UNIVERSITY OF NEVADA, RENO

Dynamic routing mechanism design in a faulty network

Gabrielle Talavera

CPE 400

Computer Communication Networks

December 4, 2019

Introduction

This project simulates a mesh network where nodes and links may fail. Nodes may fail intermittently, and as an input to the simulation, each node and link will have a certain probability to fail. When such failure occurs, the network must adapt and re-route to avoid the faulty node. The network is created based on how many nodes and the probability of node failure which the user inputs. Edges are generated randomly based on the amount of max edges that can exist. All nodes in the network except the source node may fail. The mesh network updates one time by itself in the beginning, and then will either update or end the program based on if the user inputs 'q' or not. These updates represent intervals or cycles.

In the start of the program, the terminal will take user inputs, and then finds the least-cost path from the source node to the destination node using the Floyd-Warshall algorithm. The least cost path, number of hops, and cost will be outputted. It will then go through one round of node failures. Certain nodes will have a chance of failing, and the failed nodes will be outputted to the terminal. The updated least-cost path, hops, and cost will then be outputted to the terminal. The values will change whether or not the nodes that were in the least-cost path originally failed. The program will then prompt the user to input a 'q' to quit the program, or to enter a random key to continue with updates. The nodes that failed will stay offline for the whole program.

There is an extra feature in this simulation that can compare the calculations of the Floyd-Warshall algorithm to calculations of Dijkstra's algorithm and the Bellman-Ford algorithm. If the user decides to include the features, the hops, total cost, and path for each algorithm would be outputted to the terminal.

1 FUNCTIONALITY

This project is meant to simulate intradomain routing over a mesh network between a set of routers in the same routing domain. Instead of using protocols like open source shortest path, dynamic source routing, and routing information protocol, I used the Floyd-Warshall Algorithm to obtain the shortest path from source to destination.

The Floyd-Warshall algorithm is used for solving the all pairs shortest path problem which finds the least cost path between every pair of vertices for a weighted graph. This algorithm can be run on any graph, as long as it doesn't contain any cycles of negative edge-weight. The Floyd-Warshall algorithm is used in this project because I wanted to study an algorithm that we did not talk about in class, and because I felt like this was a really efficient way to solve the dynamic routing issue. One run of the Floyd-Warshall algorithm can give all the information needed to know about the network to optimize most types of paths. From knowing the least cost path between every pair of vertices, it can get the path easily. When a node fails, the algorithm can easily run through the network again to get a new shortest path if it was affected.

The network I created is based on a local area network where nodes in the network are connected. The number of nodes is decided by user input. The source node and destination node are also user inputs. Each interval in the simulation represents time. The graph will keep going through intervals until the user inputs a 'q'. Through each interval, depending on what the probability of failure is that the user inputted, the program will randomly select which nodes should fail, and if the node's probability is higher than the probability of failure then those nodes will be disconnected from the network. I made it where the source node can not be disconnected from the network. Once the failure nodes are disconnected, they will stay disconnected throughout the entire program. The Floyd-Warshall algorithm runs through again and updates the shortest path network. The updated path, cost, and amount of hops are outputted each time the path is recomputed. This occurs so the network updates

with the faulty nodes and then finds the shortest path. Figure 1.1 shows how the user inputs and the outputs are organized.

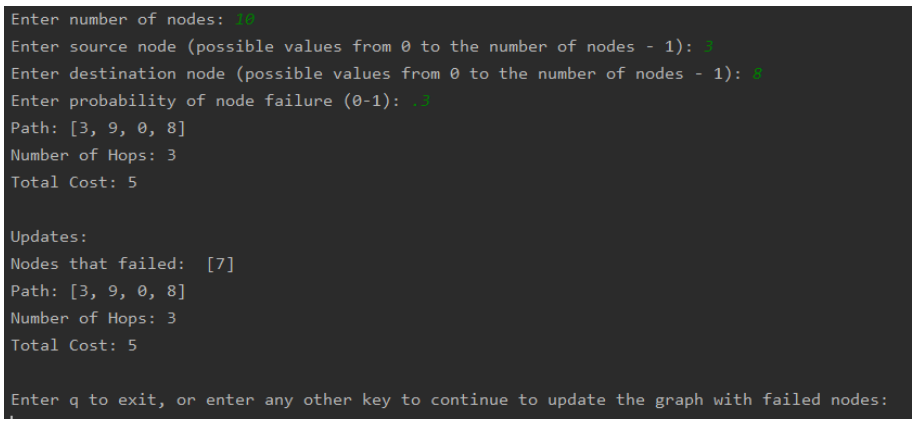

Figure 1.1: Terminal showing user inputs and program outputs

As explained above this protocol is used in different parts of the program. The Floyd-Warshall algorithm is used to calculate the shortest path in the initial network where no nodes have failed yet. It is then used again after the node failures are determined, and it updates the shortest path with the node failures into consideration. Through each interval that it goes through, it will keep updating the shortest path using the protocol. All nodes that are still online will automatically adapt to the new routes each time the Floyd-Warshall algorithm is run.

If there is no shortest path available, then the try-except block should catch the error and print "ERROR: No available path from source node to destination node." Figure 1.2 shows the exception handling when no path is available.

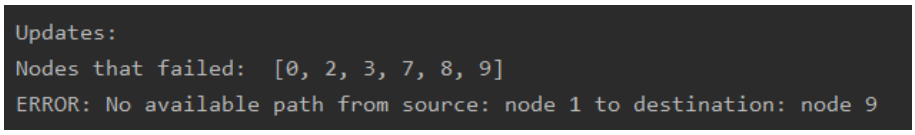

Figure 1.2: Exception handling

I also added a feature where the user can compare Floyd-Warshall algorithm with Dijkstra's algorithm and Bellman-Ford algorithm. To add this feature you have to uncomment lines

107, 112, 122 for Dijkstra's algorithm and uncomment lines 108, 113, 123 for the Bellman-Ford algorithm. This will output the total cost, hops, and path for each algorithm. Figure 1.3 shows an example output of using this feature.

```
Enter number of nodes: 50
Enter source node (possible values from 0 to the number of nodes - 1): 48
Enter destination node (possible values from 0 to the number of nodes - 1): 47
Enter probability of node failure (0-1): .3
Path: [48, 5, 23, 47]
Dijkstra's Path: [48, 39, 23, 47]
Bellman-Ford Path: [48, 39, 23, 47]
Number of Hops: 3
Dijkstra's Number of Hops: 3
Bellman-Ford Number of Hops: 3
Total Cost: 5
Dijkstra's Total Cost: 5
Bellman-Ford Total Cost: 5
```

Figure 1.3: Output using Dijkstra and Bellman-Ford algorithm comparison feature

For optimal performance on this program, the amount of nodes should be limited. While this program can handle a large amount of nodes, it can take a long amount of time for it to execute, since the Floyd-Warshall has to find the shortest path between each node, and the program also has to calculate failure for each node that is selected to fail. If there are a lot of nodes in the network, the amount of nodes that fail will be higher which also leads to a slow down in performance.

2 Novel Contribution

Typically the shortest path is computed using Dijkstra's algorithm or the Bellman-Ford algorithm. In my project I decided to use the Floyd-Warshall algorithm, used for solving the all pairs shortest path problem which finds the least cost path between every pair of vertices for a weighted graph. This algorithm can be run on any graph, as long as it doesn't contain any cycles of negative edge-weight.

I went with the Floyd-Warshall because in real life networks you would not always just have one source node and one destination node. A network is constantly sending packets from different sources to destinations. This program can easily be modified to find the shortest path from any source to any destination. I chose to make the program route from the same source and destination to show that if a node were to fail in the least-cost path between that source and destination, the protocol would be able find the next shortest least-cost path efficiently. Also multi-cast routing could be supported on this algorithm. Since all paths are known, it would be super efficient if one packet needed to be sent out to many destinations.

Another reason I picked this algorithm is because it can handle negative edge weights. Although the Bellman Ford algorithm could also handle negative edge weights, it would be different because of the Floyd-Warshall being an all-pairs shortest path algorithm. This would be especially useful in the real-world where there would be a notion of "monetary cost" to carry traffic over a link. This would be used in an inter-AS protocol, but it is worth it to mention since there would not be too many changes in my program to allow that to be implemented.

The final reason that I picked the Floyd-Warshall algorithm is because as mentioned earlier, I wanted to learn a different algorithm that is not talked about during lecture. The Floyd-Warshall one was an interesting one because instead of doing the usual single-source shortest path algorithms commonly used in networking, I got to learn about an all-pairs shortest path algorithm. I also have never heard this algorithm mentioned in any of my classes, so it was a

good opportunity to learn other ways to find the shortest path in a graph.

Another feature that is novel in my program is that my protocol can compare the Floyd-Warshall algorithm with Dijkstra's and the Bellman-Ford algorithm. As explained in section one, to add this feature, the user must uncomment lines 107, 112, 122 for Dijkstra's algorithm, and uncomment lines 108, 113, and 123 for the Bellman-Ford algorithm. This is helpful because this feautre gives insightful information on the path, total costs, and number of hops using each algorithm.

3 Results and Analysis

To test the network simulation, I ran the program under different probabilities of node failures with the node amount as twenty-five. For each probability I ran five cycles of node failures. The probabilities I ran the simulation under were: .1, .2, .3, .4, .5, and .6. Since the network is dynamically created, the network changes through each run, so I could not really compare each probability perfectly I had three tests per probability with five cycles of node failures for each three tests. The tables below show the data for each probability.

Each row represents the different tests for that probability, and each column represents the cycle of node failures that the network went through. The first column with a number followed by two numbers in parentheses represents the test number and inside the parentheses are the original hop count and total cost prior to any node failures, respectively. The data in the table has three numbers which represent the number of hops, the total cost, and the amount of nodes that failed, respectively. Fail with a number following it means that there was no available path from the source to destination, and the number that follows it, is how many nodes failed.

Probability of .1					
Test #/Cycle	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5
1 (3,4)	3, 4, 0	3, 4, 0	3, 4, 0	3, 4, 0	3, 4, 0
2 (2,4)	2, 4, 0	2, 4, 0	2, 4, 1	2, 4, 0	2, 4, 0
3 (3,3)	3, 3, 1	3, 3, 0	3, 3, 0	3, 3, 0	3, 3, 0

Probability of .2					
Test #/Cycle	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5
1 (3,4)	3, 4, 1	3, 4, 1	3, 4, 0	3, 4, 0	3, 4, 1
2 (3,5)	3, 5, 0	3, 5, 0	3, 5, 0	3, 5, 1	3, 5, 1
3 (3, 10)	3, 10, 0	3, 10, 1	3, 10, 1	3, 10, 1	3, 10, 2

Probability of .3					
Test #/Cycle	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5
1 (1,1)	1, 1, 0	1, 1, 3	1, 1, 1	1, 1, 2	1, 1, 2
2 (3,9)	3, 9, 4	3, 9, 2	3, 9, 3	3, 9, 2	3, 9, 4
3 (3,3)	3, 3, 3	3, 3, 2	3, 3, 2	3, 3, 4	Fail, 2

Probability of .4					
Test #/Cycle	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5
1 (3,4)	3, 4, 3	1, 5, 4	1, 5, 3	1, 5, 4	1, 5, 4
2 (3,4)	3, 4, 1	Fail, 3	Fail, 1	Fail, 3	Fail, 2
3 (2,6)	2, 6, 2	2, 6, 2	Fail, 4	Fail, 4	Fail, 3

Probability of .5					
Test #/Cycle	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5
1 (3,4)	3, 4, 3	1, 5, 3	1, 5, 1	1, 5, 3	1, 5, 4
2 (3,5)	Fail, 3	Fail, 2	Fail, 5	Fail, 3	Fail, 5
3 (2,3)	3, 4, 4	3, 4, 4	3, 4, 6	Fail, 9	Fail, 6

Probability of .6					
Test #/Cycle	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5
1 (3,9)	3, 9, 4	2, 11, 7	Fail, 6	Fail, 6	Fail, 3
2 (2,3)	2, 7, 6	2, 7, 7	2, 7, 8	2, 7, 8	2, 7, 9
3 (2,2)	2, 9, 9	Fail, 7	Fail, 6	Fail, 7	Fail, 7

As you can see from the data, for every increase in probability of node failure, the number of nodes that fail increase. The chance for the path to change which can lead to a change in cost and hop count also increases, since there is a higher probability that the nodes in the original path fail. As the probability increases, the chance for there being no path available

increases. Since the network is dynamic, I could not perfectly choose routes that would keep updating the path since the edges to each node could change every time. Testing it multiple times made it easier to see what was generally going on.

Analyzing each tests individually shows valuable results. The average amount of nodes that fail for each probability is: .13, .67, 2.4, 2.87, 4.07, 6.67, respectively. This shows how the number of nodes that fail increase as probability increases. Hop count and cost averages are not really important since costs are randomly assigned to each link, and a higher or lower hop count does not necessarily mean a higher or lower cost. The costs all depends on the weight on the links between each nodes which is randomly assigned as mentioned in section one. After analyzing the results, the user can see that the network simulation is consistent with expected outcomes during each run.

After each cycle, the number of hops, the cost, and the path was outputted to the terminal. Figure 3.1 shows an example output of the program. The first four lines show the user inputs, and then the lines after that show the path and calculation for the original network, the updated network with the failed nodes, and then what happens if there is no available path which is caught in the try-except block.

```
Enter number of nodes: 25
Enter source node (possible values from 0 to the number of nodes - 1): 23
Enter destination node (possible values from 0 to the number of nodes - 1): 24
Enter probability of node failure (0-1): .6
Path: [23, 6, 24]
Number of Hops: 2
Total Cost: 2

Updates:
Nodes that failed:  [0, 1, 6, 10, 14, 15, 16, 19, 20]
Path: [23, 21, 24]
Number of Hops: 2
Total Cost: 9

Enter q to exit, or enter any other key to continue to update the graph with failed nodes:
d
Updates:
Nodes that failed:  [1, 10, 14, 16, 19, 24, 25]
ERROR: No available path from source: node 23 to destination: node 24
```

Figure 3.1: Terminal Output

The feature that compares Floyd-Warshall algorithm with Dijkstra's algorithm and Bellman-Ford algorithm also gave valuable results. An example of the outputs can be seen in the figures below. Figure 3.2 shows an output where the path that uses Floyd-Warshall algorithm is slightly different from the others algorithms. Figure 3.3 shows the output where all the paths, hops, and costs are the same.

```
Enter number of nodes: 100
Enter source node (possible values from 0 to the number of nodes - 1): 27
Enter destination node (possible values from 0 to the number of nodes - 1): 89
Enter probability of node failure (0-1): .2
Path: [27, 44, 89]
Dijkstra's Path: [27, 67, 89]
Bellman-Ford Path: [27, 67, 89]
Number of Hops: 2
Dijkstra's Number of Hops: 2
Bellman-Ford Number of Hops: 2
Total Cost: 2
Dijkstra's Total Cost: 2
Bellman-Ford Total Cost: 2
```

Figure 3.2: Output where the Floyd-Warshall algorithm has a different path than the others.

```
Enter number of nodes: 150
Enter source node (possible values from 0 to the number of nodes - 1): 37
Enter destination node (possible values from 0 to the number of nodes - 1): 138
Enter probability of node failure (0-1): .3
Path: [37, 76, 138]
Dijkstra's Path: [37, 76, 138]
Bellman-Ford Path: [37, 76, 138]
Number of Hops: 2
Dijkstra's Number of Hops: 2
Bellman-Ford Number of Hops: 2
Total Cost: 2
Dijkstra's Total Cost: 2
Bellman-Ford Total Cost: 2
```

Figure 3.3: Output where all algorithms are equal to each other.

Generally, the results of all the algorithms have a similar cost, path, and hop count. During testing I obtained results where the hop counts were different for the algorithms, but I was not able to screenshot it, and I could not exactly pinpoint the failures to cause that to happen, since the network is dynamic. These specific results show that the Floyd-Warshall algorithm can successfully be used and have similar results to other algorithms when routing

in a network with faulty nodes.

After obtaining the results and analyzing them, I can conclude that the simulation could successfully route dynamically in a network where nodes may fail using the Floyd-Warshall algorithm. When nodes fail, the shortest path would update. The network was able to easily adjust itself by successfully using the Floyd-Warshall algorithm to calculate the shortest paths between all vertices.