Kealia Perrine

CPE 400 Final Project

Dynamic routing mechanism design in faulty network

Due: December 7, 2020

Functionality:

My Idea for Routing:

The idea of my project is to take a pseudo-machine learning approach. The weights of edges within the graph update based upon how reliable the nodes are that they connect. If a node is unreliable, the weight of the edge is increased. This is so when Dijkstra's algorithm is run *before nodes fail*, unreliable nodes are less likely to be part of the path. To determine unreliable nodes, the average of the failure probabilities is found. If the probability for a node is greater than that average, the node is deemed unstable and has its edge weights increased. Thus, the path generated will more than likely be a valid path.

The reason this is only pseudo-machine learning is because instead of maintaining the probability of failure for nodes and the weights of edges, the user enters the probabilities and a new random graph is generated each time the program is run. To implement a more machine learning approach, the probabilities, weights, and graph/network would persist, and the probabilities and weights would update every iteration based on whether that node failed or not.

In this project, the path is then compared to the list of the failed nodes to see if the path is valid. If no nodes within the path failed, the Dijkstra path is selected, since it was the shortest even before there were less nodes available. However, if Dijkstra's path does happen to contain a failed node, Bellman ford search algorithm AKA Distance vector algorithm is run. This is a decentralized algorithm, so it may have higher success with navigating the faulty network than Dijkstra's did.

If both these searches fail, then there is no possible path between the source and destination node.

By using updated weights and Dijkstra's together, I am hoping to clear out more unreliable nodes ahead of time, so that less effort is needed in trying to adjust paths quickly due to node failure.

Why it's novel:

A faulty network is unstable and thus network managers would typically employ a more decentralized approach to finding paths between nodes. However, I am able to use Dijkstra's algorithm because of the modifications made to the graph. By updating weights to discourage from visiting from unreliable nodes, we can use a more global algorithm like Dijkstra's since we are preemptively avoiding failed nodes. If Dijkstra's ends up not working for us, we can then run a decentralized algorithm, Bellman-Ford.

**** See code set up / function explanations in the README**

How to use it:

See README for more detailed instructions, but here are simple instructions:

- Ensure Python3 is installed. I used Python 3.9.0.
- Pip install networkx, matplotlib, and statistic.
- Numpy may also need to be installed if running on Windows (I used Linux).
- Run the program: python3 network.py
- Follow instructions within program.
- **Ensure that you close the pop up with the printed graph when done observing it. The program cannot continue until it is closed.**

Limitations/Areas of Improvement:

This idea for dealing with faulty networks only works well on smaller networks since the program uses Dijkstra's algorithm which touches all nodes in a network. This is unreasonable to expect with huge networks, such as considering the Internet as a whole. To combat this, a later edition of this simulation could mix Dijkstra's algorithm with a hop limit (how many nodes are visited). Thus, a large network could be sub sectioned into more manageable chunks for Dijkstra, or only run this idea/algorithm on smaller subnets.

Another limitation is that although through my own testing, the Dijkstra path was often accepted, having to run another algorithm, Bellman-ford, as well, can cause runtimes to increase with a large network. The same solution as before can be employed, using a hop limit for Dijkstra as well as Bellman, to cap the max run time. There could also be a couple accepted paths that are saved and only updated so often after a time limit, so the algorithms do not have to be run on each individual packet.

<u>Results:</u>

Analysis of results:

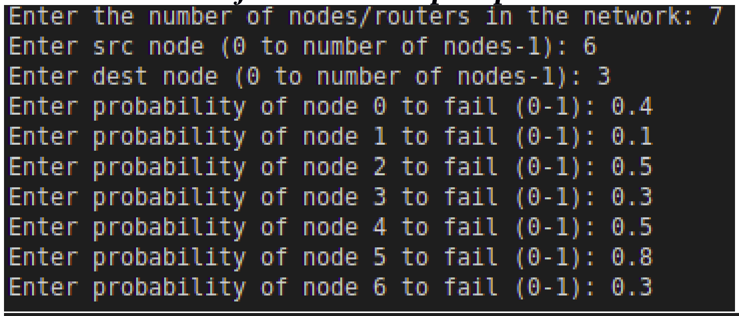
When testing my project and running it in order to obtain screenshots, most of the time the first path using Dijkstra's path was accepted. It took me about 7 tries to force the program to use the Bellman-Ford. While this may only be my experience with how the random values ended, I would conclude that updating the weights and then using Dijkstra's algorithm was a success. By updating the weights to discourage using faulty nodes, Dijkstra's algorithm was able to make reliable and short paths through the network.

Every so often there is the case where there is no path available, but that is not the fault of the algorithms or my idea for dealing with faulty networks. There will be sometimes where too many routers fail for a packet to be delivered, and there is always a bit of expected packet loss within a network. Thus, I am content with the overall capability of my program to find a path through the faulty network.

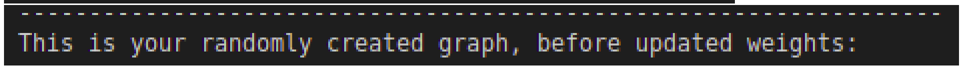
This idea for traversing a faulty network is probably best when prioritizing that a packet is definitely received, such as downloading a file, rather than speed, such as streaming. This is because updating the weights changes the network and may cause potentially shorter paths to be missed since unreliable nodes are skipped. For example, an unreliable node may have a shorter path initially, but it is ignored after the weights are updated since it cannot be depended on delivering the packet. It may not end up failing, but since the path already skipped it, it does not get a chance to deliver along that shorter route.

Screenshots/Example output:

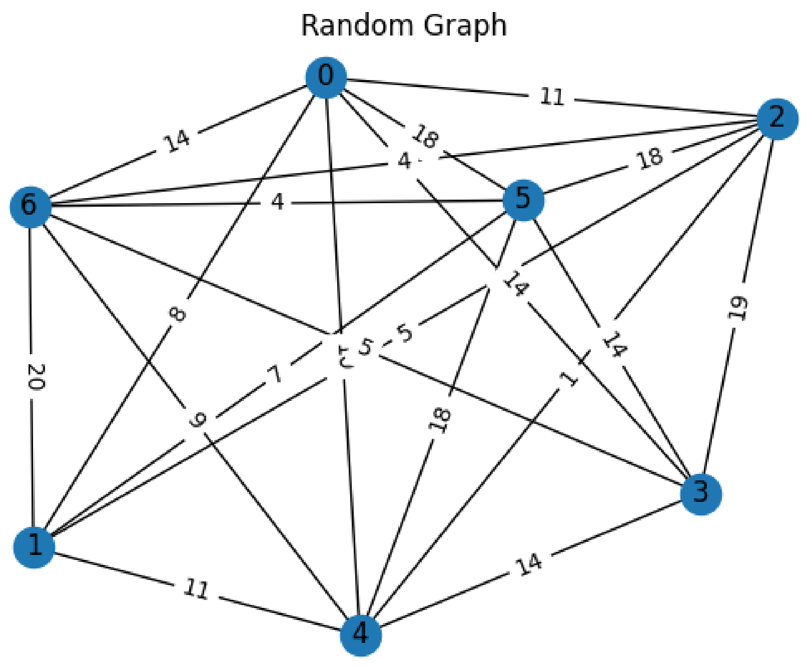
- ***Case 1: Dijkstra's is accepted path:***

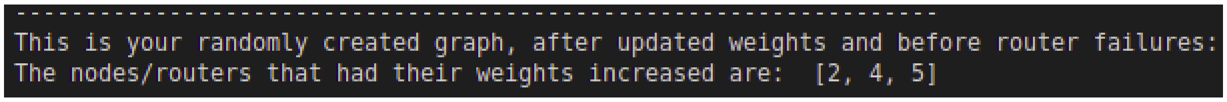


```
Enter the number of nodes/routers in the network: 7
Enter src node (0 to number of nodes-1): 6
Enter dest node (0 to number of nodes-1): 3
Enter probability of node 0 to fail (0-1): 0.4
Enter probability of node 1 to fail (0-1): 0.1
Enter probability of node 2 to fail (0-1): 0.5
Enter probability of node 3 to fail (0-1): 0.3
Enter probability of node 4 to fail (0-1): 0.5
Enter probability of node 5 to fail (0-1): 0.8
Enter probability of node 6 to fail (0-1): 0.3
```

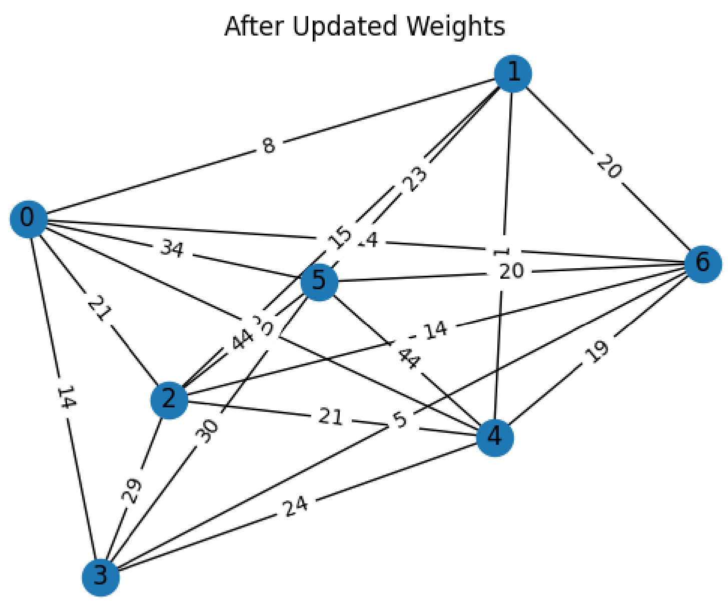


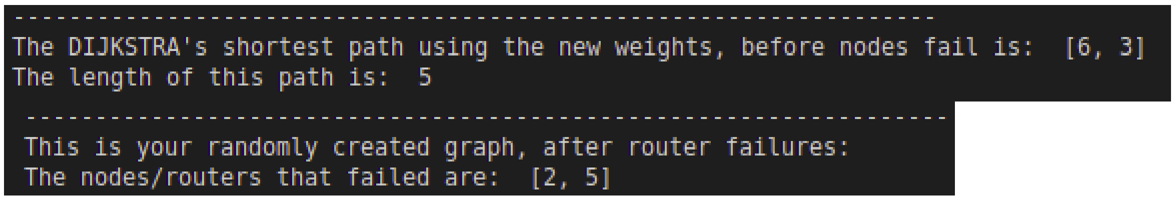
```
-------------------------------------------------------------------
This is your randomly created graph, before updated weights:
```

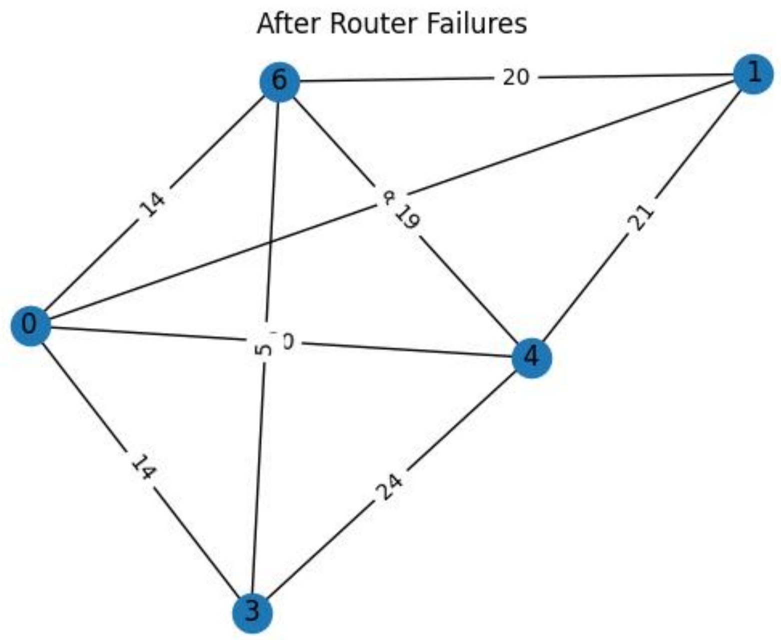


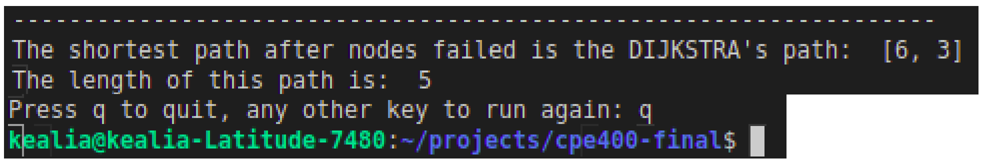


```
-------------------------------------------------------------------
This is your randomly created graph, after updated weights and before router failures:
The nodes/routers that had their weights increased are:  [2, 4, 5]
```

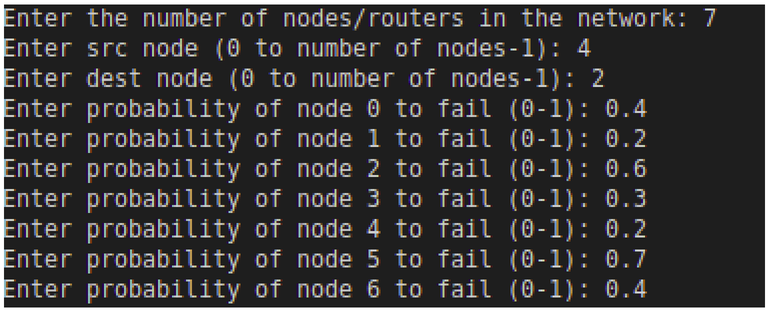



```
-----------------------------------------------------------------
The DIJKSTRA's shortest path using the new weights, before nodes fail is:  [6, 3]
The length of this path is:  5
 -----------------------------------------------------------------
 This is your randomly created graph, after router failures:
 The nodes/routers that failed are:  [2, 5]
```

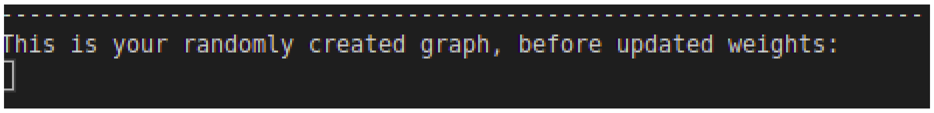



```
-----------------------------------------------------------------
The shortest path after nodes failed is the DIJKSTRA's path:  [6, 3]
The length of this path is:  5
Press q to quit, any other key to run again: q
kealia@kealia-Latitude-7480:~/projects/cpe400-final$
```
The Dijkstra's path was accepted because it was short and successfully avoided any nodes that failed.

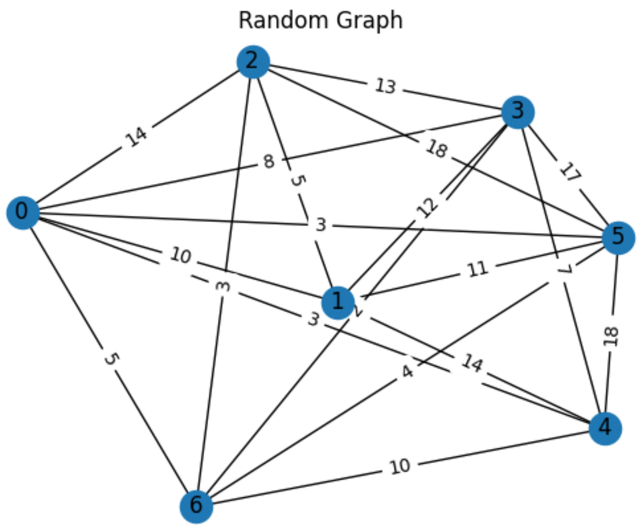
❖ ***Case 2: Dijkstra’s included failed node; Bellman-Ford is accepted path:***

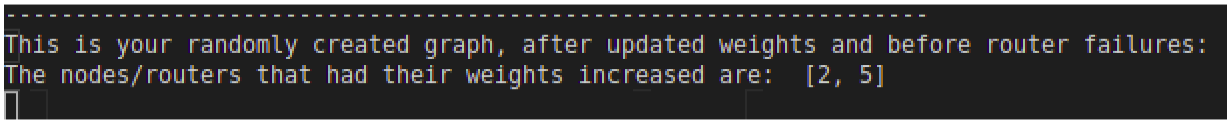


```
Enter the number of nodes/routers in the network: 7
Enter src node (0 to number of nodes-1): 4
Enter dest node (0 to number of nodes-1): 2
Enter probability of node 0 to fail (0-1): 0.4
Enter probability of node 1 to fail (0-1): 0.2
Enter probability of node 2 to fail (0-1): 0.6
Enter probability of node 3 to fail (0-1): 0.3
Enter probability of node 4 to fail (0-1): 0.2
Enter probability of node 5 to fail (0-1): 0.7
Enter probability of node 6 to fail (0-1): 0.4
```

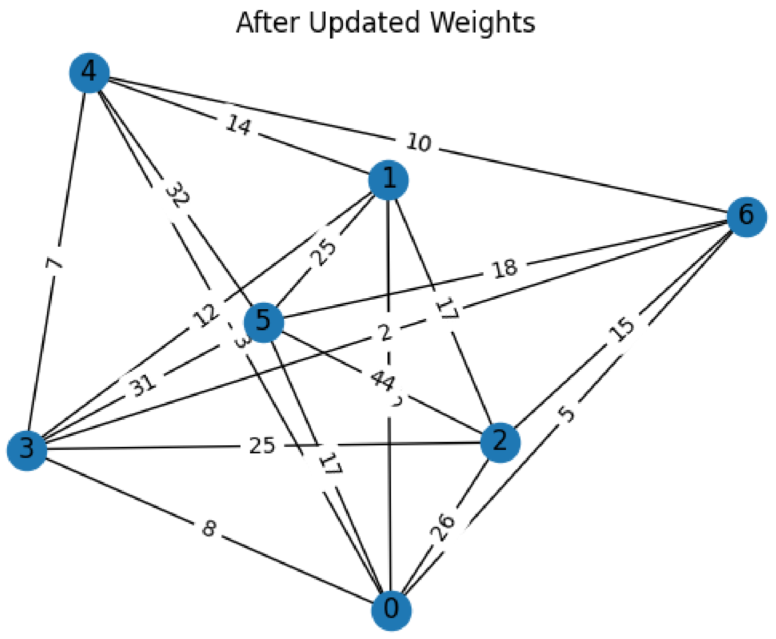


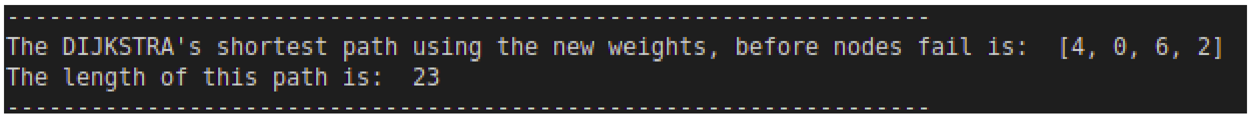
```
--------------------------------------------------------------
This is your randomly created graph, before updated weights:
```



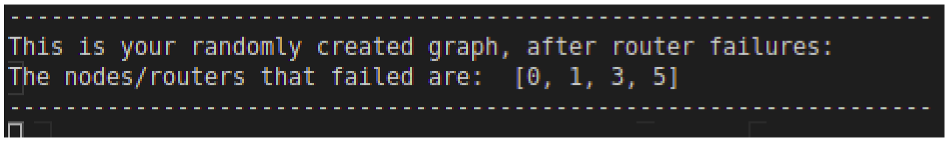


```
--------------------------------------------------------------
This is your randomly created graph, after updated weights and before router failures:
The nodes/routers that had their weights increased are:  [2, 5]
```

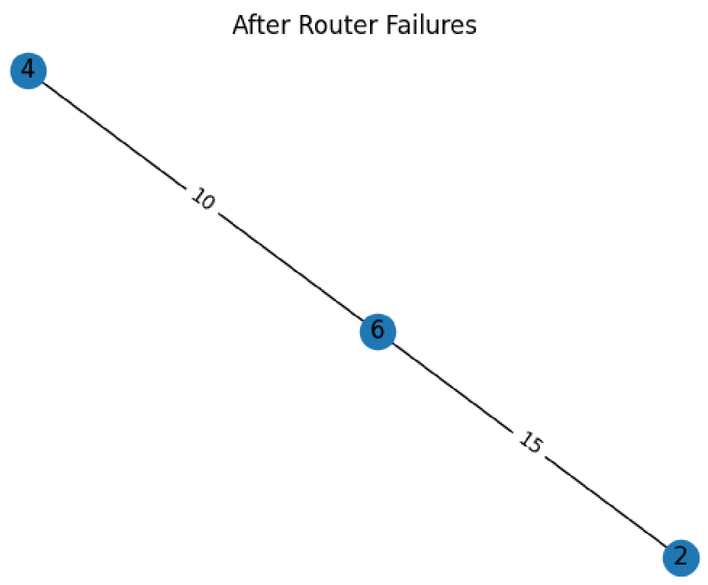


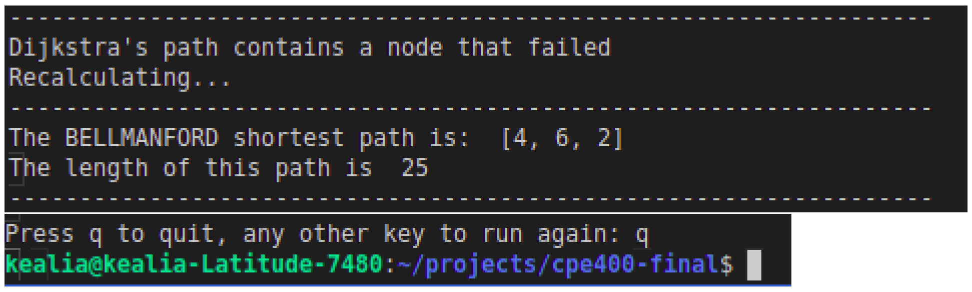


```
------------------------------------------------------------------
The DIJKSTRA's shortest path using the new weights, before nodes fail is:  [4, 0, 6, 2]
The length of this path is:  23
------------------------------------------------------------------
```



```
------------------------------------------------------------------
This is your randomly created graph, after router failures:
The nodes/routers that failed are:  [0, 1, 3, 5]
------------------------------------------------------------------
```

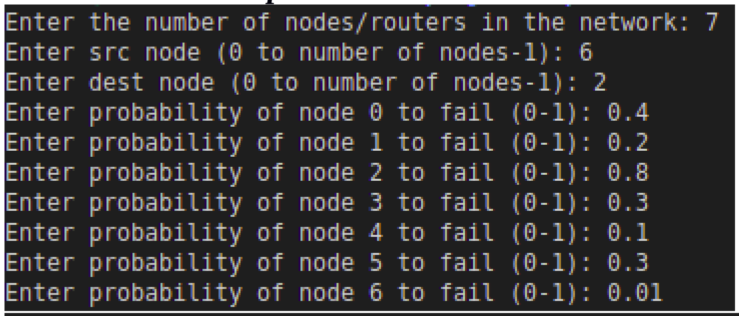




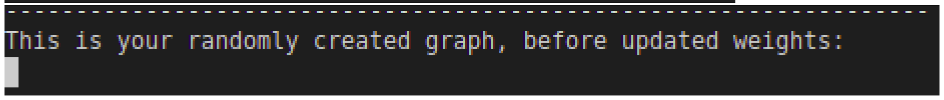
```
------------------------------------------------------------------
Dijkstra's path contains a node that failed
Recalculating...
------------------------------------------------------------------
The BELLMANFORD shortest path is:  [4, 6, 2]
The length of this path is  25
------------------------------------------------------------------
Press q to quit, any other key to run again: q
kealia@kealia-Latitude-7480:~/projects/cpe400-final$
```

The reason Dijkstra's path is rejected in this example is because node 0 failed when it's probability to fail (0.4) wasn't high enough for my program to suggest skipping it and/or it failed when we expected it wouldn't. Thus, Bellman-Ford's algorithm is run, and finds a path within the network to connect the source and destination.

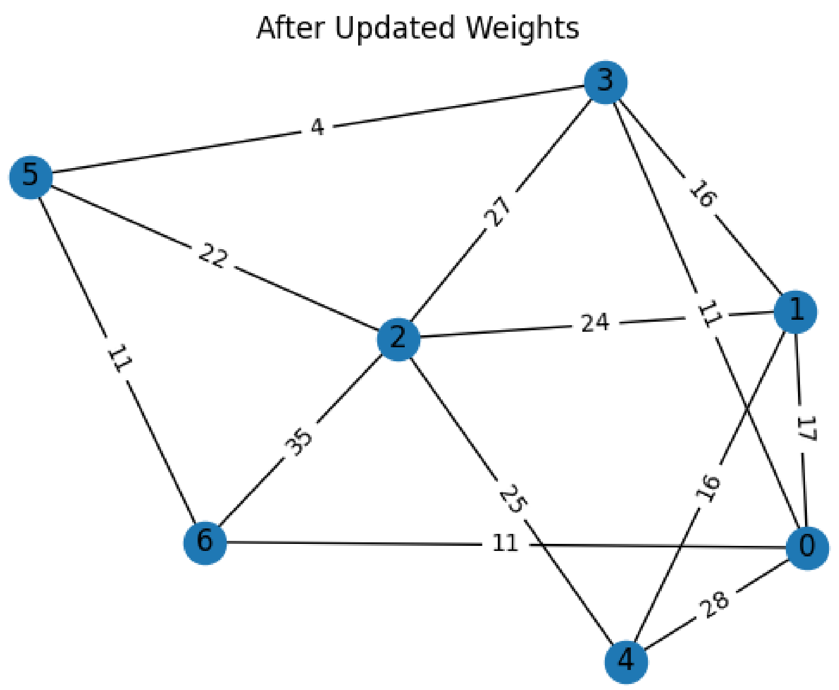
- ***Case 3 – No path exists***

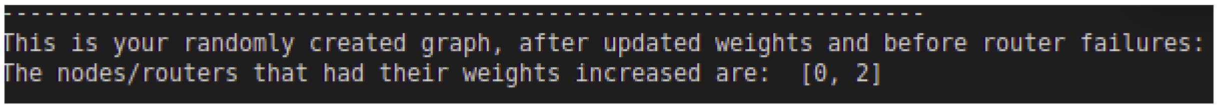


```
Enter the number of nodes/routers in the network: 7
Enter src node (0 to number of nodes-1): 6
Enter dest node (0 to number of nodes-1): 2
Enter probability of node 0 to fail (0-1): 0.4
Enter probability of node 1 to fail (0-1): 0.2
Enter probability of node 2 to fail (0-1): 0.8
Enter probability of node 3 to fail (0-1): 0.3
Enter probability of node 4 to fail (0-1): 0.1
Enter probability of node 5 to fail (0-1): 0.3
Enter probability of node 6 to fail (0-1): 0.01
```

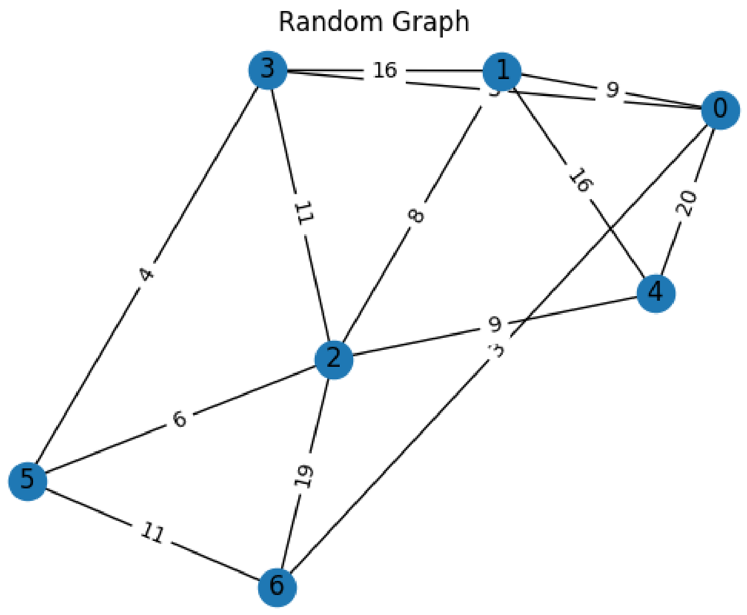


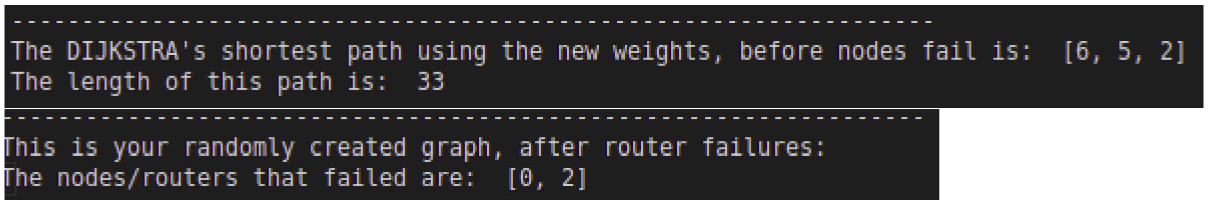
```
-------------------------------------------------------------------
This is your randomly created graph, before updated weights:
```

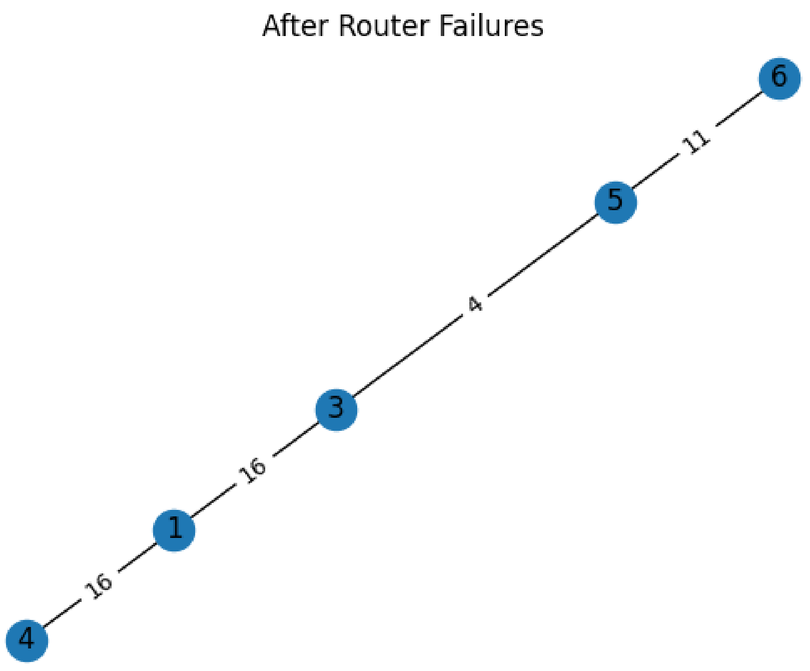


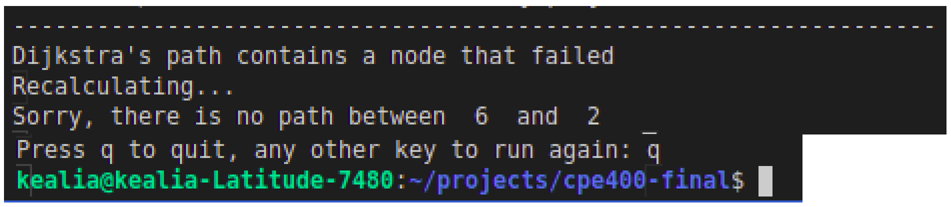


```
-------------------------------------------------------------------
This is your randomly created graph, after updated weights and before router failures:
The nodes/routers that had their weights increased are:  [0, 2]
```




```
------------------------------------------------------------------
The DIJKSTRA's shortest path using the new weights, before nodes fail is:  [6, 5, 2]
The length of this path is:  33
------------------------------------------------------------------
This is your randomly created graph, after router failures:
The nodes/routers that failed are:  [0, 2]
```

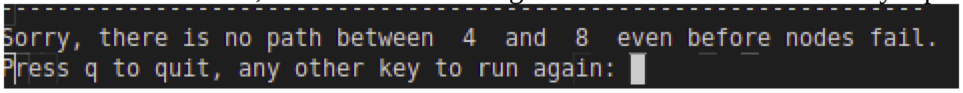




```
-----------------------------------------------------------------
Dijkstra's path contains a node that failed
Recalculating...
Sorry, there is no path between  6  and  2
 Press q to quit, any other key to run again: q
 kealia@kealia-Latitude-7480:~/projects/cpe400-final$ 
```

There is no path generate in this example because the destination node itself failed. Thus, it is impossible to create a path to a failed node. No fault to the algorithms.

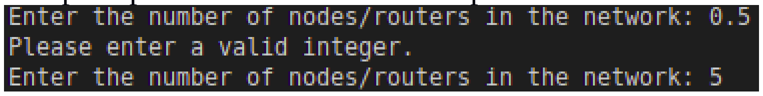
This case can also occur if the random graph created has an island node that happens to be the source or destination. However, this is more of an edge case since I could not reliably reproduce it.



```
-----------------------------------------------------------------
Sorry, there is no path between  4  and  8  even before nodes fail.
Press q to quit, any other key to run again: 
```

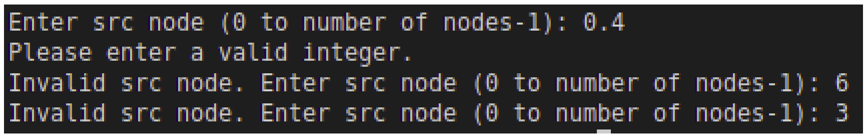
Errors/Exceptions:

- Only valid integers are allowed for the number of nodes (a fraction of a node does not exist). The prompt will re-ask until valid input is submitted.

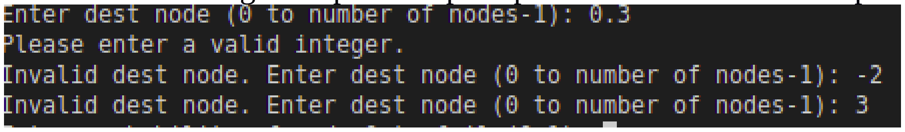


```
Enter the number of nodes/routers in the network: 0.5
Please enter a valid integer.
Enter the number of nodes/routers in the network: 5
```

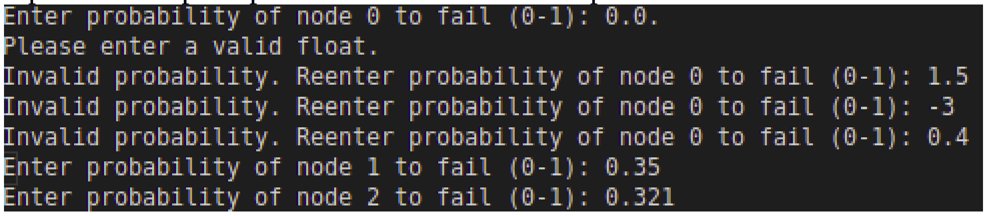
- The source node number must be an integer, and it must be 0 to number of nodes – 1. This ensures the node exists within the network. The prompt will re-ask until valid input is submitted.


```
Enter src node (0 to number of nodes-1): 0.4
Please enter a valid integer.
Invalid src node. Enter src node (0 to number of nodes-1): 6
Invalid src node. Enter src node (0 to number of nodes-1): 3
```

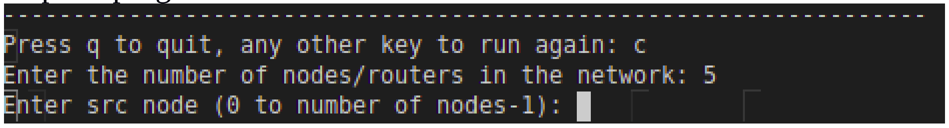
- ❖ The destination node number must be an integer, and it must be 0 to number of nodes – 1. This ensures the node exists within the network. It can be the same as the source node (although this is not an interesting example. The prompt will re-ask until valid input is submitted.

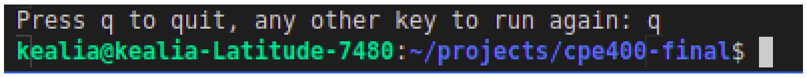

```
Enter dest node (0 to number of nodes-1): 0.3
Please enter a valid integer.
Invalid dest node. Enter dest node (0 to number of nodes-1): -2
Invalid dest node. Enter dest node (0 to number of nodes-1): 3
```

- ❖ The probability of a node failing must be between 0 and 1, *non-inclusive*. It must be a valid float number. The number of decimal points does not matter, but I recommend only going up to 3 places. The prompt will re-ask until valid input is submitted.

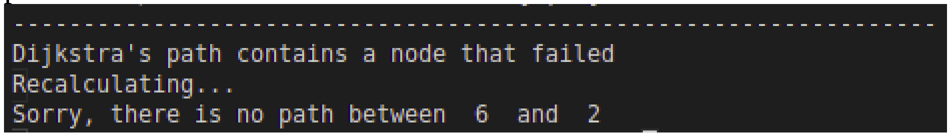

```
Enter probability of node 0 to fail (0-1): 0.0.
Please enter a valid float.
Invalid probability. Reenter probability of node 0 to fail (0-1): 1.5
Invalid probability. Reenter probability of node 0 to fail (0-1): -3
Invalid probability. Reenter probability of node 0 to fail (0-1): 0.4
Enter probability of node 1 to fail (0-1): 0.35
Enter probability of node 2 to fail (0-1): 0.321
```

- ❖ To quit, enter any character other than q, no errors on what character is entered. Entering q will stop the program.

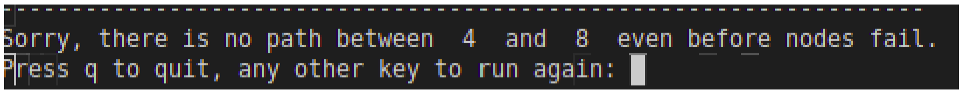

```
-----------------------------------------------------------------
Press q to quit, any other key to run again: c
Enter the number of nodes/routers in the network: 5
Enter src node (0 to number of nodes-1):
```


```
Press q to quit, any other key to run again: q
kealia@kealia-Latitude-7480:~/projects/cpe400-final$
```

- ❖ If the path doesn't exist, rather than an error on the algorithm, it is caught and prints that no path exists between the source and destination.


```
-----------------------------------------------------------------
Dijkstra's path contains a node that failed
Recalculating...
Sorry, there is no path between  6  and  2
```

This can also occur when the Dijkstra's path is first calculated and the source or destination node in the random graph is an island node. However, this is more of an edge case since I could not reliably reproduce this error. Nevertheless, it is also an error that is caught, it will print to the user no path exists and then asks if the users wants to run again, skipping running Bellman-Ford.


```
-----------------------------------------------------------------
Sorry, there is no path between  4  and  8  even before nodes fail.
Press q to quit, any other key to run again:
```