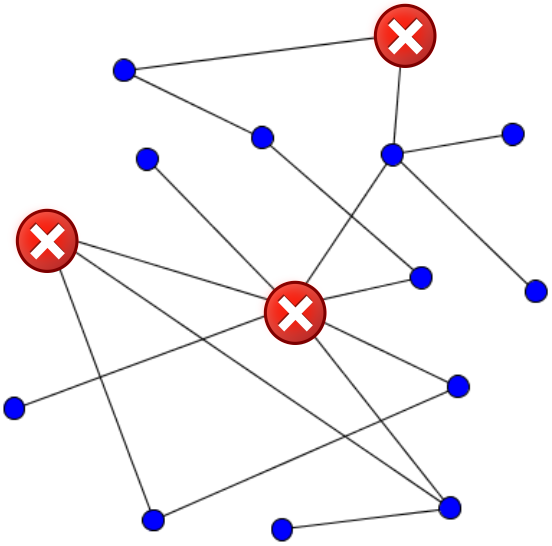
University of Nevada, Reno
Computer Science and Engineering Department

Dynamic Routing Using Bellman Ford For A Faulty Network

Instructor: Tapadhir Das
Student: Cameron McCoy



Introduction:

Maintaining reliable network connectivity is essential for seamless communication and data transfer across a variety of devices, from personal computers to large-scale server networks. However, when network nodes fail, routing algorithms may produce suboptimal routes due to outdated topology information. This can result in slower data transfer and even network downtime, which can create conflicts among multiple parties reliant on the network. To prevent these issues, dynamic routing can be used to continually update network topology and calculate the best routes in real-time. By doing so, dynamic routing protocols can ensure that accurate data is always used to inform routing decisions, even in the face of node failures or other network faults.

Overview:

The initial state of the network is represented by a hardcoded topology that is instantiated and visualized. The shortest path between two nodes is determined using this topology. Subsequently, a loop is executed until statistical anomaly detection occurs, which reveals the nodes that have failed, with a maximum limit of two nodes. Once the failed nodes are identified, all links to these nodes are removed from the network. The post-fault topology is then recalculated and visualized, including the updated shortest path between the original path nodes. This process is designed to improve the fault tolerance of the network by identifying and isolating faulty nodes while ensuring that the shortest path between important nodes is maintained.

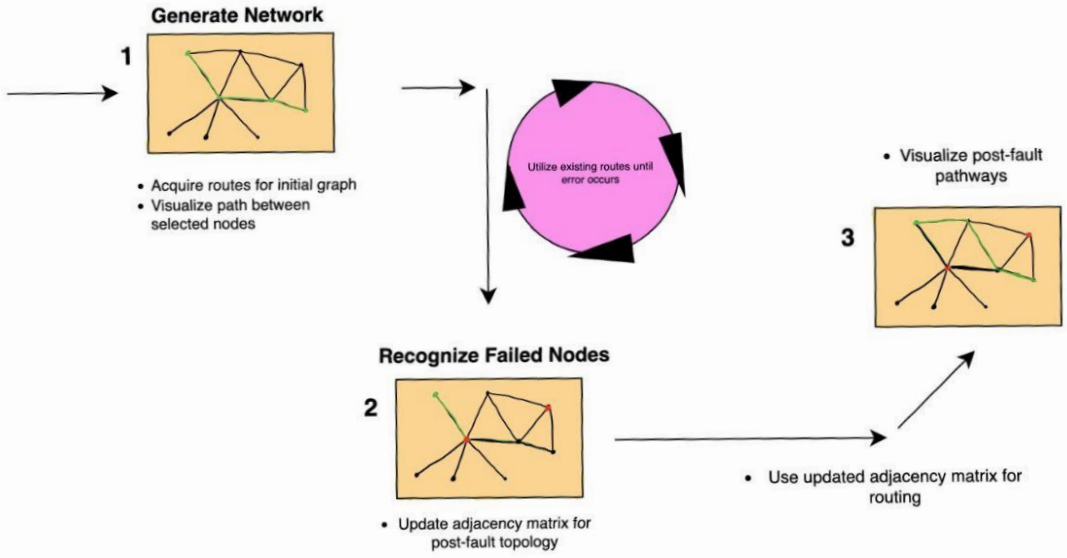


Figure 1: Dynamic Routing Flow States

Design:

Graph Generation

- Main Program / Notebook
- Router Class: Instantiates node with ID, Links, and appends to provided list
- Graph Class: Contains # of vertices and graph list. Methods include edge insert / routing
- Adjacency Matrix: Numpy Array (N x N), where N = # of routers established.

The Adjacency Matrix (AM) provides the link existence and weight data for routing calculations. A graph object is generated with N vertices. The AM is iterated through to add edges with weights to the graph object.

Routing Algorithm

- Bellman-Ford: a method for Graph object (Object should contain proper data before calling)
- Outputs shortest path from a source node to all existing vertices

The Bellman-Ford algorithm is a popular algorithm for finding the shortest path between a source vertex and all other vertices in a graph, including graphs that have negative weight edges. In the context of dynamic routing for fault networks, this algorithm can be used to calculate the shortest paths between network nodes while taking into account potential disruptions or failures in the network.

To implement this algorithm, a class called Graph was defined, which has several methods for adding edges to the graph, printing the distance from the source vertex to all other vertices, and implementing the Bellman-Ford algorithm. In the addEdge method, edges were added to the graph, with each edge consisting of a source vertex, a destination vertex, and a weight.

The BellmanFord method implements the Bellman-Ford algorithm. First, initialize the distances from the source vertex to all other vertices as infinite. Next, relax all edges |V| - 1 times. During each relaxation, update the distance and parent index of adjacent vertices of the picked vertex. I only considered vertices that were still in the queue. Finally, check for negative-weight cycles, and if one was detected, terminate the algorithm and print a message indicating that the graph contains a negative-weight cycle.

If no negative-weight cycle was detected, the algorithm printed the distance from the source vertex to all other vertices using the printArr method. The view_graph_list method allows for viewing the graph in the form of a list.

Statistical Anomaly Detection

- Integer containing random number generated between 0 and 100
- While Integer > 5, continue regenerating Integer until condition is met indicating node failure
- During generation loop, time is printed representing fully function routing state
- After failure, two random nodes are selected as failed making sure nodes are not direct neighbors

Node Removal and Update

- Create copy of current Adjacency Matrix (AM)
- Put 0 for all links that contain failed nodes from Anomaly Detection in AM
- Create New Graph Object with existing number of routers and add new linked from updated Adjacency Matrix
- Run Bellman-Ford on new updated Graph object

Graph Visualization (NetworkX / Matplotlib)

- Create NetworkX Graph Object with nodes from Router List
- Add edges with weights from accurate Adjacency Matrix
- Spring_layout for node seed position
- Run single_source_bellman_ford on Graph with provided source node putting short paths in a list
- Color shortest path edges Green, all other black
- Faulty nodes will be colored Red.

Running Program: (https://github.com/cabmeron/DR_BF_FN)

Requirements:

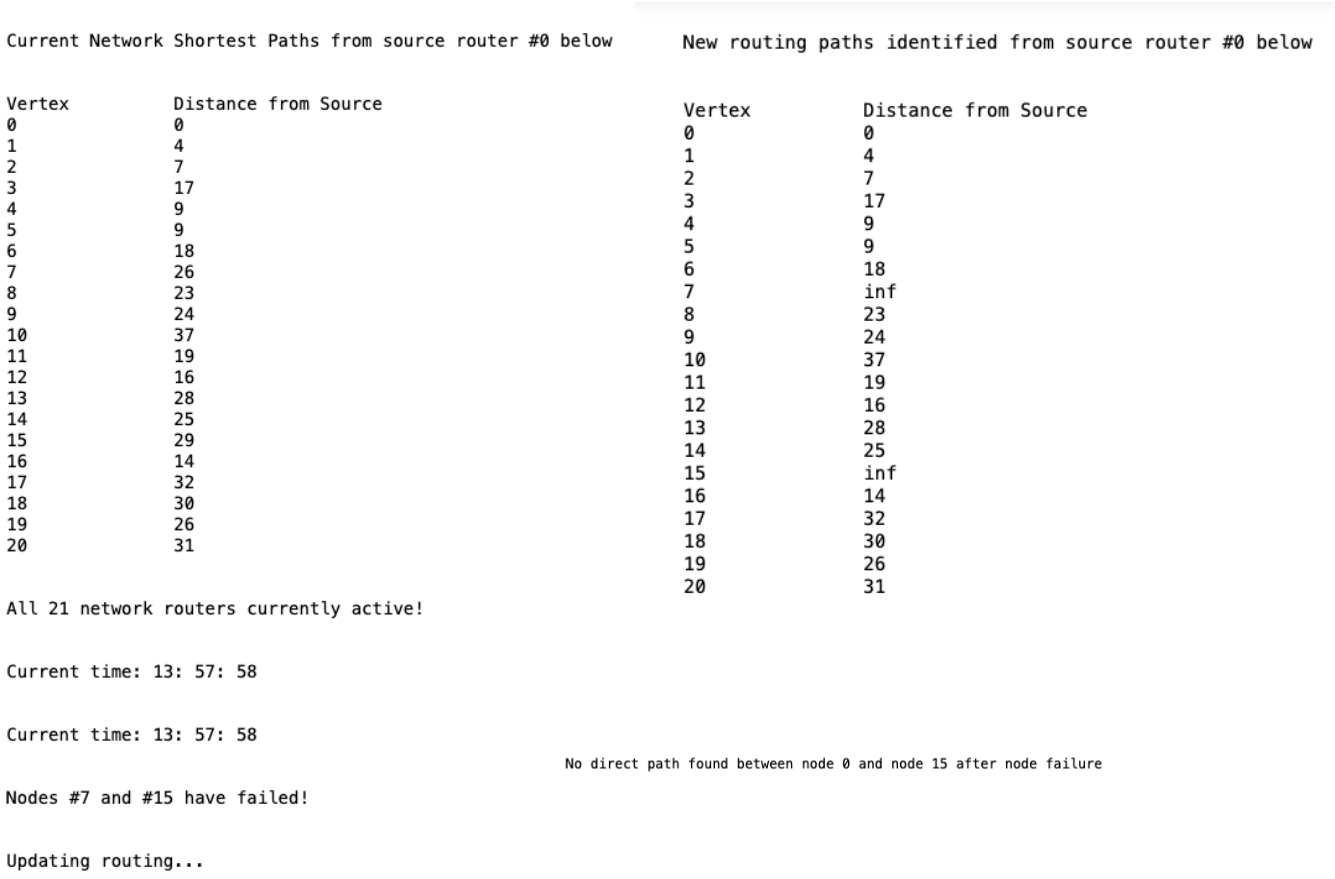
- **Python 3.7 w/ prepared Anaconda Environment**
- **NetworkX**
- **Matplotlib**
- **Anaconda (v23.1.0)**

1) Make sure NetworkX, Matplotlib, Anaconda, and Python with provided versions are installed
2) Download project files from https://github.com/cabmeron/DR_BF_FN
3) Setup provided Anaconda Environment

   conda env create --name envname --file=userEnvironment.yml

4) Open main_notebook.ipynb
5) Click Run on cell (command Enter)
6) Scroll down to see result outputs

Example Output: Running All Cells in Main



```
Current Network Shortest Paths from source router #0 below

Vertex          Distance from Source
0               0
1               4
2               7
3               17
4               9
5               9
6               18
7               26
8               23
9               24
10              37
11              19
12              16
13              28
14              25
15              29
16              14
17              32
18              30
19              26
20              31

All 21 network routers currently active!

Current time: 13: 57: 58

Current time: 13: 57: 58

Nodes #7 and #15 have failed!

Updating routing...
```

```
New routing paths identified from source router #0 below

Vertex          Distance from Source
0               0
1               4
2               7
3               17
4               9
5               9
6               18
7               inf
8               23
9               24
10              37
11              19
12              16
13              28
14              25
15              inf
16              14
17              32
18              30
19              26
20              31
```

```
No direct path found between node 0 and node 15 after node failure
```

Figure 2: Printed Data Regarding Network State

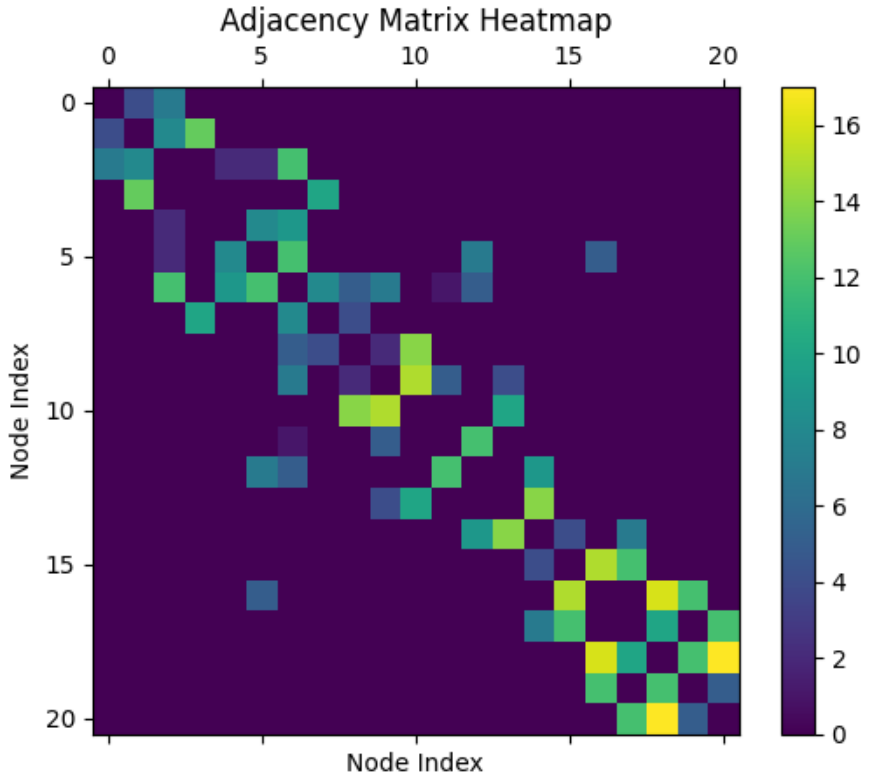


Figure 3: Adjacency Matrix for Graph before Node Failure

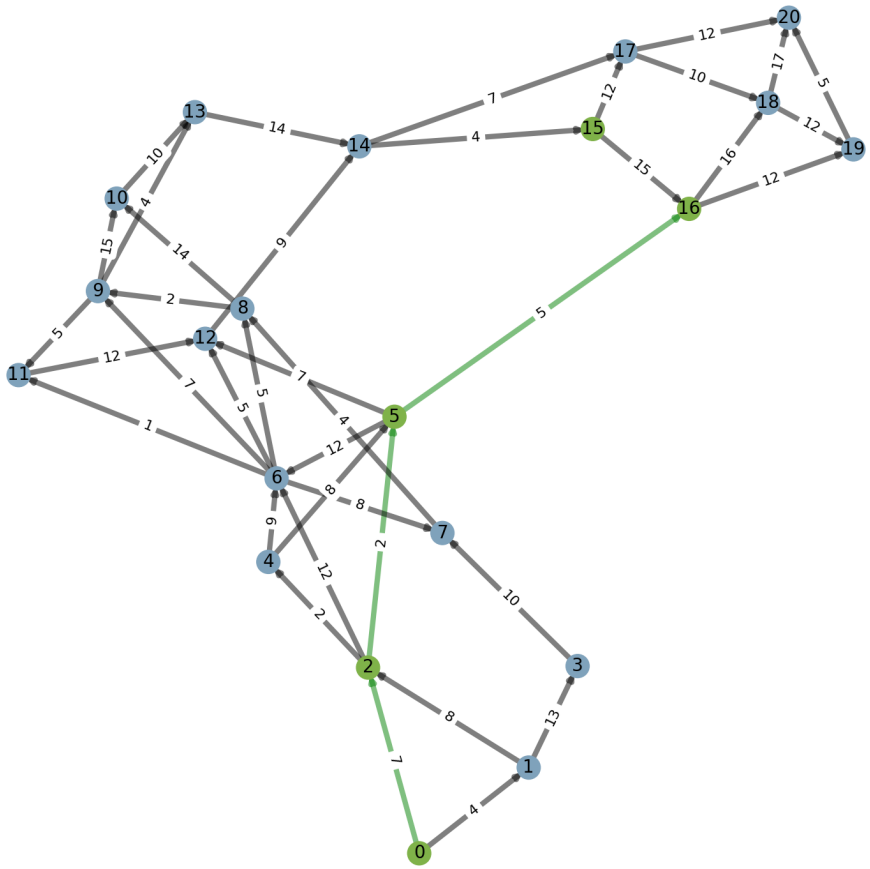


Figure 4: Pre-Failure Graph with Shortest Path Between Nodes

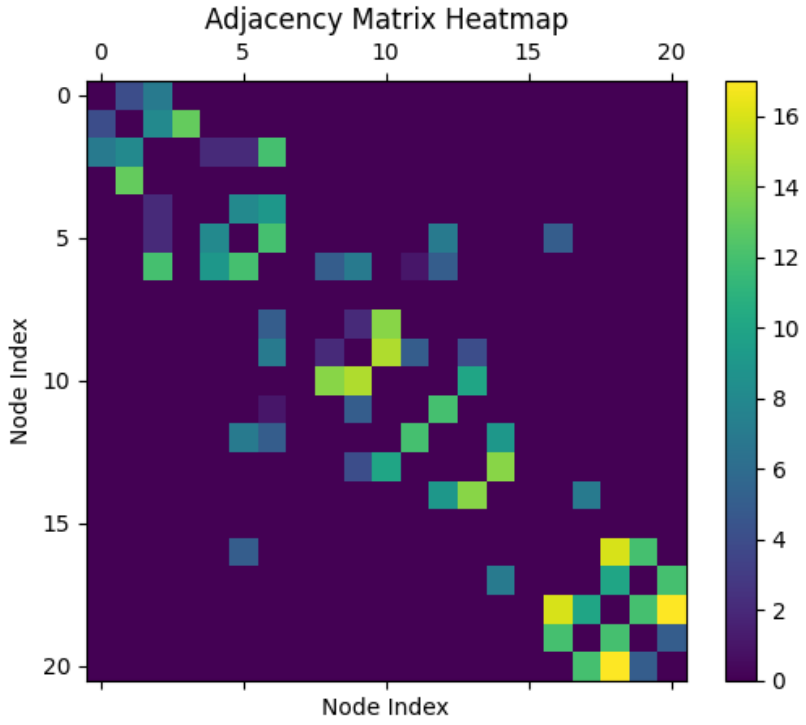


Figure 5: Adjacency Matrix After Node Failure

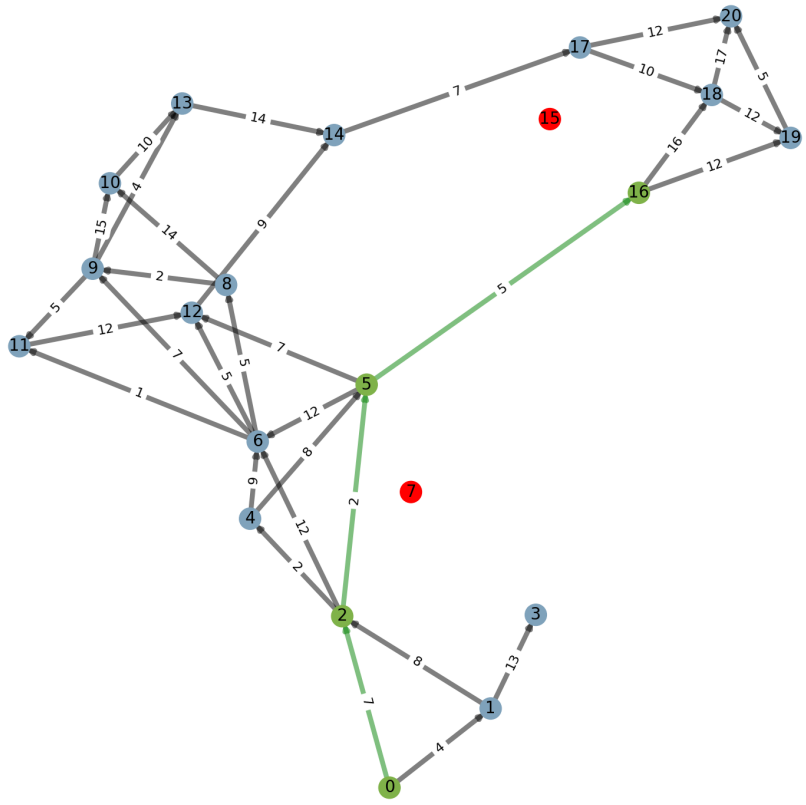


Figure 6: Graph After Node Failure with Shortest Path Between Nodes

Limitations:

- Router topology is hardcoded by design

This was done to prevent a node with a single edge becoming inaccessible in the case of its edge node failing. One solution would be to create a user interface that allows for the user to create nodes and edges within a graph and impose limitations for nodes with single edges. This will create the illusion of custom topologies while still preventing inaccessible nodes from occurring

- Failed nodes are randomly selected, not chosen by user (only two at a time)

A user interface can be created allowing users to select multiple nodes for failure. No duplicate nodes can be selected alongside. This may allow for unrealistic failure experiences that may waste computational resources for user interest.

- Edge weights cannot be generated without going into source code (not true model)

An improvement would implement more realistic modeling of factors that contribute to edge weight such as processing, queue, transmission, and propagation delay

Conclusion:

In conclusion, the implemented program leverages the Bellman-Ford algorithm to dynamically route network paths in the event of a node failure. By iteratively updating the distance and predecessor information of each node, the algorithm is able to effectively adapt to network changes and find optimal paths between nodes, even when certain nodes become unavailable. This approach is particularly beneficial for maintaining network robustness and minimizing communication downtime, as it allows for quick recovery and re-routing in the face of failures.

Moreover, the integration of statistical anomaly detection techniques enabled the system to efficiently identify when and which nodes failed. This not only allowed for timely detection of network disruptions but also facilitated prompt corrective actions. In addition to this, the program provided visualization capabilities, enabling users to better understand the network topology and the impact of node failures. By visualizing pre- and post-failure graphs and adjacency matrices, users can easily monitor and assess the network's performance, resilience, and overall health.

Combining the Bellman-Ford algorithm with statistical anomaly detection and visualization tools results in a comprehensive solution for managing network paths in dynamic environments and ensuring uninterrupted communication between nodes.