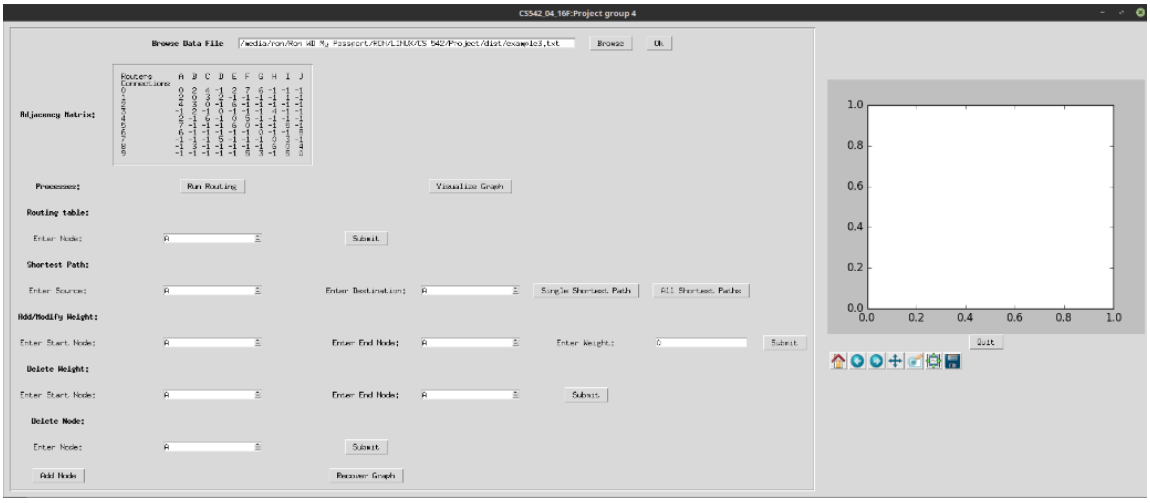
- Test Case 1.1: Example 1



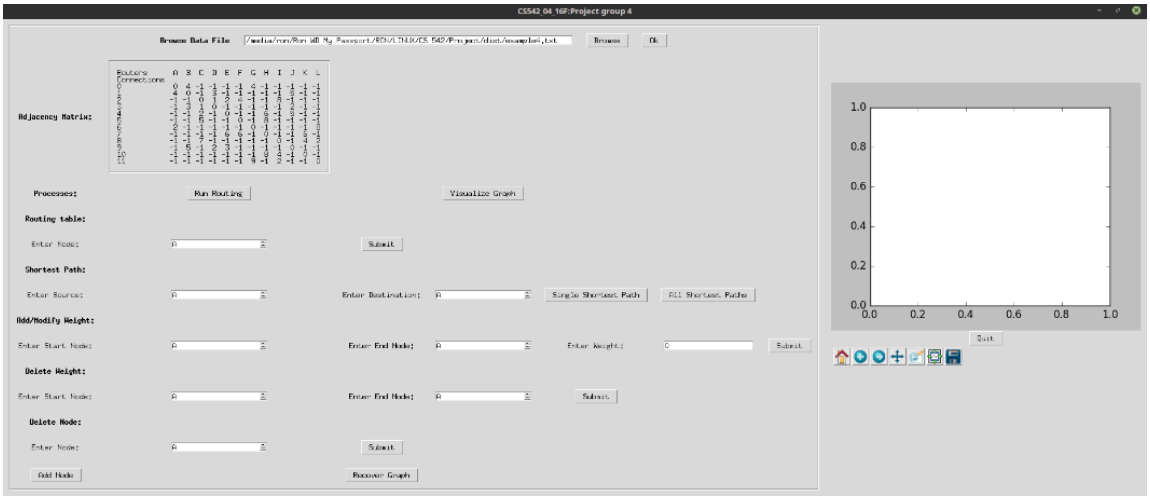
- Test Case 1.2: Example 2



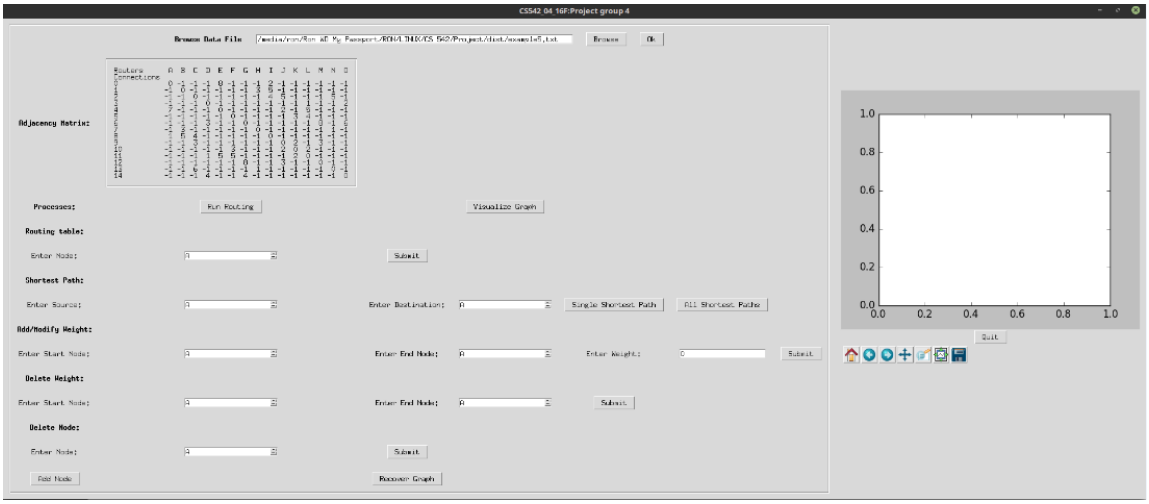
- Test Case 1.3: Example 3



- Test Case 1.4: Example 4



- Test Case 1.5: Example 5

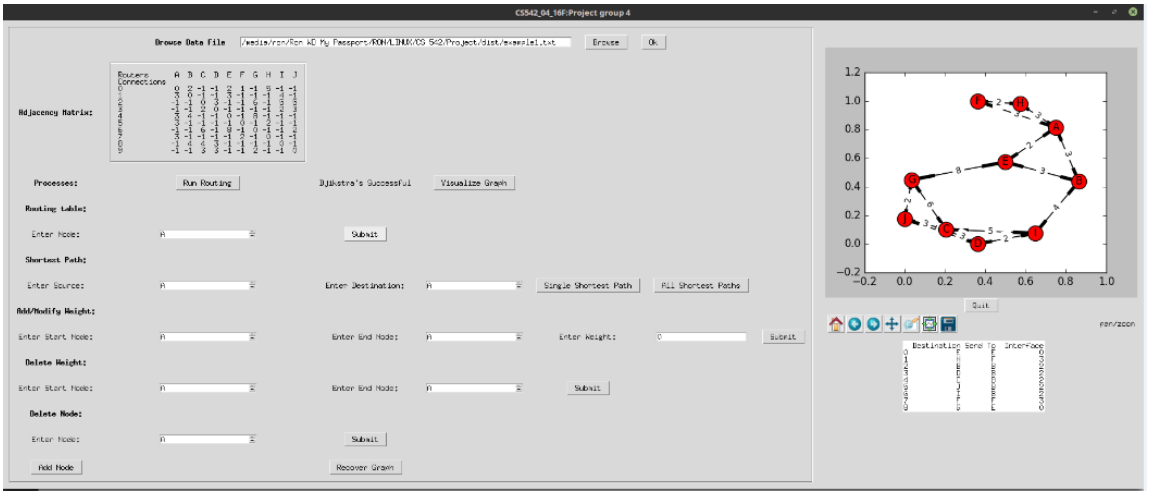


7.2 Test Case 2

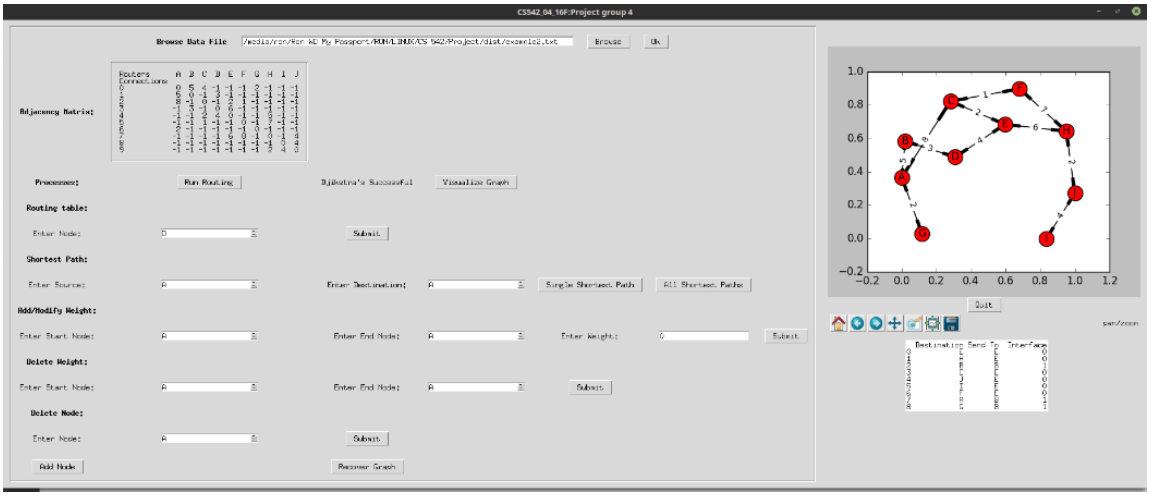
Description: Run Link State Algorithm and display topology graph and Routing Table.

Expected output: Topology graph of the entire network and Routing Table for a chosen node should be displayed.

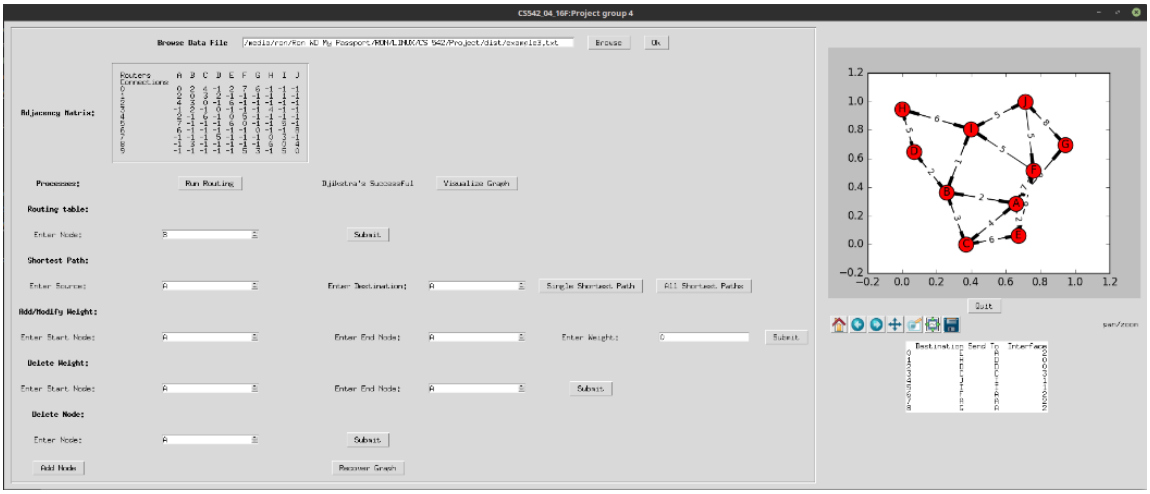
- Test Case 2.1: Example 1



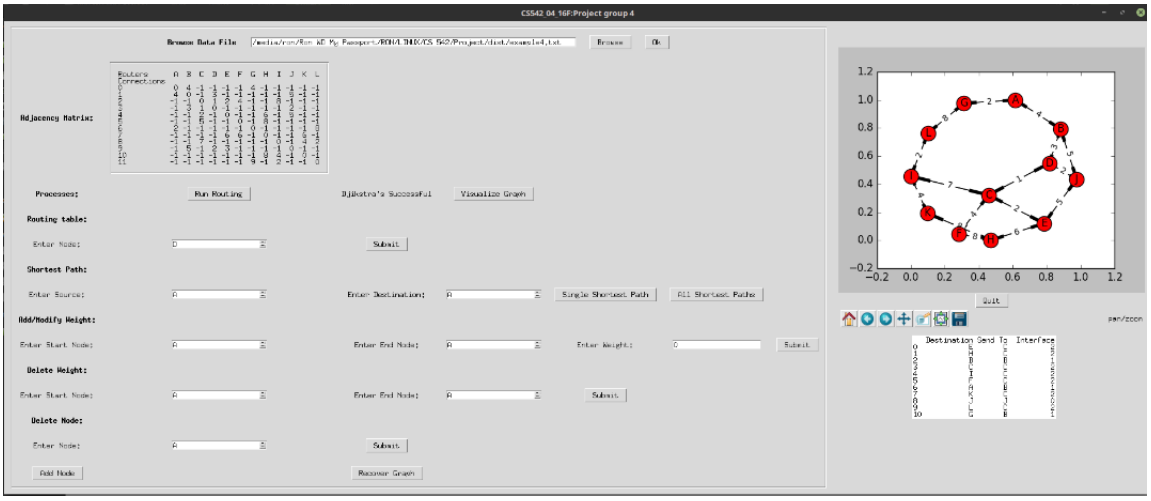
- Test Case 2.2: Example 2



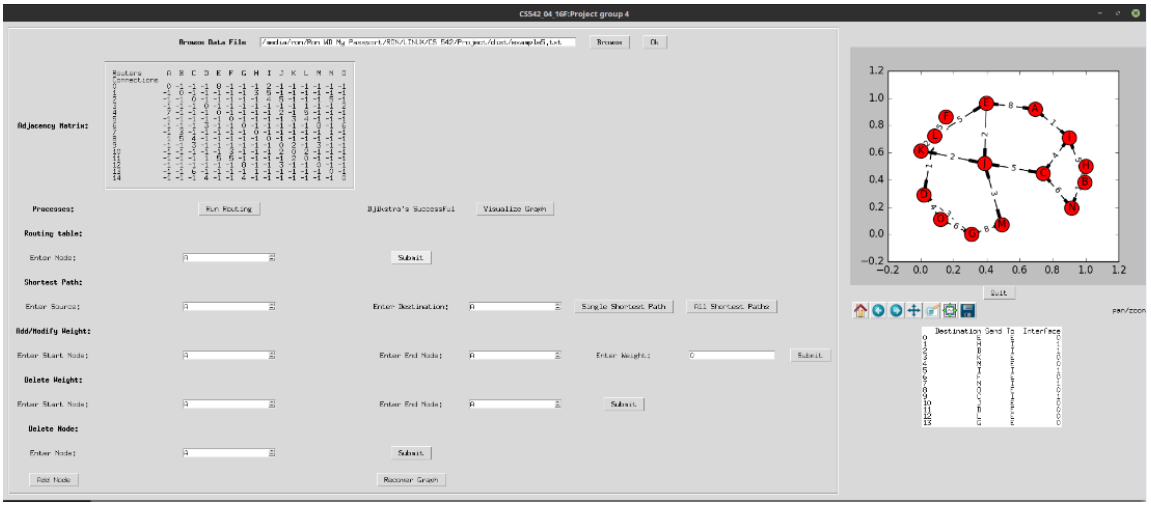
- Test Case 2.3: Example 3



- Test Case 2.4: Example 4



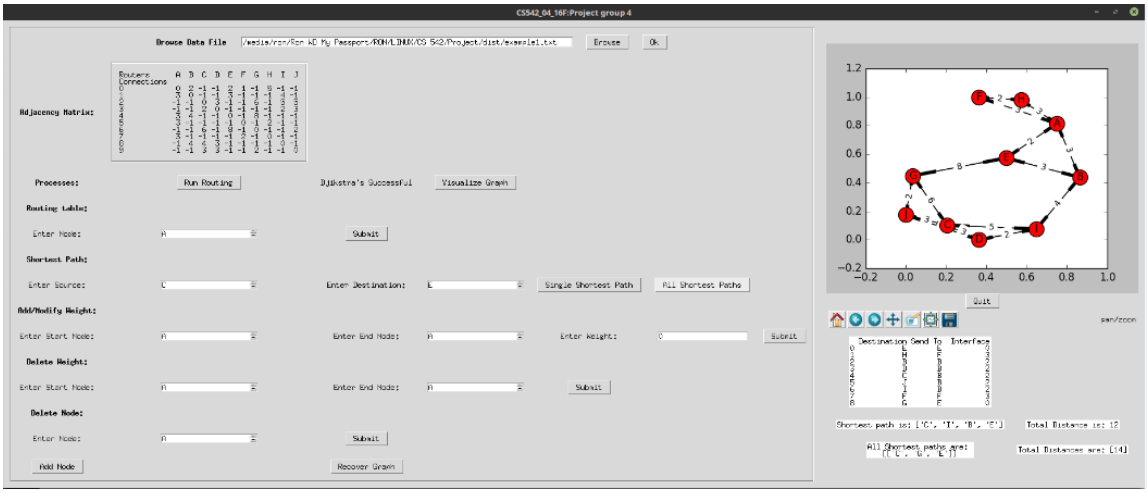
- Test Case 2.5: Example 5



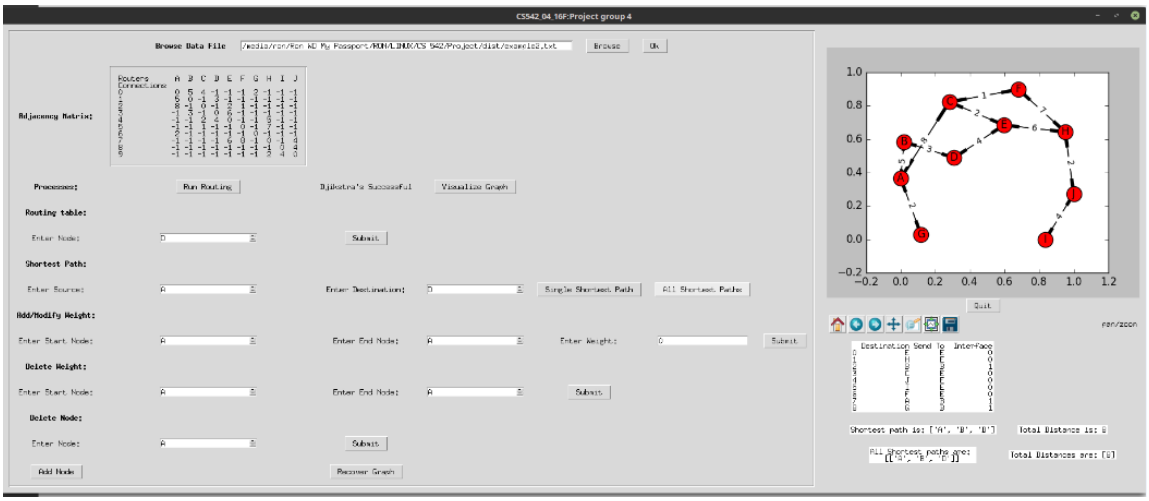
7.3 Test Case 3

Description: Display Shortest path from source to destination.
Expected output: Shortest path along with distance should be displayed.

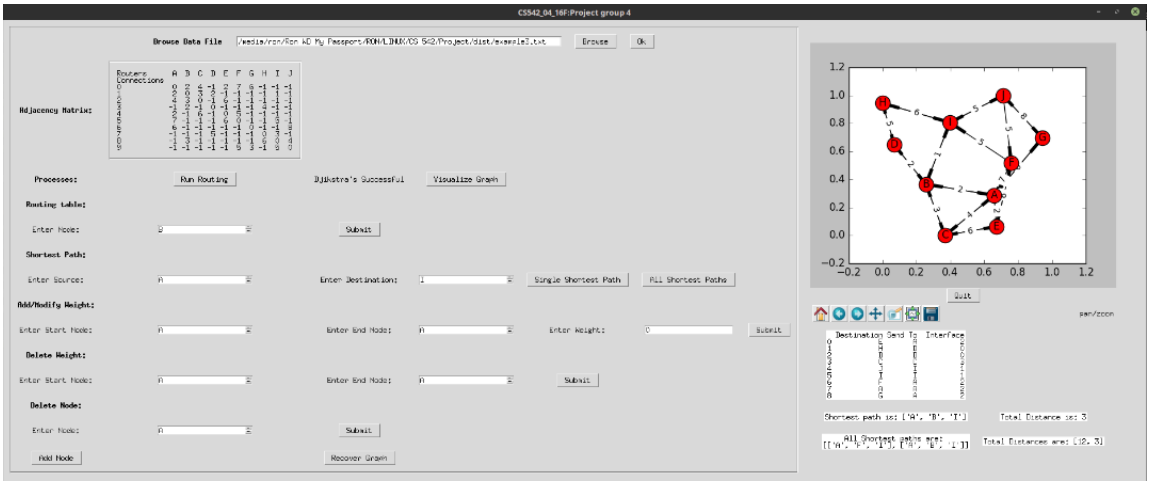
- Test Case 3.1: Example 1



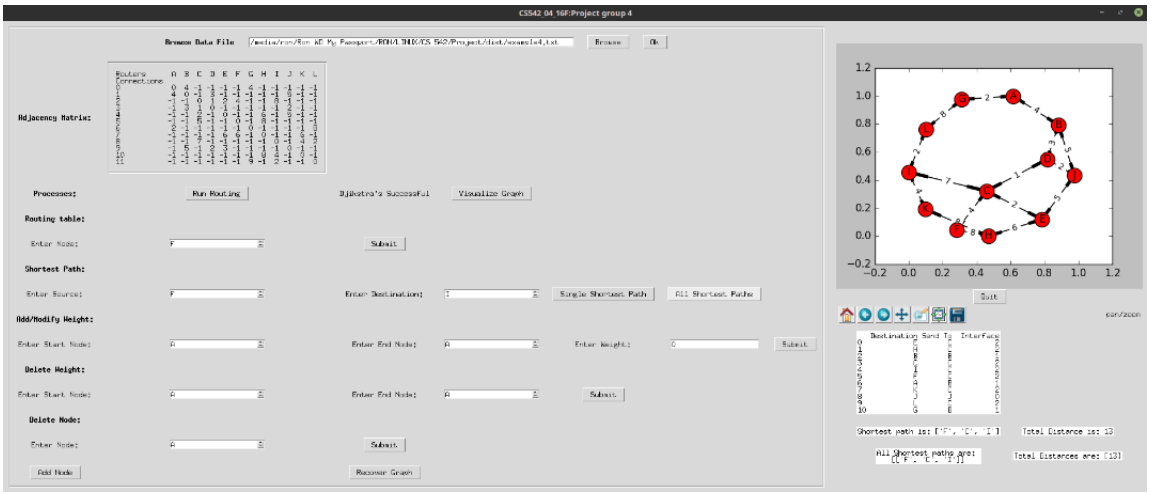
- Test Case 3.2: Example 2



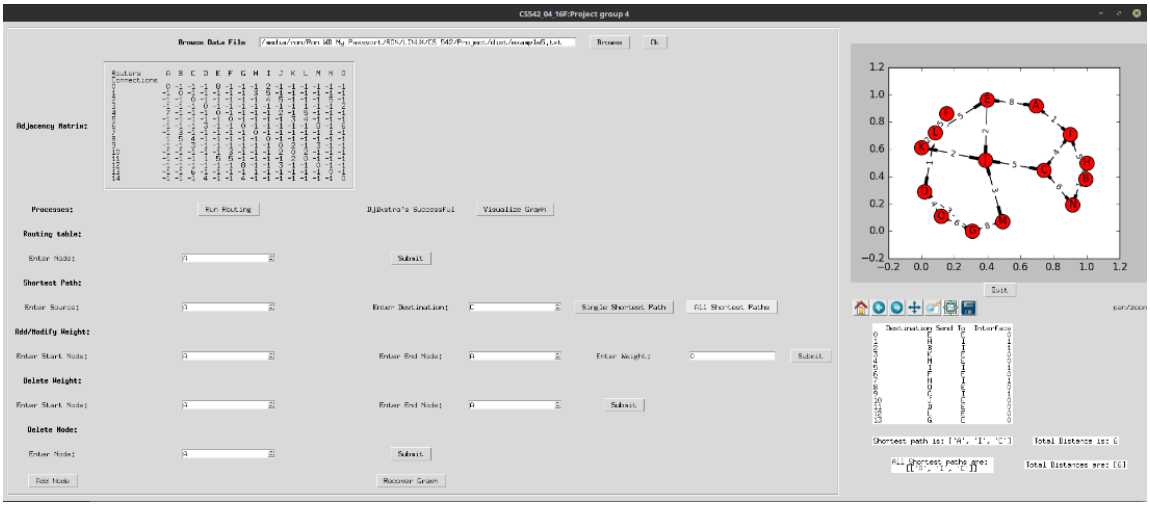
- Test Case 3.3: Example 3



- Test Case 3.4: Example 4



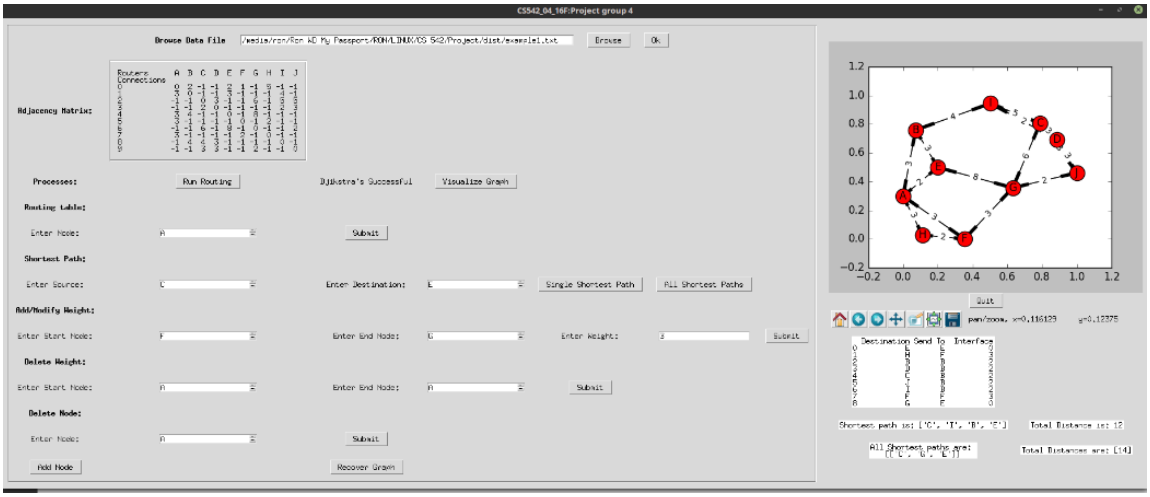
- Test Case 3.5: Example 5



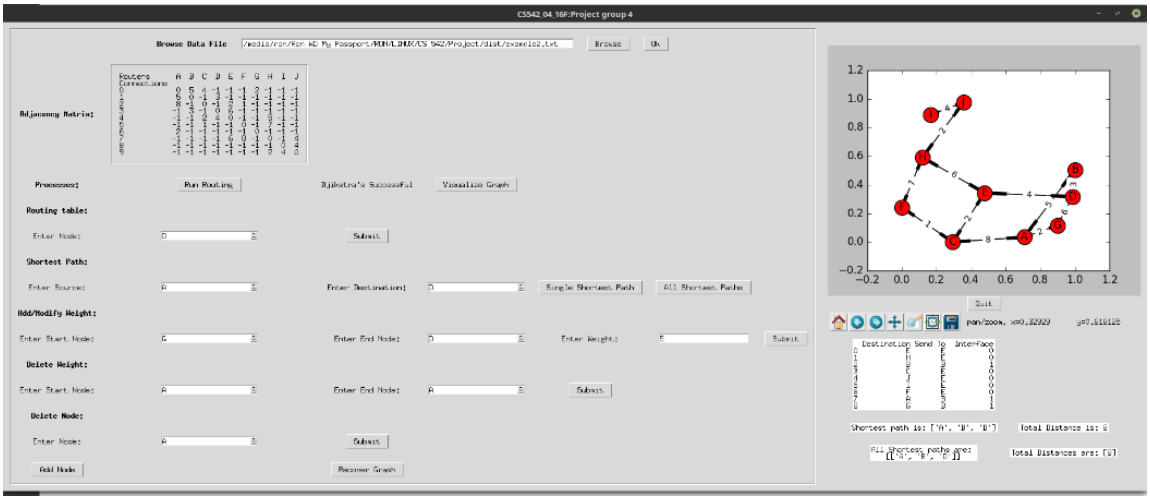
7.4 Test Case 4

Description: Add/Modify the weight of edge between two noded.
Expected output: Change should be reflected in the graph.

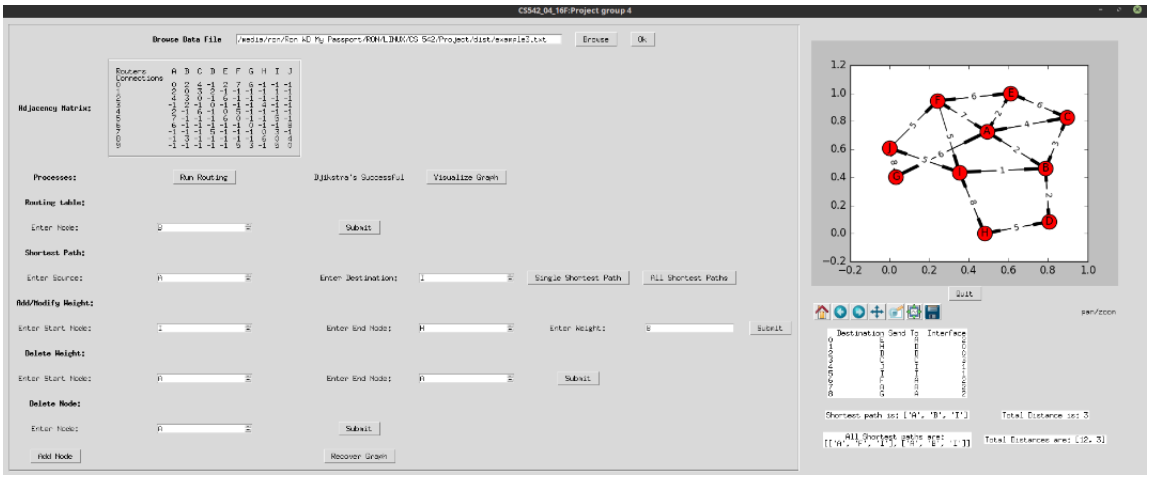
- Test Case 4.1: Example 1



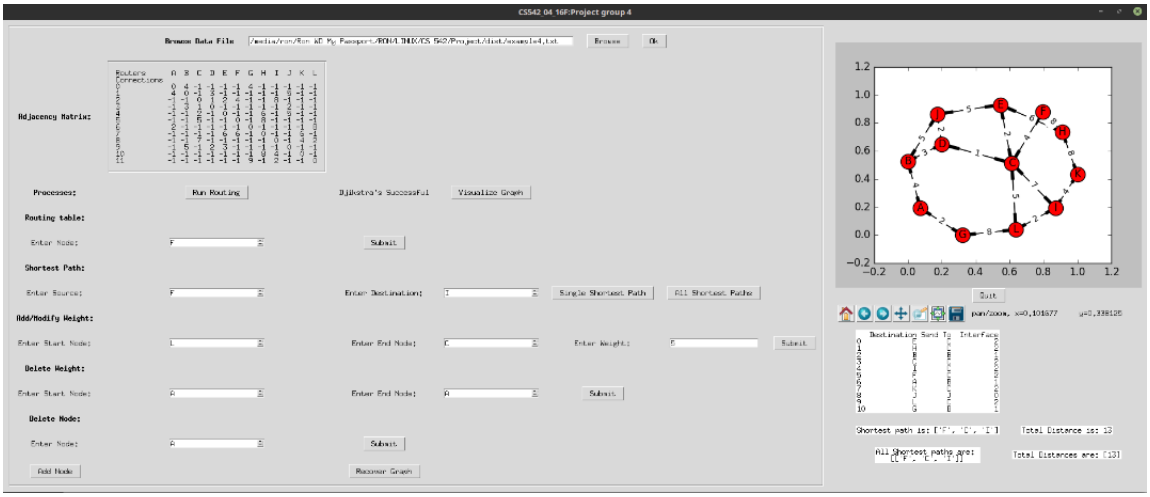
- Test Case 4.2: Example 2



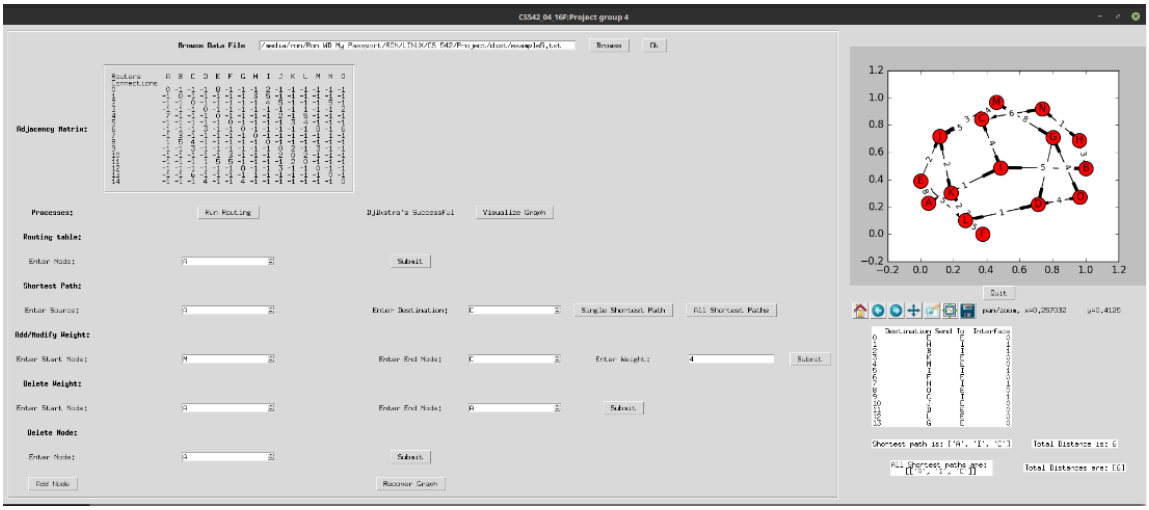
- Test Case 4.3: Example 3



- Test Case 4.4: Example 4



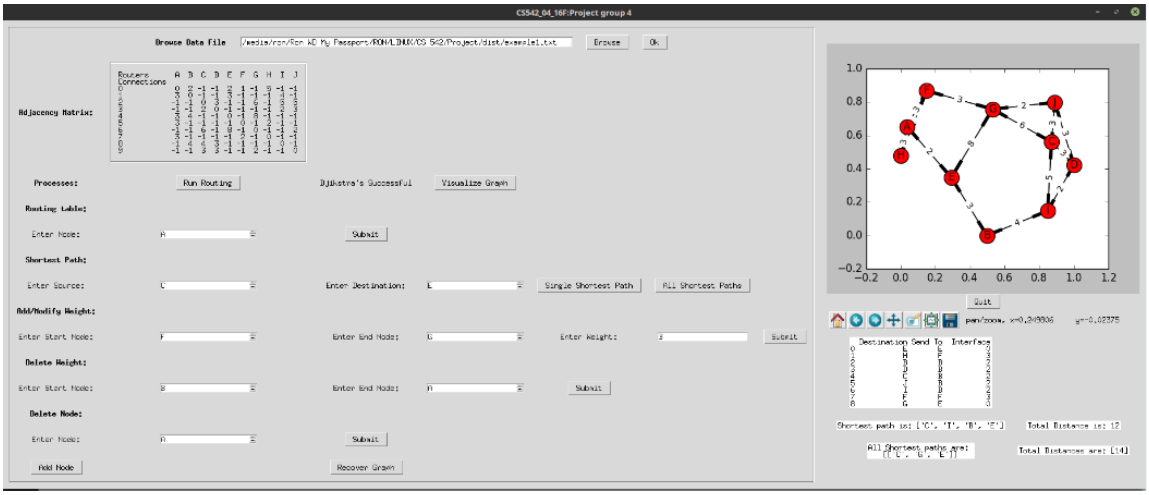
- Test Case 4.5: Example 5



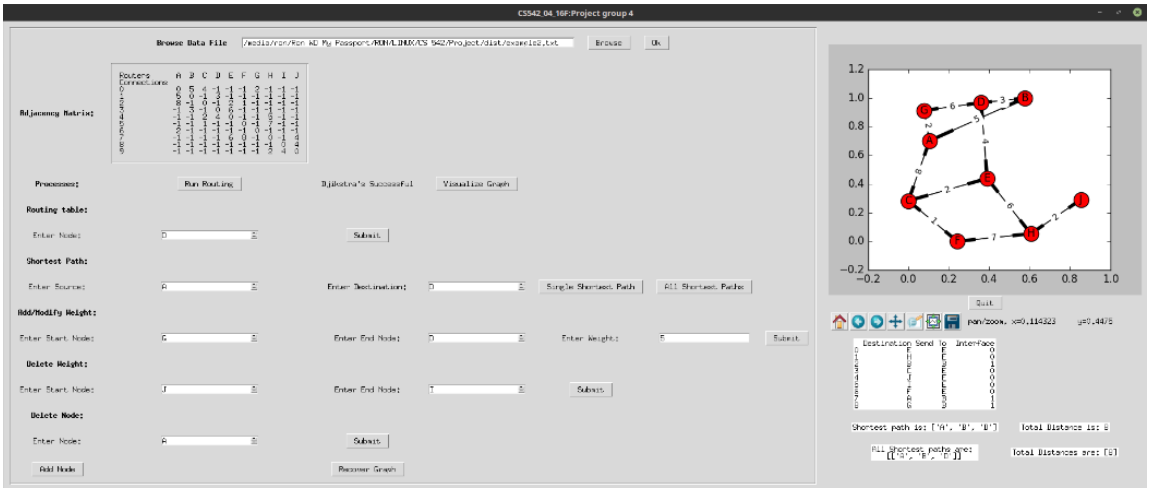
7.5 Test Case 5

Description: Delete an edge from the graph.
Expected output: Change should be reflected in the graph.

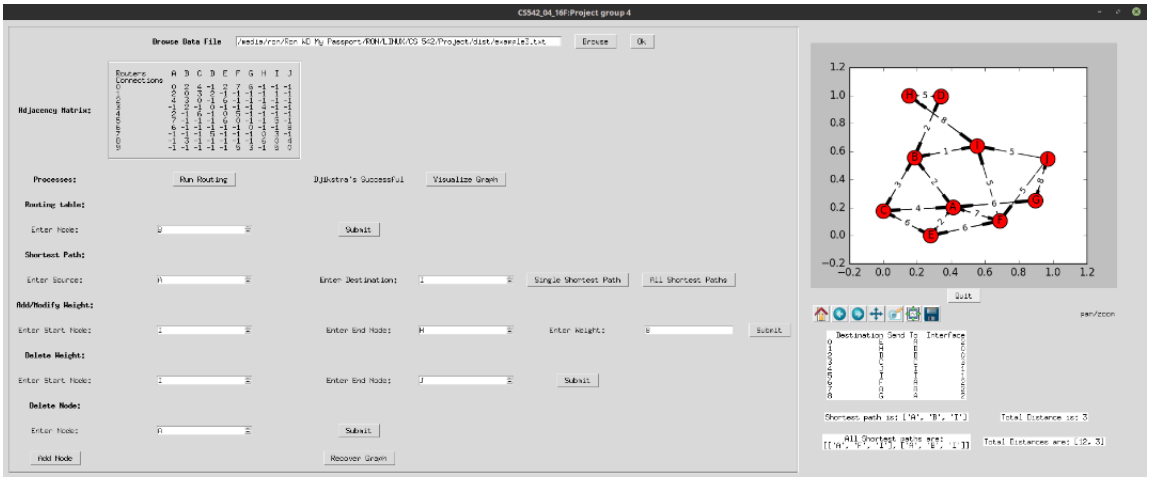
- Test Case 5.1: Example 1



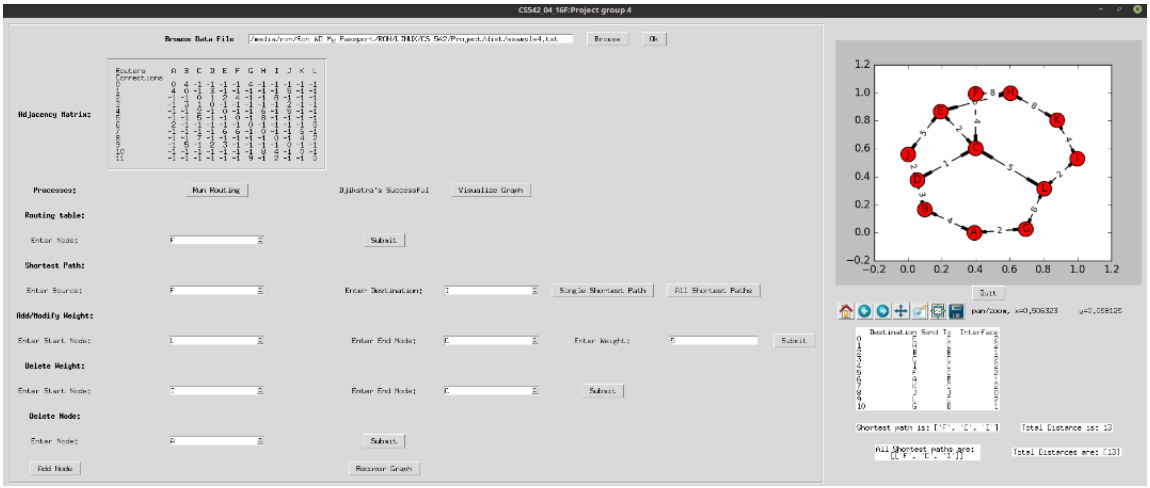
- Test Case 5.2: Example 2



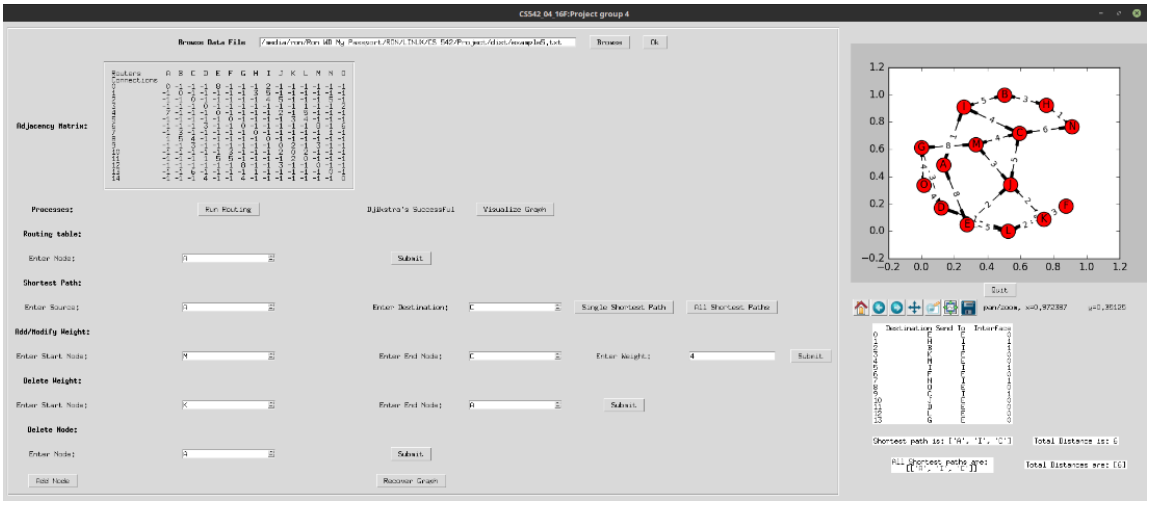
- Test Case 5.3: Example 3



- Test Case 5.4: Example 4



- Test Case 5.5: Example 5

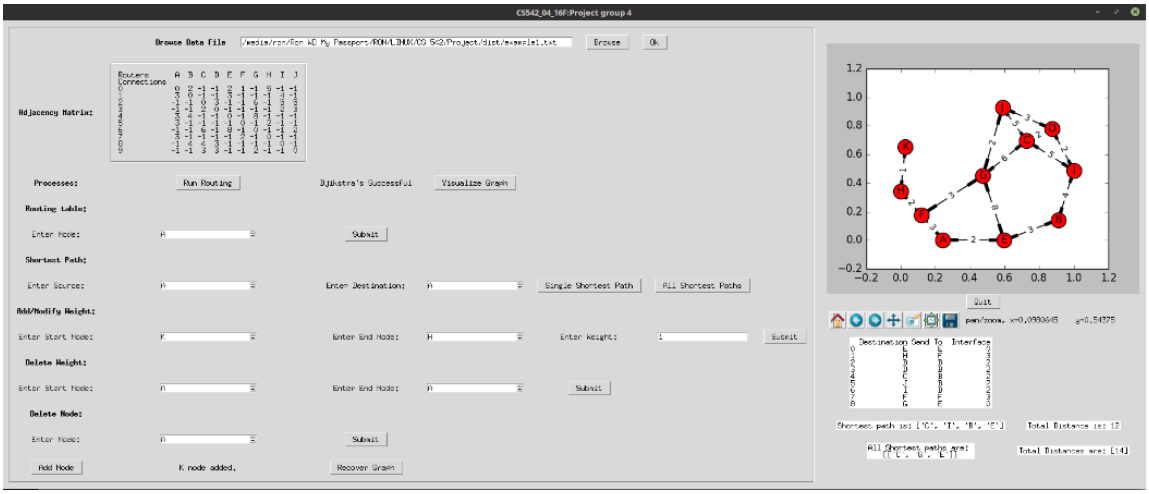


7.6 Test Case 6

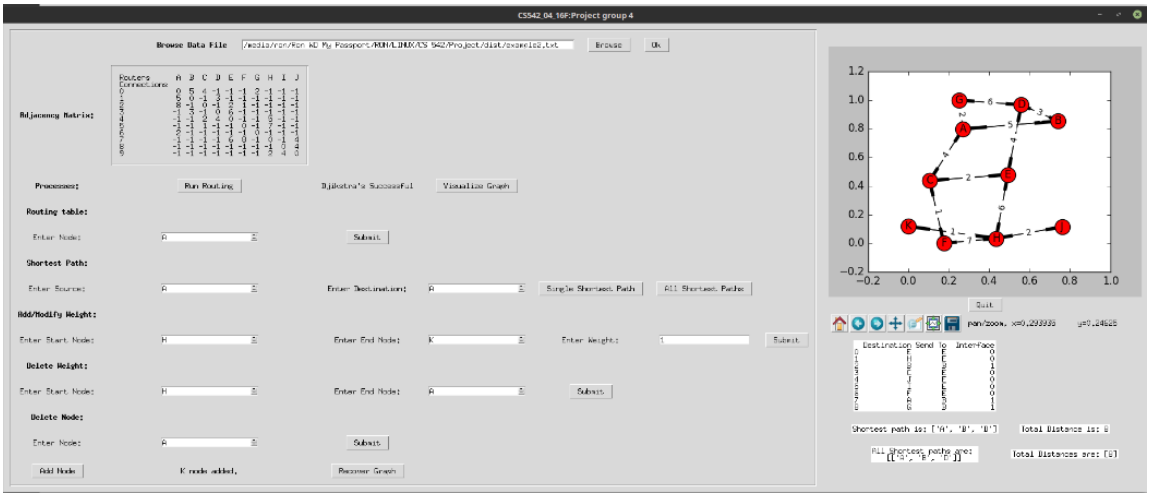
Description: Add new node to the graph.

Expected output: A new node (without edges) should be added.

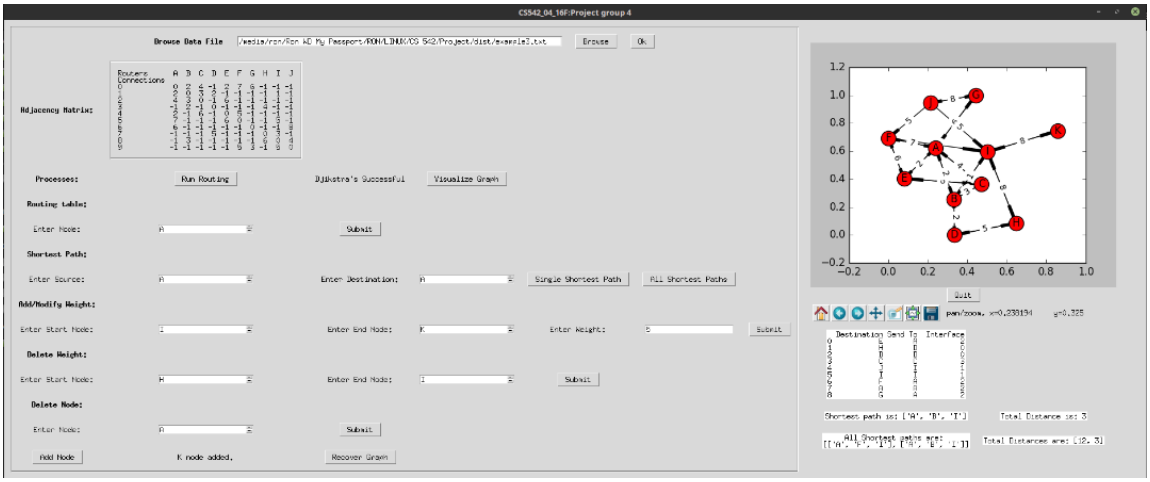
- Test Case 6.1: Example 1



- Test Case 6.2: Example 2

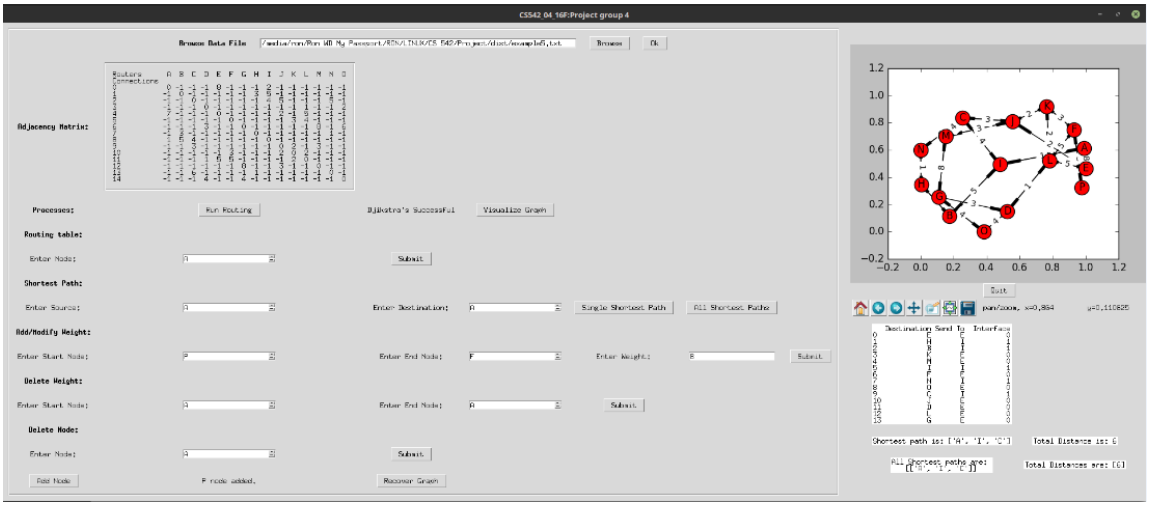


- Test Case 6.3: Example 3



- Test Case 6.4: Example 4

- Test Case 6.5: Example 5

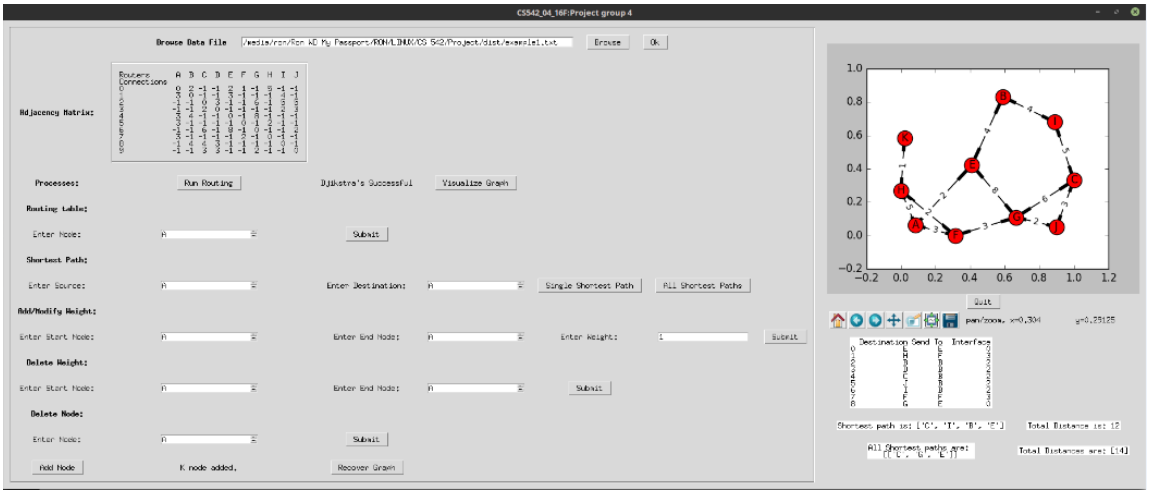


7.7 Test Case 7

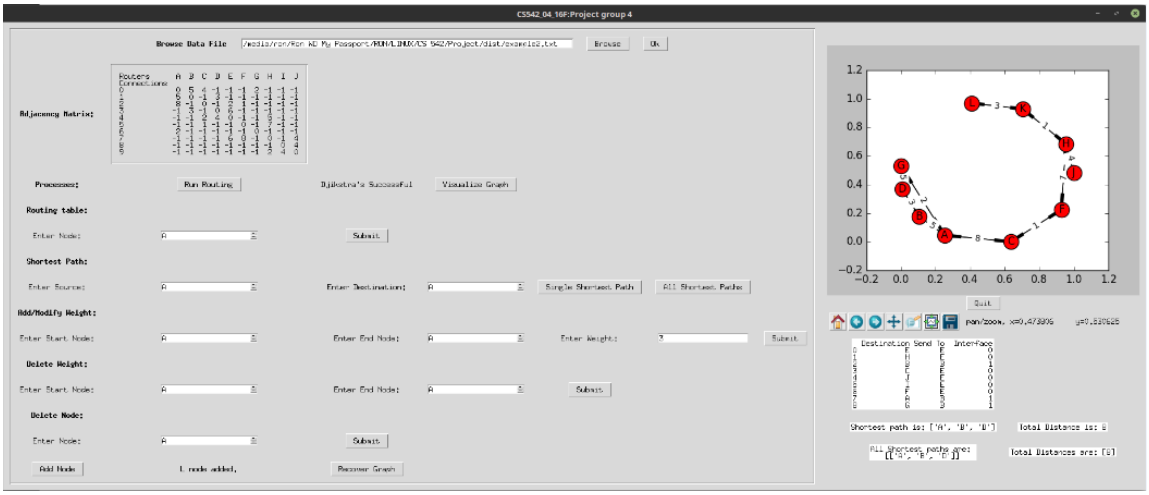
Description: Delete the existing node from graph.

Expected output: Node and its respective edges will be deleted from the graph.

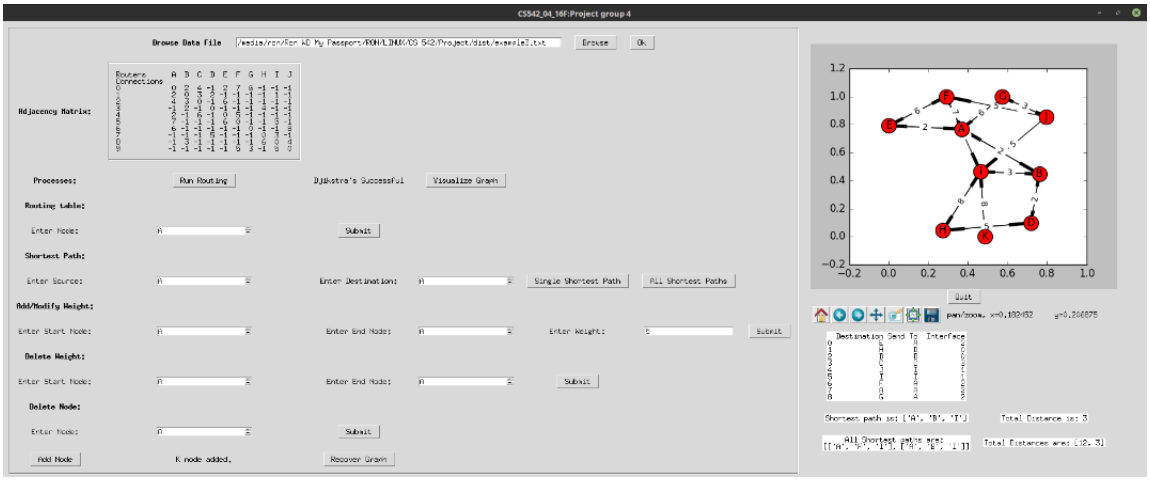
- Test Case 7.1: Example 1



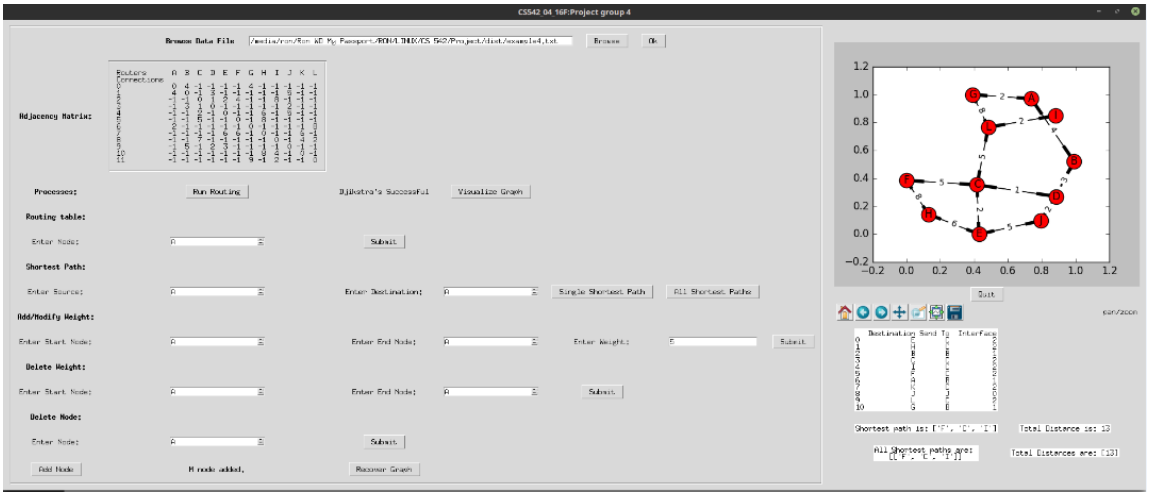
- Test Case 7.2: Example 2



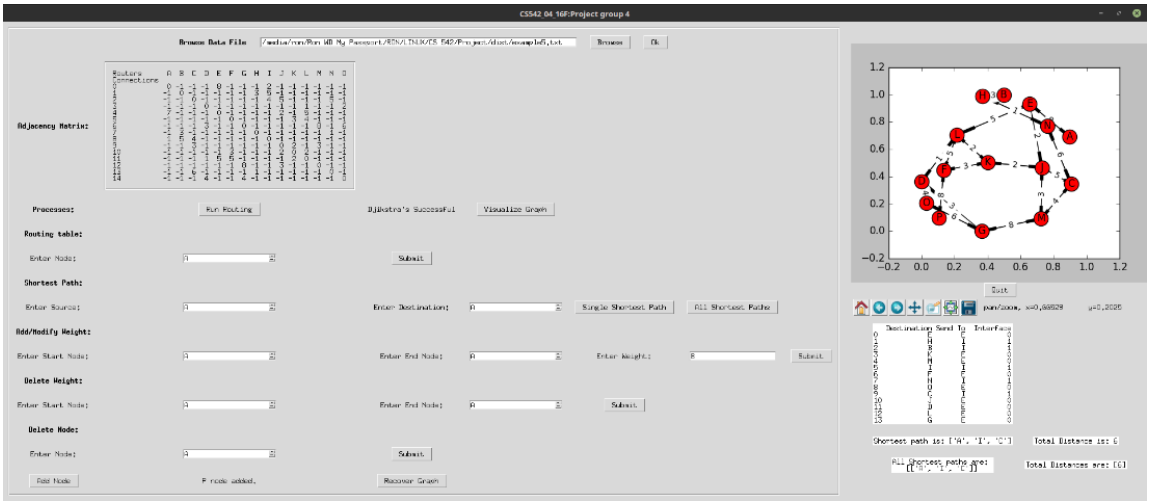
- Test Case 7.3: Example 3



- Test Case 7.4: Example 4



- Test Case 7.5: Example 5

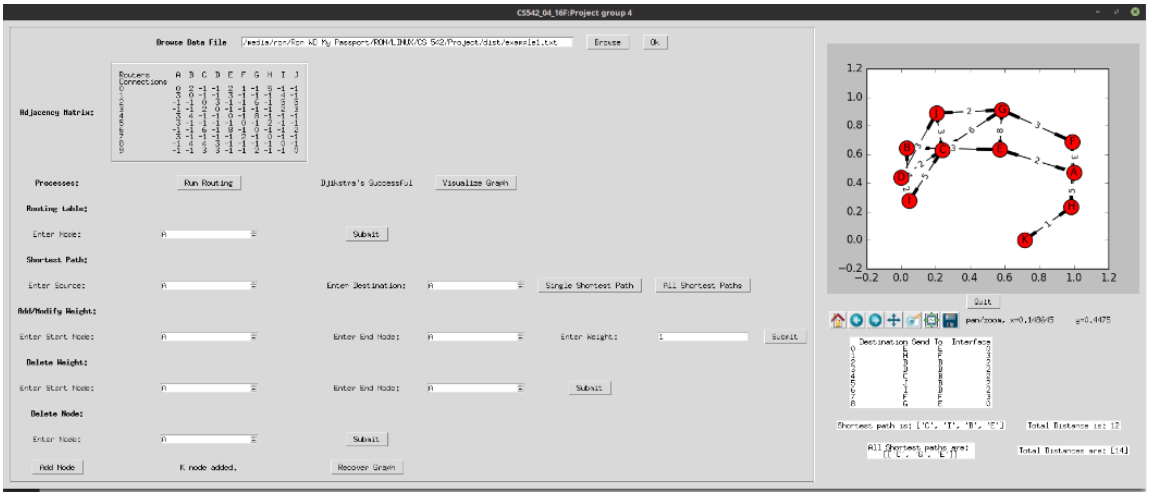


7.8 Test Case 8

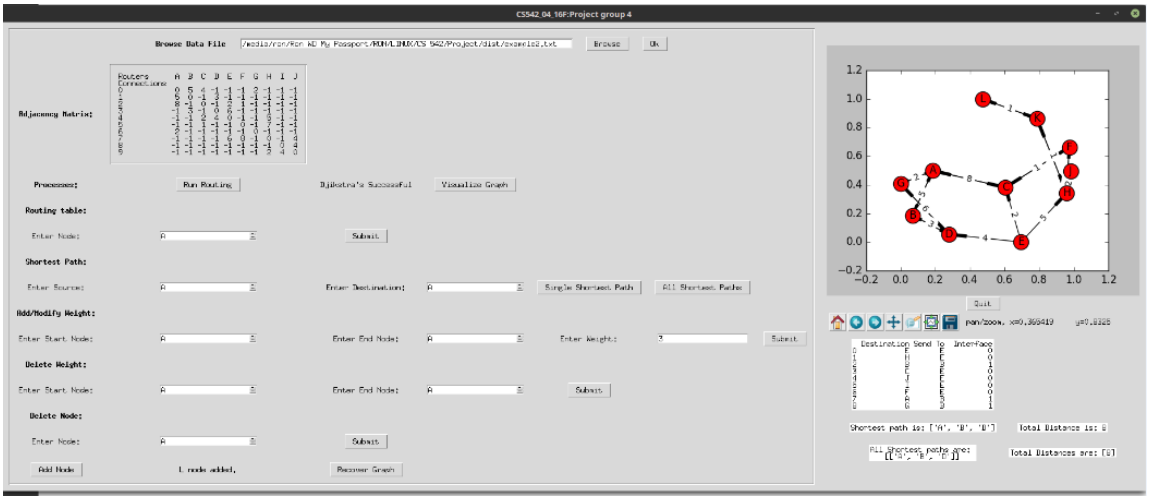
Description: Recover the graph.

Expected output: Any deleted nodes should be added back to the graph.

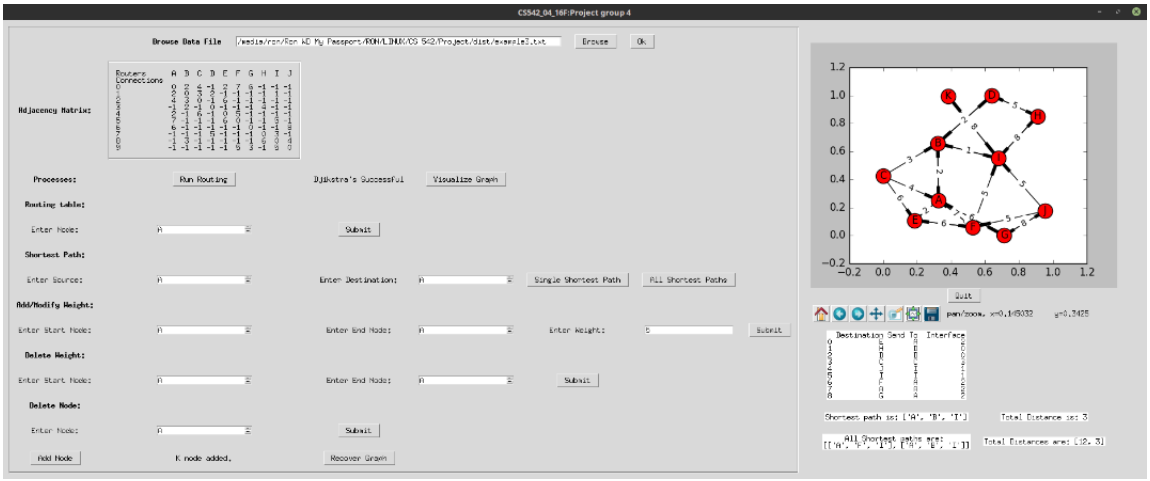
- Test Case 8.1: Example 1



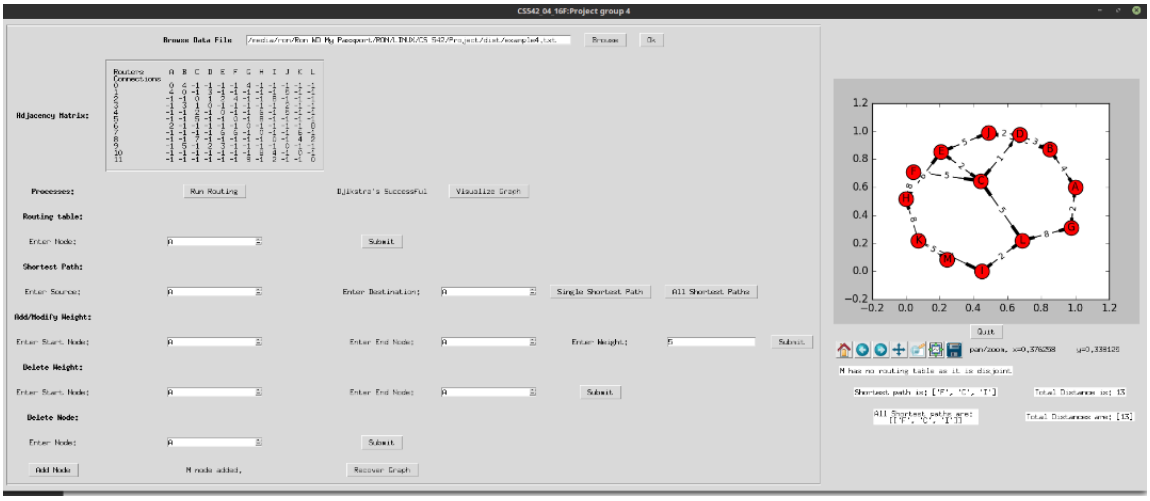
- Test Case 8.2: Example 2



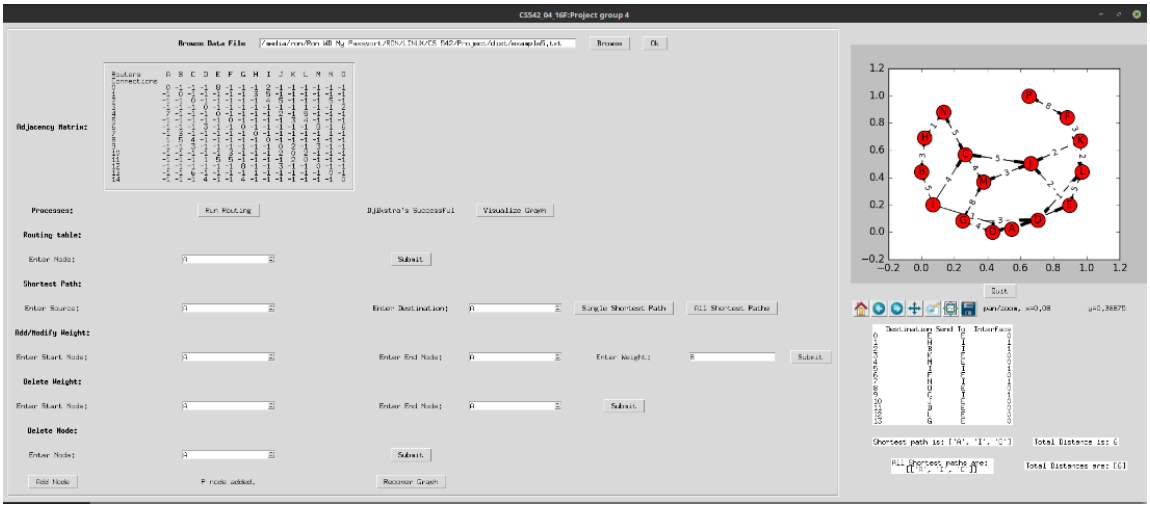
- Test Case 8.3: Example 3



- Test Case 8.4: Example 4



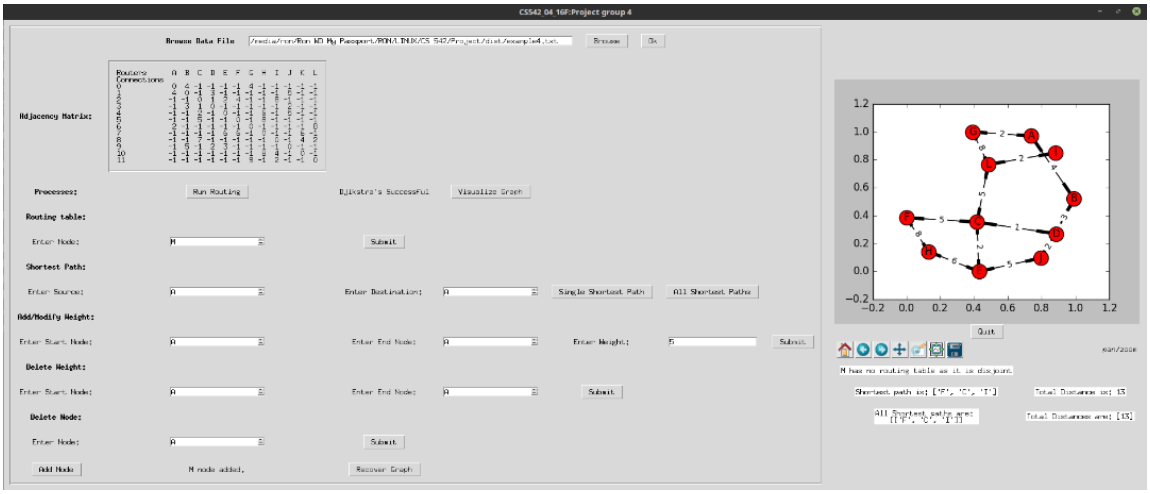
- Test Case 8.5: Example 5



7.9 Test Case 9

Description: If any node is disjoint.

Expected output: No Routing table will be displayed for that node.



8 Back-End Design

This section gives a brief description of the functions running in the back-end. Refer to Readme Section for location of the source code.

- **open_file()**

  It will open a file dialog box and we can choose any network file to process.

  Input: None

  Output: Open the file

- **read_data (file_name)**

  It will use pandas to read the given text file.

  Input: file_name(File selected from open_file)

  Output: Read the file and print adjacency matrix into GUI

- **mat2dict(data,adj_mat)**

  It will convert the matrix into python dictionary, as we are using dictionary for every other function to process the data.

  Input: data(table of data), adj_mat(array of the given data)

  Output: Create a dictionary, named graph

- **visualize(graph)**

  This function is used to display the topology graph in the canvas.

  Input: graph

  Output: Topology graph printed on canvas

- **mod_weight(graph)**

  It will add or modify an edge between two nodes.

  Input: graph

  Start_node, end_node, weights are taken from GUI

  Output: It will add or modify the weight

- **del_edge(graph)**

  It will delete the edge between two nodes.

  Input: graph

  Start_node, end_node, are taken from GUI

  Output: It will delete the edge

- **add_node(graph)**

  It will add a new available node to the graph.

  Input: graph

  Output: Node will be added to graph

- **remove_node(graph)**

  It will delete the node from the graph given by user.

  Input: graph

  Del_node (name of node) taken from GUI

  Output: Node will be deleted from the graph

- **recover_node()**

  It will restore any nodes previously deleted from the graph.

  Input: None

  Output: Original graph before deletion.

- **node_interface(graph)**

  It will assign an interface number to each outgoing link of each node.

  Input: graph

  Output: Interface table for each node

- **process_dijkstra()**

  It will the run the dijikstras algorithm on entire graph.

  Input: None

  Output: all_distances, parent (parent for each node), path (path from every node), route (routing table)

- **dijkstra(graph,start)**

  Dijkstra's algorithm core function

  Input: graph, start (starting node)

  Output: all_distances, parent (parent for each node)

- **shortest_path(src,parent,distances)**

  In this function all the possible paths will be found from each node to every other possible node.

  Input: Src (Source), parent (parent of node) distances (distances of source to every other possible node)

  Output: path (path from every node)

- **routing(Path,src,interface)**

  It will find the routing table for each node present in the graph

  Input: path (path from every node), Src (Source), interface (interface of the node)

  Output: route (routing table)

- **single_path(all_path,all_distances)**

  It will give the single shortest path from source node to destination node.

  Input: all_path (all possible paths), all_distances (all given distances given in the graph)

  Src (source node), dest (destination node) taken from GUI.

  Output: Shortest path with the distance will be displayed.

- **find_all_paths(graph)**

  It will give all possible paths from source node to destination node.

  Input: graph

  Src (source node), dest (destination node) taken from GUI.

  Output: All paths with the distance will be displayed.

- **create_routing_table(routes)**

  It will display the routing table for all nodes in graph.

  Input: routes (routing table)

  Output: It will display the routing table

9 Testing Hardware Specifications

As mentioned in Section 1., the program executables were tested on three different platforms. The hardware specifications of the machines are outlined below.

Table 1: Hardware Specifications

	Linux Mint 18	**Linux Mint 17.3**	**Windows 7 Ultimate**
Architecture	64-bit	64-bit	64-bit
Processor	Intel i5-6600K	Intel i5-2450M	Intel i5-2410M
Clock Speed	4.4 GHz	2.5 GHz	2.3 GHz
Memory	16 GB DDR4	8 GB DDR3	3 GB DDR3
Memory Clock	3000 MHz	1333 MHz	133 MHz