COMP4322 Link State Routing Project

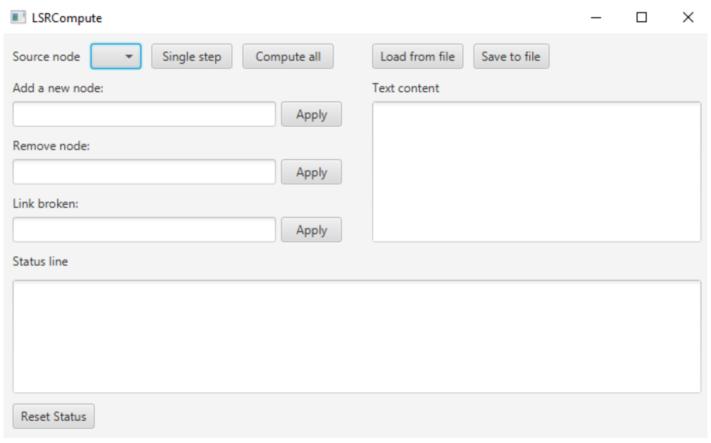
16076398d Chung Kwok Tung, 16088575d Yuen Ho Yin

Introduction

In this project, we developed a Java program that accepts .lsa input and simulate the process of a link state routing (LSR) algorithm, the Dijkstra's algorithm [1] in the provided network topology. The LSR algorithm is used for a router to broadcast information to other routers (assuming that the router has the knowledge of the global network). It is called "link state routing" because it traverses further based on the known states of the links (i.e. costs) from the frontier nodes (thus "route by link states"). To fulfill this idea, Dijkstra's algorithm does exactly that and thus is classically used as the LSR algorithm.

This program can be used in either command-line or GUI, and both can compute in either compute-all (CA) mode or single-step (SS) mode. In addition, we also support changing and saving the network topology of the provided .lsa in our GUI, such as to remove link, add nodes, and remove nodes.

In the following sections, we will first summarize our LSR algorithm corresponds to our Java code. Then we summarize our design ranging from our program architecture, to our unit tests. Finally, we wrap up our report with describing our task distribution.



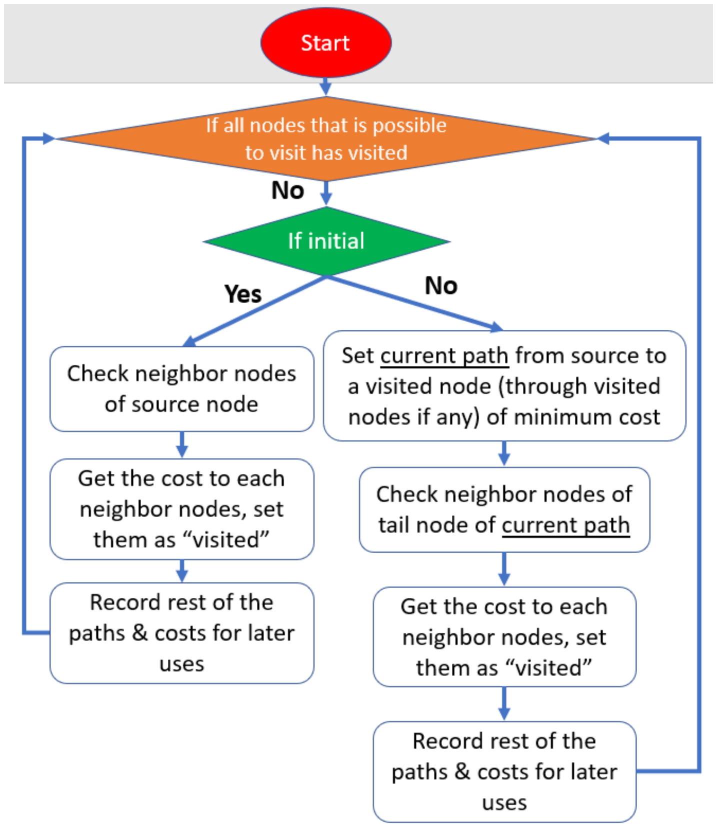
LSR Algorithm: Dijkstra's Algorithm

The Dijkstra's Algorithm (labelled as DA from now on) is classic graph traversal algorithm [1]. It is basically a graph traversal algorithm that traverses all nodes, of which the edges used to traverse all nodes induce minimum cost. To give another perspective, DA is same as the uniform cost search algorithm except without the goal node (i.e. the goal is to traverse all nodes).

Below demonstrates the pseudocode of our DA implementation.

```
for each node:
    dist[node] = MAX_INTEGER
    visit[node] = false
    path[node] = empty array

dist[source node] = 0
for loop 0 to number of nodes:

    for each node:
        // get the tail node of the current shortest distance path, or source node initially
        if not visit[node] and dist[node] < minDist:
            minDist = dist[node]
            minNode = node

    if (there is still node(s) that is possible to visit):
        visit[minNode] = true

        for all neighbor node of minNode:
            cost = edge cost to neighbor node
            if (dist[minNode]+cost < dist[neighbor node]):
                dist[neighbor node] = dist[minNode] + cost
                path[neighbor node]  = path[minNode] + [neighbor node]
            print(Found neighbor node. Path: path[neighbor node]. Cost: dist[neighbor node].)
```

To summarize the pseudocode, during the while loop, the code initially checks the cost to the neighbor nodes from the source node, mark the neighbors as visited with their cost and set the current path as the path of minimum cost (from source to neighbor). Here, for any path, we let the head node as source node, and let as tail node on the other end. Then, from onward, until all nodes are possible to visit from the source are visited, the code traverses from the path of minimum cost to check the neighbors of its tail node, one node away from the tail node at a time. This way the code can switch to other paths if it is more cost-effective, realizing the Dijkstra's Algorithm. Below shows the flow chart of our implemented DA.


Start
If all nodes that is possible to visit has visited
No
If initial
Yes
No
Check neighbor nodes of source node
Get the cost to each neighbor nodes, set them as "visited"
Record rest of the paths & costs for later uses
Set current path from source to a visited node (through visited nodes if any) of minimum cost
Check neighbor nodes of tail node of current path
Get the cost to each neighbor nodes, set them as "visited"
Record rest of the paths & costs for later uses

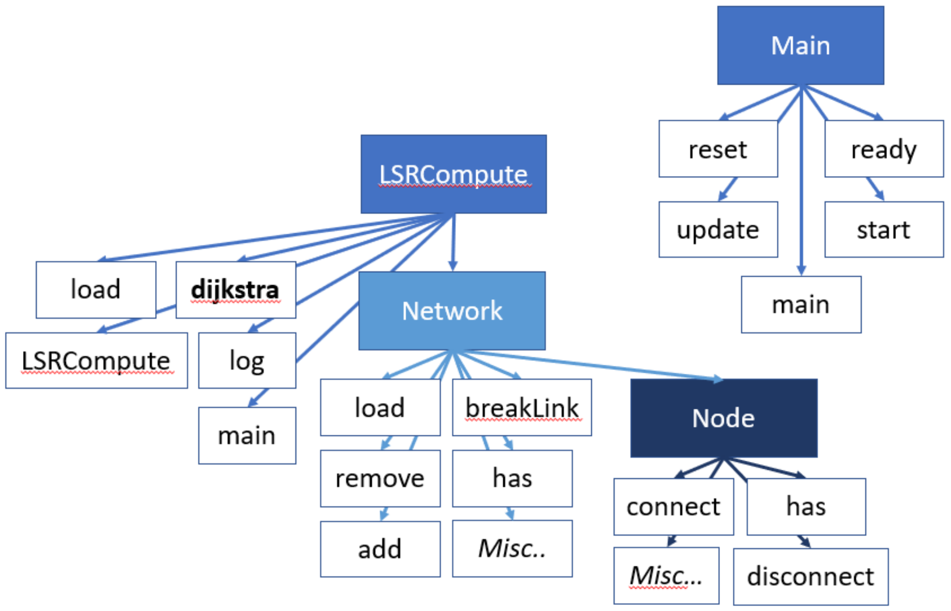
Please note that in our actual Java implementation, there are some variations such to simplify the control flow of our program while keeping them equivalent (e.g. output formatting).

Architecture Design

Our program consists of two files, *Main.java* and *LSRCompute.java*. The *Main.java* includes the Main class, and the *LSRCompute.java* includes the LSRCompute class with a subclass Network, and a subclass in class Network, Node. A brief summary of each role for each class is as follows:

- *Main*: All and only the GUI event of our program
- *LSRCompute*: DA and logging. subclass: *Network*
- *Network*: file parsing, general graph operations (check if node exists, remove links etc), subclass: *Node*
- *Node*: node-centric operations (link nodes, check node's neighbors)

Below shows a diagram the demonstrates the hierarchical architecture of our classes and methods. Blocks with white background are methods, and the rest are either parent class or child class. The "Misc" methods is a collection of helper methods which will be elaborated below. The main algorithm is tagged as bold: **dijkstra**. For starter, see the root class Main and LSRCompute.



Here we elaborate the methods in detail:

For the Main class, the *reset* method clear the status window for user to rerun or run another round of LSR simulation; the *update* method update the displayed text (about the file network topology) after modification or file loaded; the *ready* method binds all the GUI element with java variable; the *main* method simply runs the GUI on launch; the *start* method sets the event listener for all buttons.

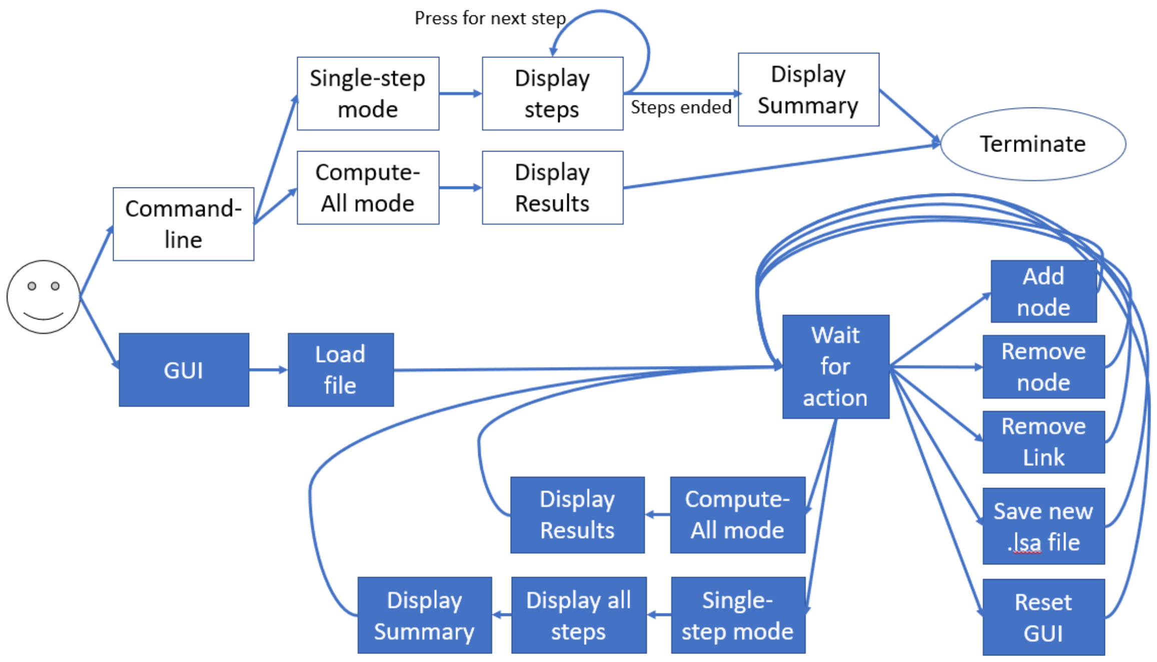
For the LSRCompute class, the *load* method is an entry method to the *load* method of Network class; the *dijkstra* method is the main algorithm of this project, as described in the previous section; the *LSRCompute* method is a helper method to create the Network class for command-line mode; the *log* method formats the text output of the program; the *main* method handles the entry of the application in command-line mode.

For the Network class, the *load* method parse the input file according to the .lsa format; the *breakLink* method is an entry method to remove edge between the given two nodes using the *disconnect* method in Node class; the *remove* method remove node from the network by disconnecting all nodes to the given node; the *add* method add new nodes to the network; the *has* method check if a node exists in the Network; the rest of the methods (*Misc…*) are mostly related to output formatting (e.g. *toString*), array manipulation (e.g. *indexOf*), or class constructor.

For the Node class, the *connect* method connect the given two node and the Network class; the *has* method check if a given node is a neighbor of another given node (unlike in Network class); the *disconnect* method is a sub-method of the *breakLink* method in Network class (as explained); and the rest of the methods (*Misc…*) are similar as in Network class.

Application Functionality

In this section, we give an overview of the capability of our program using a simple UML diagram (application flow) below. As explained in the Introduction section, apart from fulfilling the mandatory parts with support of both GUI and command-line (in both SS and CA mode), we also implement the optional features such as topology updating, file saving.



Above Communication UML Diagram shows all functionality at a glance. In the GUI, the user can freely modify the imported network topology in various ways (add & remove node & link), save the modified (or unmodified) network as .lsa, and computes the results in CA or SS mode while experimenting with different network setups.

Here, the mandatory features and optional features of our program are implemented as instructed in the project requirement. For the file saving feature, we add that in extra for the convenience of the user, such to create multiple topological profile for a network experimenting LSR.

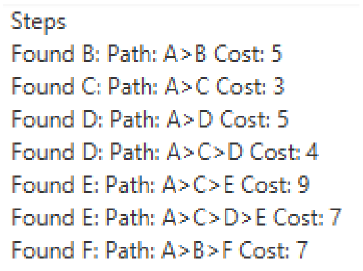
Test Cases

In this section, we present test cases that minimum test each feature of our program, as well as to test the correctness of our DA.

To test the correctness of our DA, we first present a minimum case of a simple path. We then present two cases where there are two simple paths to a goal node, of which for one case a simple path is definitively lower cost, another has our DA to switch path amidst the algorithm due to lower cost. Finally, we wrap up with a classic case, Figure 1 in the project requirement document.

To test each feature, we present two cases for “remove node” and “remove link” based on this simple path, of which such action would make DA’s ‘flooding’ impossible. Another two cases where ‘flooding’ is still possible. Then, we present two cases for “add node” and “add link” where one of the cases will result in traversing to this node first due to lower-cost, another case would be to not traverse this node first due to higher-cost.

In all the test cases, we display the status of the LSR from our source node using SS (single step) mode. That is, we step-by-step show the LSR as follows:



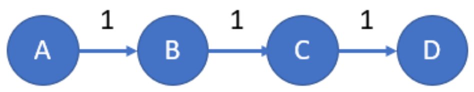
```
Steps
Found B: Path: A>B Cost: 5
Found C: Path: A>C Cost: 3
Found D: Path: A>D Cost: 5
Found D: Path: A>C>D Cost: 4
Found E: Path: A>C>E Cost: 9
Found E: Path: A>C>D>E Cost: 7
Found F: Path: A>B>F Cost: 7
```

In above example, the source node is A, and it first found B, C, and D in the first three steps. In the forth step, the program found A>C>D to be the current lowest-cost path. The program keeps running until all nodes are traversed with lowest cost.

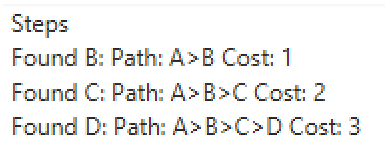
Below shows the topology for each test case. Please note that for all test cases, the source node is A, and the directional edges indicate the expected flooding direction (the actual network is still undirectional).

Test Cases: Correctness of the Dijkstra’s Algorithm (DA)

Simple path

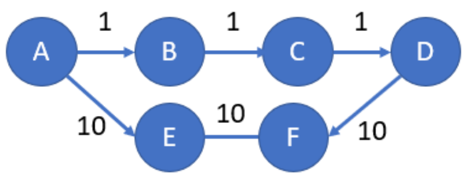


This test case demonstrates the basic scenario applying the DA. As shown below using the defined status text, our program successfully performs the LSR in the expected direction: A>B>C>D. This is successful because each step of DA explores unvisited node with the lowest edge cost.

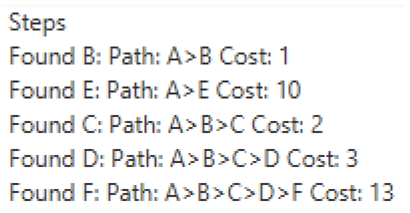


```
Steps
Found B: Path: A>B Cost: 1
Found C: Path: A>B>C Cost: 2
Found D: Path: A>B>C>D Cost: 3
```

Simple paths with no switching

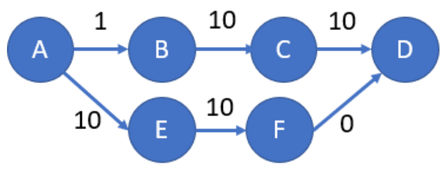


This test case demonstrates a scenario where the LSR first traverses one path (A>B>C>D) since A checked the cost of A>B is lower than A>E. Then the program finally traverses to F using the path of lowest cost currently (A>B>C>D>F). Once again, our program is successful as shown below. It is successful in the sense that it has no path switching as stated.

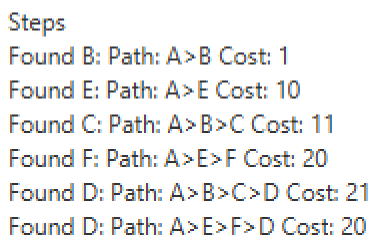


```
Steps
Found B: Path: A>B Cost: 1
Found E: Path: A>E Cost: 10
Found C: Path: A>B>C Cost: 2
Found D: Path: A>B>C>D Cost: 3
Found F: Path: A>B>C>D>F Cost: 13
```

Simple paths with switching



This test case demonstrates a scenario where the DA switches to other routes for flooding due to the other path being lower cost. That is, initially it would be A>B>C (cost = 11), since A>E is lower (cost = 10), it goes to A>E>F. Now, A>E>F has higher cost (cost = 20), so it switches to A>B>C>D. It goes on until all path has traversed. Once again, our program is successful as shown below. We can see that in the last two steps, the two paths derives as expected in our figure: A>B>C>D and A>E>F>D.



```
Steps
Found B: Path: A>B Cost: 1
Found E: Path: A>E Cost: 10
Found C: Path: A>B>C Cost: 11
Found F: Path: A>E>F Cost: 20
Found D: Path: A>B>C>D Cost: 21
Found D: Path: A>E>F>D Cost: 20
```

Classic case (Figure 1 in Project Requirement Document)

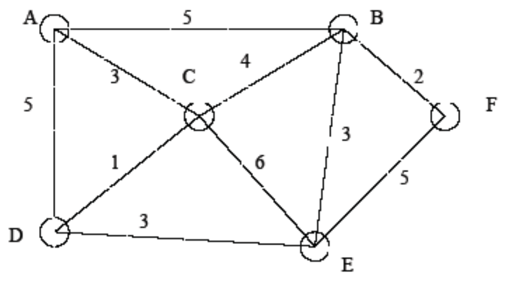
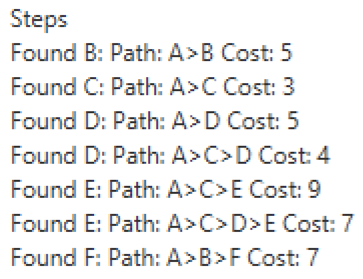


Figure 1. An example of network topology

Since the program is able to do the basic traverse (first test case), follow paths of lowest cost (second test case), and switch paths if necessary due to cost (third test case), we then show our program running a classic example that simulate a practical network. As expected, our program works, since any network is just a combination of the above three test cases (i.e. cost-driven traversal).

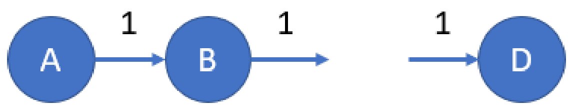


```
Steps
Found B: Path: A>B Cost: 5
Found C: Path: A>C Cost: 3
Found D: Path: A>D Cost: 5
Found D: Path: A>C>D Cost: 4
Found E: Path: A>C>E Cost: 9
Found E: Path: A>C>D>E Cost: 7
Found F: Path: A>B>F Cost: 7
```

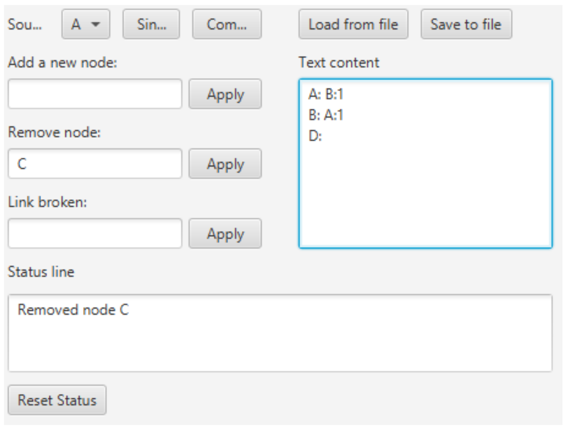
Test Cases: Correctness of each feature

Here, each test case is based on the basic simple path test case in the above section. If there is any difference to the basic simple path, we modify the test case using our program correspondingly.

<u>Remove Node: flooding is impossible</u>



(Please note that the edge in this case is also removed, however it is left in this graph above just to illustrate the situation.) Here we demonstrate a scenario where the node removal causes a network partition. As shown below, we remove node C from the simple path test case.

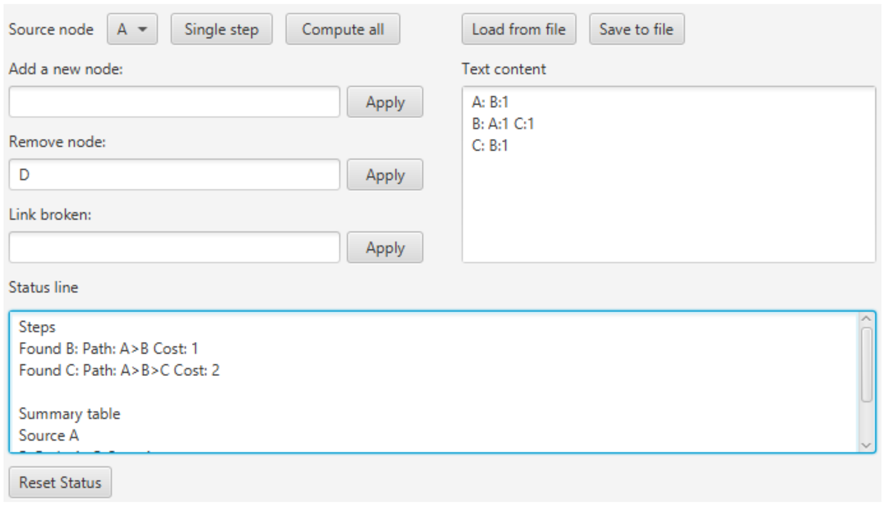


Remove node: C

```
A: B:1
B: A:1
D:
```

```
Removed node C
```

Then we run our test case. By running this test case, we expect that D is simply unreachable since A is the source node. The program terminates recognizing that D is not part of its sub-network. In other words, it cannot flood to D and the program knows that. As expected, the program terminates successfully (A>B). This is due to our program implement a for loop instead of a while loop check if all nodes are visited (i.e. for loop upper-bounded the number of iterations of the DA).

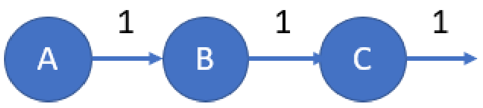


Remove node: D

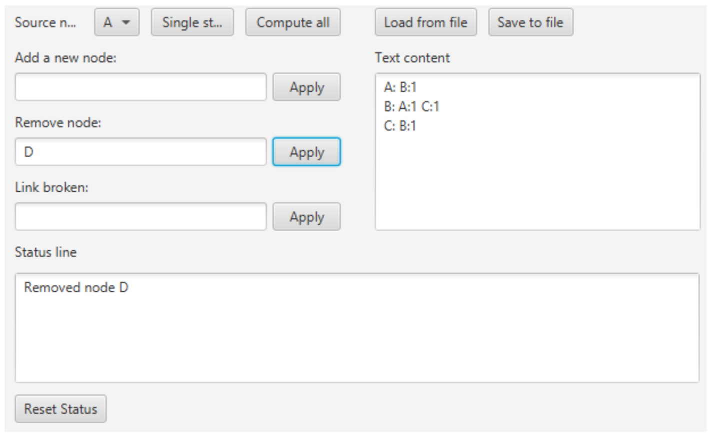
```
A: B:1
B: A:1 C:1
C: B:1
```

```
Steps
Found B: Path: A>B Cost: 1
Found C: Path: A>B>C Cost: 2

Summary table
Source A
```

Remove Node: flooding is possible



As shown above, we first remove node D and show the output of our program as below.

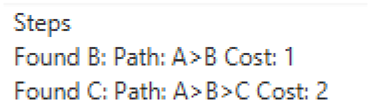


D

```
A: B:1
B: A:1 C:1
C: B:1
```

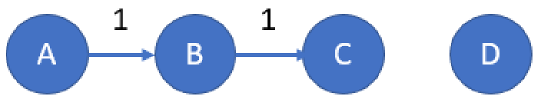
```
Removed node D
```

Then, we run the DA. We expect that the program can flood the whole network as there is no network partition (a path of length 3). As expected, the program performs successfully traverse the whole network in a simple path: A>B>C, as shown below.

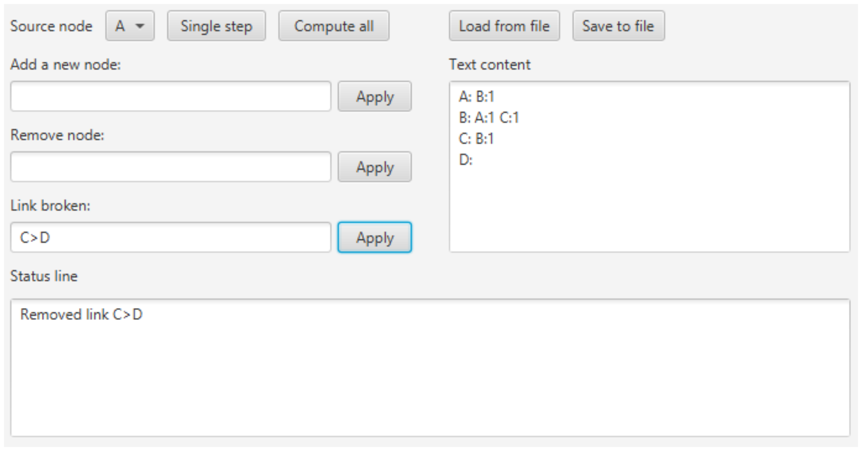


```
Steps
Found B: Path: A>B Cost: 1
Found C: Path: A>B>C Cost: 2
```

<u>Remove Link: flooding is impossible</u>



Here we demonstrate a case where a link removal results in network partition. We first show the output of our application by deleting such link (C > D) as below.

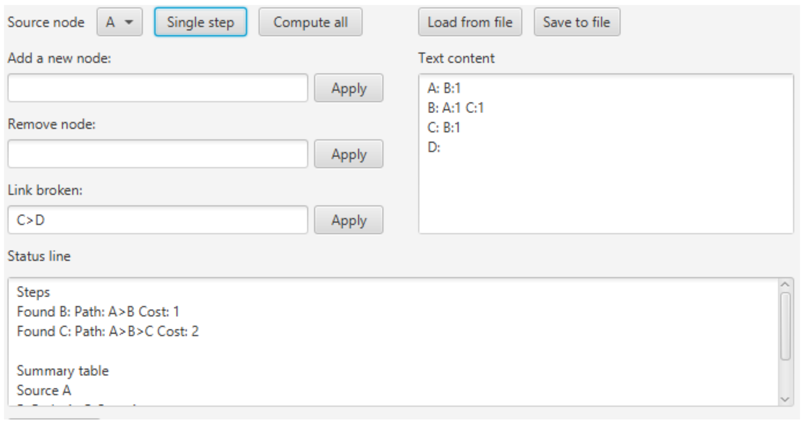


C>D

```
A: B:1
B: A:1 C:1
C: B:1
D:
```

```
Removed link C>D
```

Then, we run the DA in single step. Similar to the first test case in this sub-section (removing node: flooding is impossible), it is expected that the program acknowledges that D is impossible to reach, and terminates accordingly. As expected, the program performs successfully as shown below.

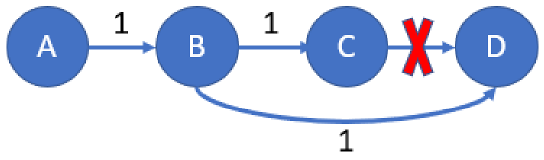


```
C>D
```

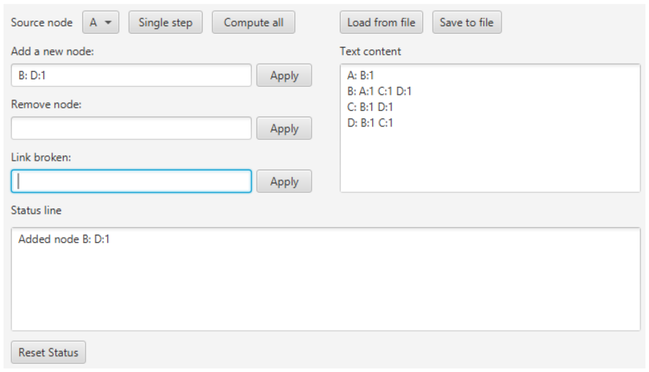
```
A: B:1
B: A:1 C:1
C: B:1
D:
```

```
Steps
Found B: Path: A>B Cost: 1
Found C: Path: A>B>C Cost: 2

Summary table
Source A
```

Remove Link: flooding is possible



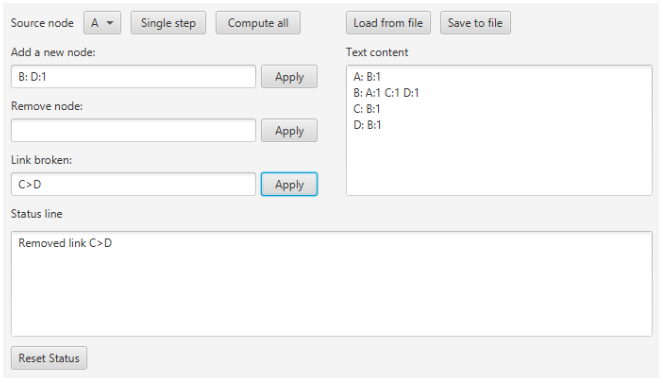
Here we demonstrate a case where flooding is possible even after removal of C>D. To achieve this, we add an additional link B>D (by add a node as follows: "B: D:1") on top of the simple path test case, resulting the image above. Here, we demonstrate that we add B>D and remove C>D using our program as below.



```
B: D:1
```

```
A: B:1
B: A:1 C:1 D:1
C: B:1 D:1
D: B:1 C:1
```

```
Added node B: D:1
```



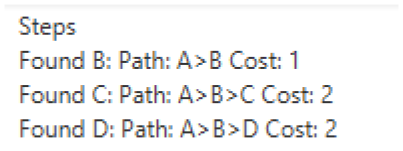
```
B: D:1
```

```
C>D
```

```
A: B:1
B: A:1 C:1 D:1
C: B:1
D: B:1
```

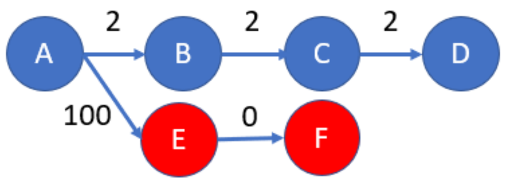
```
Removed link C>D
```

Then, we run DA in single step. It is expected that the program simply floods the network by A>B>C, then B>D. As expected, our program performs successfully flooding to D from B as below.



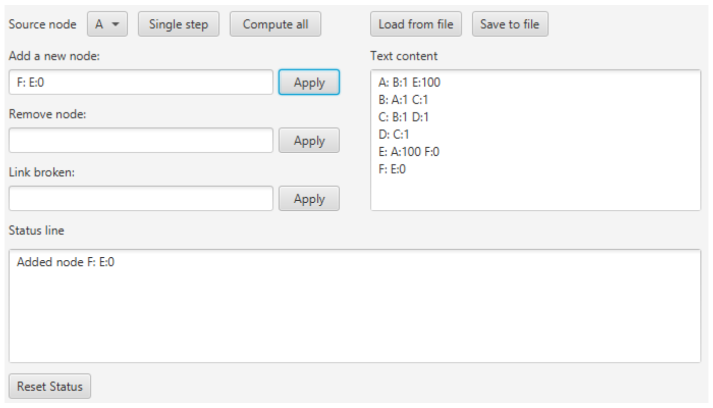
```
Steps
Found B: Path: A>B Cost: 1
Found C: Path: A>B>C Cost: 2
Found D: Path: A>B>D Cost: 2
```

<u>Add Node: no change in traversing priority</u>



In the “Add Node” test case, we would like to test if adding a new node would change the traversal behavior. To do this, we introduce two nodes that has higher edge costs and see if the DA works as expected (we already know that the DA works by testing the correctness of DA). We need to add new nodes because E would have been “visited” at the first iteration (according to DA). An additional node would be needed to see if traversing to the new F actually is delayed (i.e. E>F is the last step), making the new node does not change the previous traversing priority.

To add a new node, we first perform as follows in our program.

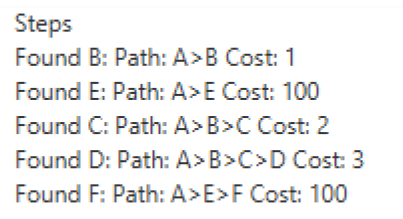


Add a new node:
```
F: E:0
```

Text content
```
A: B:1 E:100
B: A:1 C:1
C: B:1 D:1
D: C:1
E: A:100 F:0
F: E:0
```

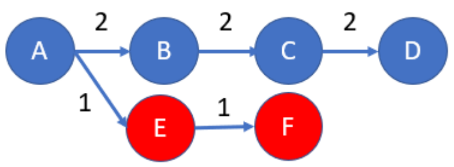
Status line
```
Added node F: E:0
```

Then, we run DA in SS mode. As expected, our program performs successfully by traversing E>F the last step as shown below.



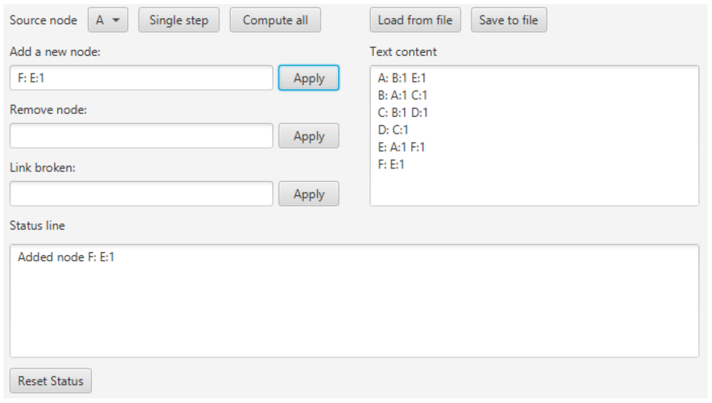
```
Steps
Found B: Path: A>B Cost: 1
Found E: Path: A>E Cost: 100
Found C: Path: A>B>C Cost: 2
Found D: Path: A>B>C>D Cost: 3
Found F: Path: A>E>F Cost: 100
```

<u>Add Node: changes in traversing priority</u>



Similar to the above test case, we add node for the simple path test case. However, this time we specify a lower edge cost such that traversing A>E>F would become the priority instead of the original A>B>C>D. Again, two nodes are needed since E would be "visited" at the first iteration, so we can only see a change in traversing priority with the addition of F.

Firstly, we add the two new nodes as follows:

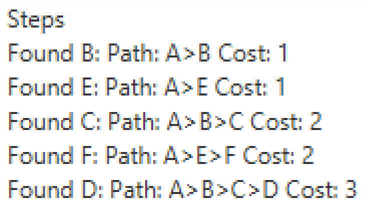


```
F: E:1
```

```
A: B:1 E:1
B: A:1 C:1
C: B:1 D:1
D: C:1
E: A:1 F:1
F: E:1
```

```
Added node F: E:1
```

Then, we run DA in SS mode. As expected, our program switch the traversing priority in traversing A>E>F first (due to the lower cost) that would otherwise be A>B>C>D if E>F does not exists. That exists, the E and F addition works such to affect the DA.



```
Steps
Found B: Path: A>B Cost: 1
Found E: Path: A>E Cost: 1
Found C: Path: A>B>C Cost: 2
Found F: Path: A>E>F Cost: 2
Found D: Path: A>B>C>D Cost: 3
```

Add Link

In this program, add link is the same as "Add Node", that is, to specific the additional linkage between two nodes via "Add a new node" as shown in the test case "Remove Link: flooding is possible". If there is a link exists between the two nodes already, this "additional linkage" replaces the current link via its link cost.

Since "Remove Link: flooding is possible" is working as expected (for adding links in existing nodes), and "Add Node" test cases are working as expected (for adding links of new nodes), both sufficiently represent the "Add Link" test cases, deeming "Add Link" being successfully.

Conclusion

In this project, we implemented the LSR using the Dijkstra's Algorithm. We fulfill all the mandatory features such as SS and CA mode, and optional features such as remove links, save file, etc. We demonstrated that our DA is implemented elegantly by pseudocode, verbally, and by flow diagram. We then show our architecture design using a hierarchical diagram, and our application design with a communication UML diagram. Finally, we demonstrate a comprehensive coverage of test cases, each with the topology of the input case drawn and discussed.

Contribution

The contribution of the team is as follows:

Group No: 14		
Members and contributions:		
Name	**Student ID**	**Contributions**
Chung Kwok Tung	16076398d	The whole java codebase (GUI & command-line)
Yuen Ho Yin	16088575d	Test cases, report and presentation slides

Reference

[1] Dijkstra, Edsger W. "A note on two problems in connexion with graphs." *Numerische mathematik* 1.1 (1959): 269-271.