CPE 400 Final Project: Dynamic routing mechanism design in a faulty network

By Paige Mortensen and Marissa Floam

Problem Statement

When learning about networking in a class, it is simple to create a perfect, controlled example to reference and work off of. But, when dealing with real networking scenarios, it becomes much more difficult to ensure every possible user scenario is accounted for and handled reasonably. Within networking as a whole, quick adjustments need to be made in extremely short amounts of time in an attempt to maintain operational continuity. The purpose of our project is to create our own simulation of dynamic networking protocols in an attempt to learn more about networking efficiency and network rerouting when a node and or link inevitably fails.

Functionality

Protocol functionality:

In a real-life networking scenario, it is guaranteed that routers and or the links between those routers will run into scenarios where they completely fail, are temporarily unavailable, or have too many packets within its queue. When this occurs, the network needs to have a way to efficiently reroute these packets in order to make sure they still get to the desired destination. Within our code, we simulated both Dijkstra's and the Bellman-Ford routing algorithms.

Looking at the Bellman-Ford algorithm, the shortest path is calculated in a bottom-up manner. Dynamic programming is used to calculate paths and maintain previous path information as it traverses through the graph; accumulating this information requires more overhead. Additionally, this algorithm is slower, as it has a greater time complexity of. For Dijkstra's algorithm, a shortest path tree is created. The greedy algorithm principle is used, and individual decisions are made with each step. This algorithm requires less overhead as each node does not retain the entire path taken before, it can only see the next step ahead. In contrast to Bellman-Ford, Dijkstra's performs faster, as it has a lesser time complexity.

When our code is run, the user is prompted to choose between either viewing the current state of the graph, simulating a node failure, simulating a link failure, or running both algorithms on the same graph state to find the shortest path. To simulate a node failure, each node is assigned a predetermined failure rate. When this option is selected, a node is randomly chosen to fail based on these weighted probabilities. The failed node is then deleted from the graph, and all of its links are also deleted. To simulate a link failure, a similar procedure is followed. Each link has also been assigned a failure rate and, during a link failure, this value is used to randomly select a link to fail. When this occurs, the link is removed from the graph. In both scenarios, we also check to ensure that each node has at least one viable link, or else it is removed from the graph. This is used to simulate a router that is no longer a viable transport. If the user chooses to "Run Pathing Algorithms", both routing algorithms are run on the same state of the graph and

each algorithm is individually timed. Additionally, both algorithms print out the shortest path found and the total distance of said path.

Error Handling:

There are numerous different scenarios in our team's simulation that require error handling. First, before the user is able to simulate a link or node failure, the program has to check to ensure there are nodes and links available in the graph. The program also checks for this before any pathing algorithms are called. If there are no nodes or links in the mesh network to fail, the pathing simulation cannot run. In addition to this, both Dijkstra's and Bellman-Ford's pathing algorithms require a source node to determine the distance from that node to all other nodes. Because of this requirement, the source node can never fail. So, we set the failure probability of the source node 'a' to zero. Furthermore, as a user attempts to simulate node or link failure, the removal of nodes or links is dependent on what they are connected to. Our deleteNode and deleteLink functions ensure proper handling of node and link deletion. Deletion of nodes requires the links attached to it be deleted as well. In the event that a failed node and its connected links are deleted, there may be a stray node leftover with no connections in the graph. Both of these functions delete any nodes that are no longer connected to the graph to prevent this from happening.

Novel Contribution

The main novel contribution in our project includes the ability to simulate two different pathing algorithms on a mesh network after a link or node failure. In order to understand which pathing algorithm is more efficient in rerouting after the failure of a node or link, our team compared the time it took to compute the path as well as the distances of each. An example of the program running is depicted below:

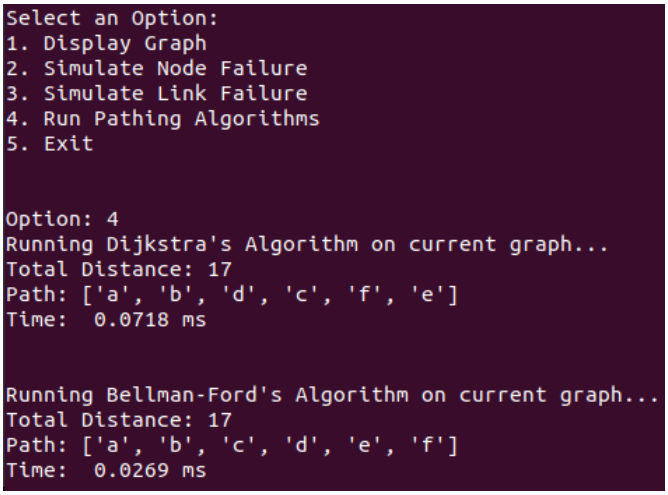

Fig. 1: The program running both pathing algorithms on the default graph

As shown in Fig. 1, Dijkstra's and Bellman-Ford's pathing algorithms route through the graph differently. While they have the same total distance, the time it takes to run Bellman-Ford's algorithm on the default graph is shorter. After a node fails, we can re-run the pathing algorithms and the program will output the following:

```
Option: 2
Node b has failed.
Node b has been removed from the graph.
Remaining nodes:  ['a', 'c', 'd', 'e', 'f']
Remaining links:  ['ac', 'cd', 'de', 'df']
```

Fig. 2: The program prints out remaining nodes and links after a node failure

```
Option: 4
Running Dijkstra's Algorithm on current graph...
Total Distance: 25
Path: ['a', 'c', 'd', 'f', 'e']
Time:  0.0724 ms


Running Bellman-Ford's Algorithm on current graph...
Total Distance: 25
Path: ['a', 'c', 'd', 'e', 'f']
Time:  0.0727 ms
```

Fig. 3: The program prints out a new comparison of the pathing algorithms after a node failure

As shown in Fig. 2 and Fig. 3, the pathing algorithms are able to adapt to the faulty node and continue to re-route and find the shortest possible path. In this instance of the simulation, Bellman-Ford actually took slightly more time to compute than Dijkstra's, but the total distance was the same. Below is a visualization of both pathing algorithms in action.

Before node removal:

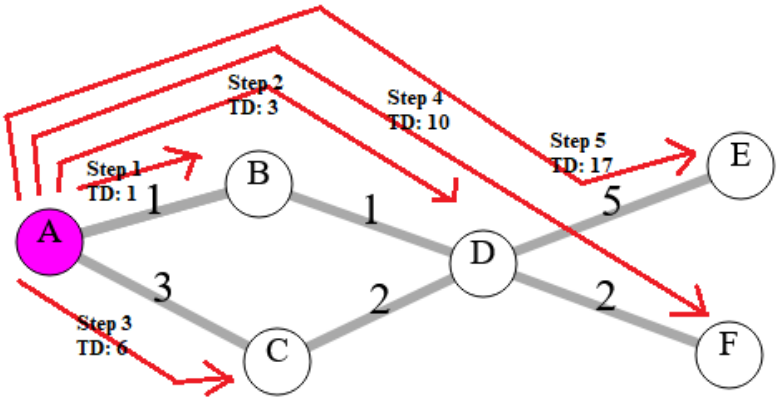

Fig. 4: Visualization of Dijkstra's running on our default graph. TD = Total Distance

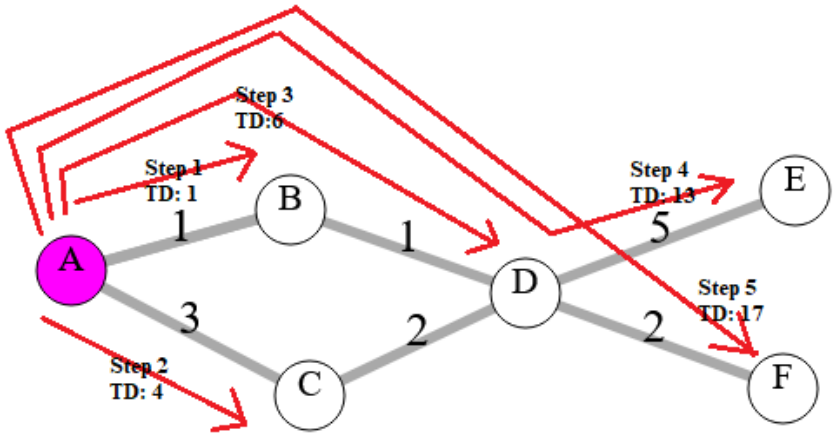

Fig. 5: Visualization of Bellman-Ford running on our default graph. TD = Total Distance

After node removal:

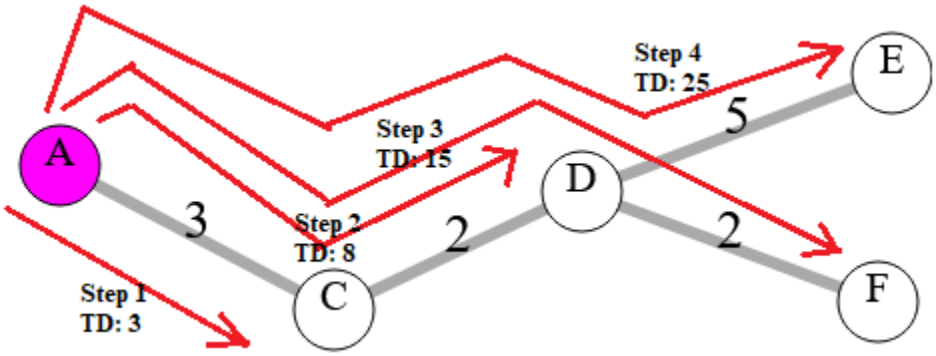

Fig. 6: Visualization of Dijkstra's running on the graph after node 'b' failure. TD = Total Distance

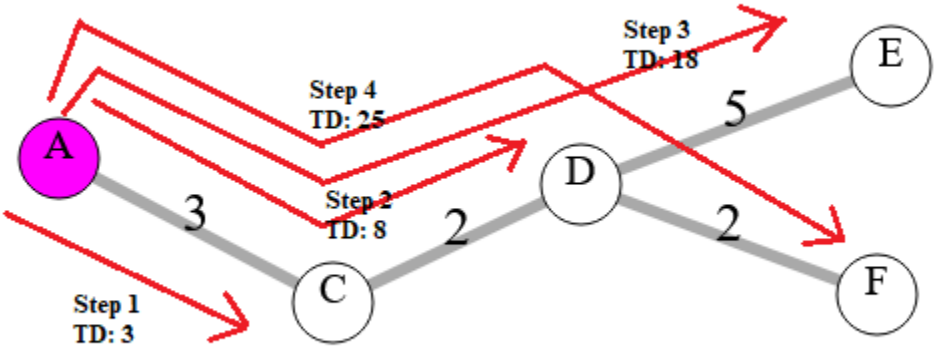

Fig. 7: Visualization of Bellman-Ford running on the graph after node 'b' failure. TD = Total Distance

As shown in Fig. 4 and Fig. 5, the pathing each algorithm takes is different but the total distance ends up being the same. This is why it is necessary to calculate the time each algorithm takes to compute the shortest possible path to further prove which is more efficient in the event of a failed node or link. Fig. 6 and Fig. 7 show the algorithm running on the graph after the

removal of the failed node, ‘b’. In this case, both the algorithms also take different paths but end up with the same total distance. Thus, the analysis of which algorithm works more efficiently is necessary and will be discussed in the results section.

The Code: Functions and Classes

Graph Class

A simulation of a mesh network makes most sense when displayed as a graph, where each node can represent a router and the weighted edges between them represent the delay in communication. The default constructor of the Graph class includes initialization of the links, nodes, probability failures, and source node of the Graph. The required methods include node deletion, link deletion, and graph display.

The attributes contained within the Graph class are as follows:

- nodes: Contains a dictionary of all the nodes as keys and their failure probabilities as values.
- links: Contains a dictionary of all the nodes as keys and a dictionary of its neighbor nodes as values. The nested dictionary contains all the connected nodes as keys and the distances as values.
- linkProbFailure: Contains a dictionary of all links as keys and their failure probabilities as values.
- sourceNode: Sets the graph’s starting node as ‘a’ to ensure proper pathing.

The methods contained within the Graph class are as follows:

- deleteNode: This function takes the graph and the failed node as parameters. It deletes the failed node from the graph’s set of current nodes. Then, it deletes the failed node from the graph’s set of current links. Next, it deletes all links to the failed node’s old neighbors. Finally, the function will delete all nodes that are no longer connected to the graph. After a complete removal of the failed node, the function prints out which node has failed as well as the remaining nodes and links.
- deleteLink: This function takes the graph and the failed link as parameters. It deletes the failed link from the graph’s set of current links. Then, it deletes the link from the graph’s set of links with probability failure. Finally, the function will delete all nodes that are no longer connected to the graph. After a complete removal of the failed link, the function prints out which link has failed as well as the remaining links.
- displayGraph: The last Graph class method displays the current graph links and nodes.

Functions

- dijkstra: The first pathing algorithm used was Dijkstra's. This function takes the source node, the set of nodes, and the set of links contained in the graph as parameters. It initializes some variables to hold unvisited nodes, visited nodes, and node predecessors. Then, in a loop, this algorithm iterates through all unvisited nodes and through the connected nodes of the current node. It updates the distances after comparing their weights. The function prints the total distance of the path, and returns the dictionary of visited nodes.
- bellmanFord: The second pathing algorithm we used was Bellman-Ford. This function takes the dictionary of links as well as the source node as parameters. It first prepares the distance and predecessor for each node. Then, it relaxes the edges by checking the best-known path from source to another node, and updates the path to reflect the minimum cost. Lastly, the Bellman-Ford algorithm checks for negative weights in the links. It returns two dictionaries of distances and predecessor nodes.
- nodeFailure: This function takes the graph as a parameter and randomly picks a node to fail based on the weighted probabilities. It will print which node has failed and then delete the node from the graph.
- linkFailure: This function takes the graph as a parameter randomly picks a link to fail based on the weighted probabilities. It will print which link has failed and then delete the link from the graph.
- findShortestPath: This function takes the graph as a parameter and sets the source node and current nodes to the graph. Then, it runs dijkstra() and bellmanFord(), calculating the total time each takes. The function prints the shortest path found by each algorithm, the total distance, and the total time each algorithm took to run.
- Menu: The menu function displays a menu to the user that allows them to input an option to simulate node or link failure, display the graph, or run the pathing algorithms.
- main: The main function in the program displays the title, loads in the graph, and calls the appropriate functions based on the user input from the menu. If the user inputs '1', the program will display the current graph. If the user inputs '2', the program will check to see if there are links and nodes available in the graph, and if so, it will call nodeFailure. If the user inputs '3', the program will check to see if there are links and nodes available in the graph, and if so, it will call linkFailure. If the user inputs '4', the program will check to see if there are links and nodes available in the graph, and if so, it will call findShortestPath. In any of these cases if there are no links or nodes available, the program will exit. Finally, if the user inputs '5', the program will exit. There is a default error message that will print if the user enters an option that is not listed.

Results

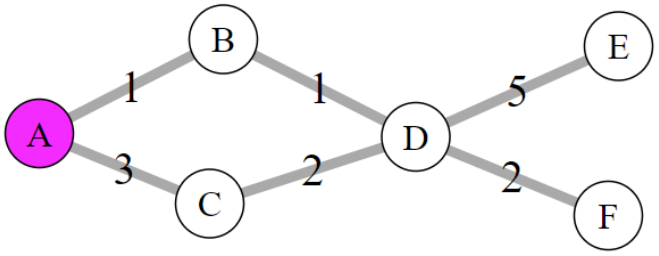

Fig. 8: Our original, default graph

Each time we run the program on our original graph [Fig. 8], both shortest-path algorithms find a different path with the same total distance. However, each algorithm takes a different time to run. To observe this, we ran our program 100 times and took the average time each algorithm took to run. For Dijkstra's algorithm, we found the average run time to be 0.0316 ms. For Bellman-Ford, we found the average runtime to be 0.0006 ms. Additionally, when the same procedure is followed after a node failure [Fig. 9], Dijkstra's is found to run an average of 0.0230 ms, while Bellman-Ford is found to run an average of 0.0004 ms. All runtime averages are depicted below in Table 1.

Table 1: Average runtime for each pathing algorithm

Algorithm	**Average Runtime before Node Failure (ms)**	**Average Runtime after Node Failure (ms)**
Dijkstra's	0.0316	0.0230
Bellman-Ford	0.0006	0.0004

Our data suggests that Bellman-Ford has a quicker runtime, but we know that Dijkstra's has a more efficient time complexity. Taking this into account we came to the conclusion that because our simulation is run on such a small set of nodes and links, the difference in time it takes to run each algorithm is negligible. The operating system allocates resources and executes instructions differently each time the program is run, which explains the seemingly random runtimes of the two algorithms. Our simulation would likely show better results in a very large mesh network, similar to a network in the real world.

Simulating Node Failure:

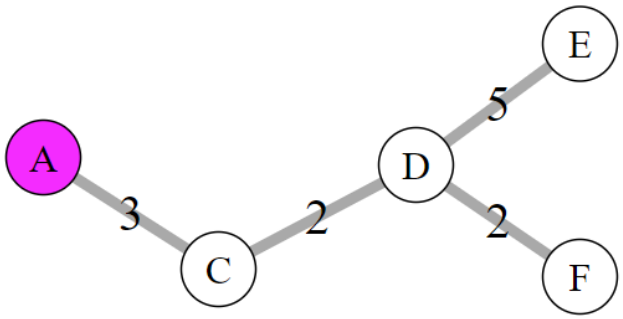

Fig. 9: The graph after node B failure

Simulating Link Failure:

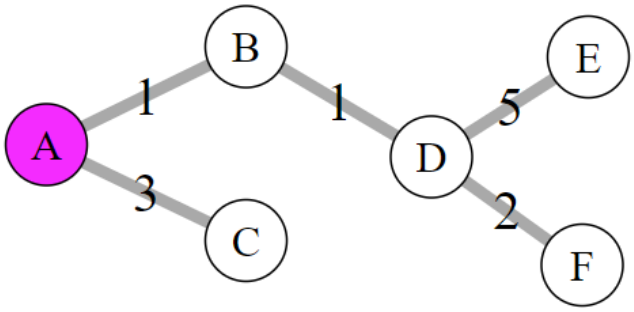

Fig. 10: The graph after link ‘cd’ failure