CPE 400 Final Project Technical Report

Topic 6: Dynamic Routing Design in Faulty Network

Colton Morley and Jason Brown
Spring 2022, University of Nevada, Reno

Problem Statement

In a simulation, it can be easy to create an efficient network. In the real world, however, technology can malfunction and break regularly. Mesh networks are especially efficient because the nodes are directly and dynamically connected. However when one of these nodes fails, and information can no longer be transmitted through it, it can cause loss or large delays.The goal of this project is to effectively minimize the damage caused by faulty nodes within a mesh network. The team will use a dynamic version of Dijkstra's Algorithm to account for faulty nodes and generate new fastest path tables that will result in a new maximally efficient network after a node has failed.

Functionality

The team has created a simulation using Python to generate nodes, links, and failures. This is then used to simulate a node failure based on individual nodes' likelihood to fail. Whenever a node failure is detected, the routing algorithm is conducted again to determine new shortest paths from each node to all others. At this point, the network is once again optimized, without the failed node. After this, the user can then use the 'add node' and 'add edge' feature to simulate the node coming back online. At this point, a similar process as to the one previously used occurs, and the shortest node distances and node paths are recalculated. The realistic aspect of this simulation comes from the generation of edge weights and node processing times in the graph used. The simulation generates edge weights that represent buffers, loss, and other delays, while nodal processing delay is generated separately, in random but realistic time intervals.

Functions and Classes

Graph Class:

The Graph class is the backbone of this project and supports the entire functionality of the mesh containing nodes and edges connecting the nodes.
It contains:

- *Nodes:* An initially empty set to contain the names for each of the nodes which are created.
- *Edges:* A default dictionary of graph edge values that connect each node.
- *Distances:* a dictionary of distances from each node to every other node.
- *Failure Likelihood:* A default dictionary of default values given to the nodes to simulate their random probability of failing during operation.
- *Nodal Processing Time:* A list of randomized float values between 0.0 and 5.0 to simulate the processing delay of each node.

Its methods:

- *addNode*: A function which takes a name for the node and it's failure likelihood. This function appends the node set, failureLikelihood set, and node processing time set.
- *removeNode:* A function which removes a given node from the graph. It removes the node from the nodes set, then loops through the nodes and deletes any edges that it connects to.
    - Note - although it's titled removeNode, it also removes the Node's edges
- *addEdge:* A function which takes the two connecting nodes and the edge's weight as parameters. It then appends the two node's edges data and distance data.
- *printDefaultNodeDelay:* A function which prints a display on the terminal of each node's randomized processing delay.
- *display:* a function which displays a summary of the graph. It outputs a list of the currently existing nodes, their failure likelihood (%) and the nodes they're connected to through their adjacent edges.

loadGraph:

The 'load graph' function takes the parameters: graph, node, edge, and failure probability. It takes the Graph class's object, then edits and appends the graphs node, edge and failure values given the default values which were declared at the start of the file. Simply put, the function puts the default data where it needs to be. The function then returns the graph object.

Dijkstra's Algorithm:

Dijkstra's Algorithm was chosen to implement this simulation because it collects more information about the vertices of the graph that can be used for calculations and because a network will not contain any negative weights that can cause issues with the Dijkstra algorithm.

This instance of Dijkstra's algorithm is used to calculate the shortest distance from each mesh node to every other node and returns a dictionary which stores lists of paths to other nodes from each node. It takes into account the weights of the edges and the processing times of the node (which is treated as additional weight). It also contains a dictionary for the paths of the algorithm and records the last hop of each calculation. For example, in order to get to 'F' from 'A', the last hop recorded is 'D'.

nodeFailure:

NodeFailure is a function which takes the graph object as a parameter and runs an infinite loop until a certain condition is reached and a break statement is activated.

This function is responsible for simulating a real world failure probability scenario. It does this by creating a random number between 0 and 100, then comparing that number to the default failure probabilities we previously gave the nodes. If a node fails, this function calls the removeNode function to remove the failed node from the graph and all it's adjacent edges. It has no return value.

Menu:

The menu function is responsible for the user's input and the program's response. It displays a table of options that the user has to select from, then creating a control flow for which functions should be called based upon the user's input. The menu options are listed below and their according functions are called if the user selects that option.

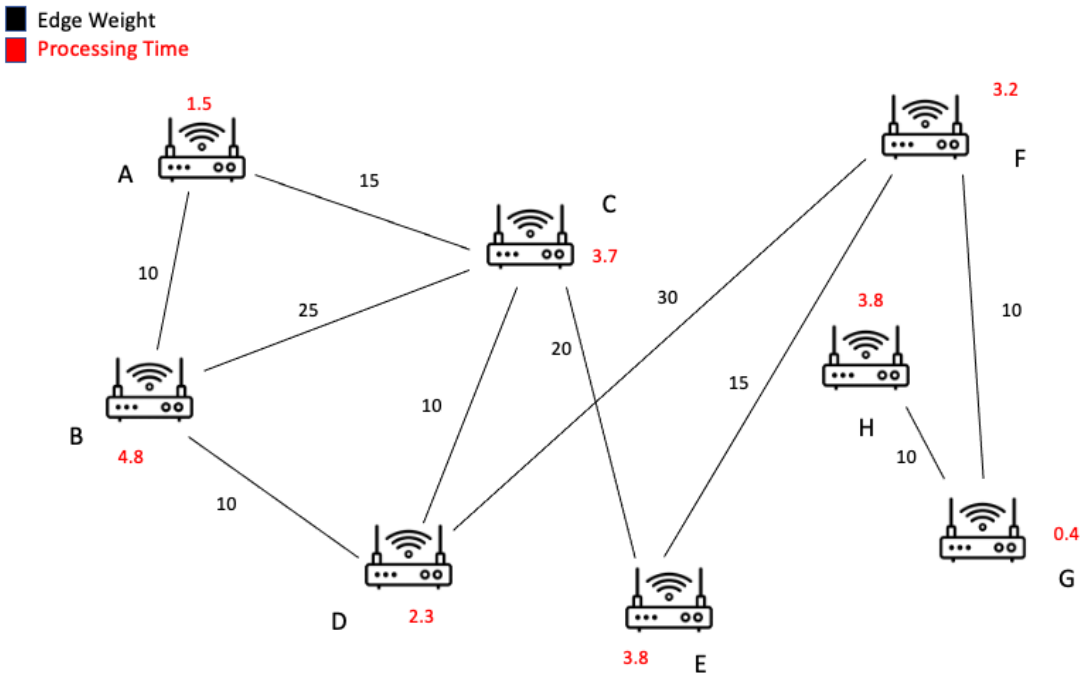
```
def Menu():
    print("\n\n\n+----------------------+")
    print("| Please Select Option |")
    print("+----------------------+")
    print("[1] Simulate Until Failure")
    print("[2] Create Node")
    print("[3] Create Edge Between Nodes")
    print("[4] Force Node Failure")
    print("[5] Display Node Processing Delay")
    print("[6] Calculate Node's Distance to Surrounding Nodes")
    print("[7] Display Graph")
    print("[8] Exit\n")
    userInput = input("Select Option:  ")
    return userInput
```

Results

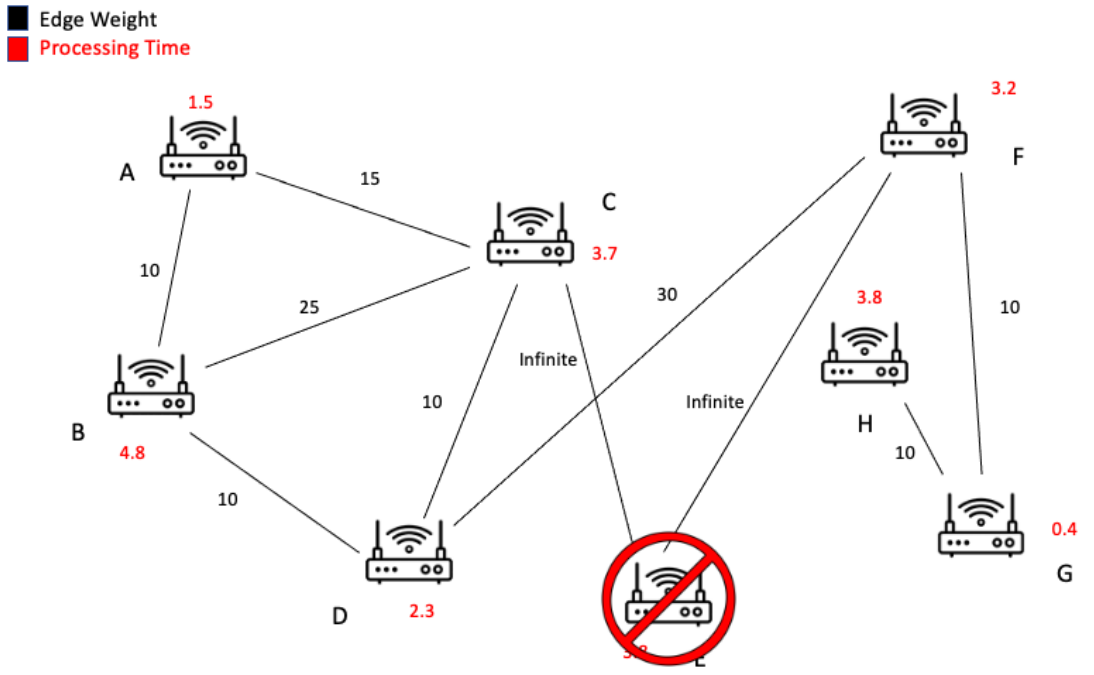
The results of running this simulation with a network small enough that it can be checked with math by hand shows a successful dynamic routing algorithm that correctly responds to failed nodes and calculates distances accordingly. An example run of the simulation is illustrated below for clarity.

Within this simulation, the nodal processing times are randomized, so although an exact replica may be impossible to recreate, a similar outcome should be able to be obtained.

When the program is first created, a simulated mesh network is generated which is similar to the one drawn below. It consists of 8 nodes labeled A-G, with each node having a processing time and each edge having a weight to represent other delays.



The option ‘run until node failure’ was then selected, at which point node E failed, which had edges to C and F. The routing algorithm was then successfully rerun with the exclusion of node E.



This then results in a new shortest path table for each node being generated, along with the overall travel time. The table below represents paths from node A, however these calculations are done for each node in the network.

Table 1. Travel times from Node A

To Node	Last Hop	Total Time (ms)
A	A	0
B	A	14.8
C	A	18.7
D	B	27.1
F	D	58.3
G	F	68.7
H	G	79.5

This table demonstrates the successful calculation of new shortest paths to nodes throughout the network after a node in the network has failed.