# COMP3221 Assignment 1 Report

## Network Topology:

### Creating The Network Topology

Firstly, the program adds a specific number of nodes to the graph with the first node starting at the letter A.

Since all the nodes are disconnected, the first node of the disconnected graph is chosen to be the first node of the connected graph.

The program then picks a random node in the connected graph, that has a degree of less than three, if it has a degree of greater than three a different node is chosen. The program then chooses a random node in the disconnected graph and adds an edge between the two chosen nodes. Finally, the node chosen from the disconnected graph is then removed from said disconnected graph. This process is repeated until the disconnected graph is empty.

If the number of edges wanted is greater than the number of nodes, a second loop will happen to add edges between nodes at random. After all edges are assigned, they are then given a random weight from 0.1 to 9.00.

A configuration file is then written for all nodes where it contains the degree of their node, it’s neighbours and the ports and weights associated with those nodes.

The first stage results with a fully connected graph with minimum one link between each node:

A network of dots and lines

Description automatically generated

The next stage prioritises nodes with less than 4 connections and adds connections randomly:

A diagram of a network

Description automatically generatedIt is observable that the topology has the original structure, but simply has more connections (such as B now connecting to C etc.

Finally, weights are added to the diagram randomly as discussed previously.

A diagram of a network

Description automatically generated

As such, the program always creates a fully connected graph. Additionally, built in functions allow for the user to specify how many nodes and links they would like in the graph. For basic testing of the node functionality a standard 3 node graph with 2 connections was used.

## Routing Algorithm:

### Generalized Implementation

Firstly, the program is given a graph *G* and a starting node *starting\_node.* The program will then set a dictionary*, shortest\_path,* which holds the least cost path from *starting\_node* to every other node as well as the path taken. This dictionary will be updated if there is a shorter path found for the destination node.

This approach is accomplished by first looping through the neighbours of the starting node and finding the edge cost of each neighbour which is then added to the heap. This heap is sorted from least edge cost to greatest edge cost. The path to the neighbour is then checked to see if it already exists inside the shortest path dictionary:

* If it is: the weights of both paths are compared, and the shorter path is chosen.
* If not: the neighbour, its path and path cost is added to the *shortest\_path* dictionary. The neighbour is also added to the heap queue to be accessed at a later iteration.
* The algorithm will find all the neighbours of the current node.
* We will greedily explore the least cost path iteratively, until all paths have been searched.
* We update path costs when they are better than the original pathing.
* To implement the greedy nature of the algorithm we used a priority queue that allows the user to pop the least great distance and explore that node etc.
* As new nodes are discovered, their distance and the node themselves are added to this priority queue and looped through accordingly (whilst there are still nodes to discover.
* Similar to the original Dijkstra’s algorithm that ours is modelled off, it has not been tested for negative weights (as we assumed there would be no negative weights) but can handle infinite weightings.

This method will make sure all connected nodes are visited leaving the unvisited\_nodes list empty.

### Modifications and Optimizations

Instead of finding every node in the graph and applying Dijkstra’s algorithm with each node set as the destination node. The program implements a modified algorithm, where it sorts all the new neighbours from least to greatest edge cost, storing them in a heap queue and then looping through the heap, popping the least edge cost neighbour which will then allow the algorithm to check its neighbour’s, and so on.

## Implementation Methodology:

### Node Object Implementation

The program implements its own node object, so it can keep track of its state (whether it is online or not) and the state of its neighbours.

The functionality inside the node object include:

* Its own map of the nodes its aware of and their edge costs.
* The ability to take itself online and offline (for testing purposes)
* The program implements the ability to remove and add a connection to simulate a node going offline or coming back online respectively. This allows for the network to handle node disconnections, and being aware of which nodes are online and whether they should be included in the least cost path algorithm.

Our threading is implemented through assigning one listening thread to the listening functionality, one for sending packets to other nodes, one for routing calculation and the main thread is used to read console input for the CLI to help with testing node failure.

For the listening functionality of the node, the program treats the node as a server that opens its assigned port for listening to all incoming information. During initialisation the program makes sure all clients are connected to the server. To accomplish this, the program uses a while loop that does not stop until the number of sockets connected equals the number of neighbouring nodes in its config file. It then checks if a client has sent information to the server. The server will then read this information which will go towards expanding the node’s map of the network.

For the sending functionality, the node first connects to each of its neighbours through the ports given in its config file. If the node is unable to connect, an exception is raised saying: “Failed connecting to {port}: {exception}”. After the node has connected to all neighbours, a while loop is executed if the node is online, where the node sends a map of its known network to its neighbours and waits three seconds for a return message. If a message is not received within the next 3 seconds, the program assumes that neighbour is offline. To handle when a node’s server comes back online, it catches the *ConnectionResetError* exception and continues looping back through the while loop.

For the routing calculations, the program checks whether the node needs to recalculate their routing table which is decided by the *reroute\_flag* flag, the flag true when there is a change in the **decode\_topology()** function in NodeObj.py or if the host node becomes aware of itself or a different node going offline. If the flag is True it is flipped and the routing calculation function **routing\_table(),** which is contained inside the nodegen.py file, is called. See section 2.1 for **routing\_table()** implementation.

## 4 Simulation Results:

There are three main incompletions of our implementation.

1. **Reconnecting disconnected topologies**. If the topology of the network ever becomes disconnected and reconnected(i.e., the original topology is split into two separate topologies because of nodes going down), when the reconnection occurs, there is a likelihood that the topologies may conflict. If this happens then there will be a conflict between the network as the network is unsure which topology is correct. I.e., if two disconnected topologies occur then one topology will miss out on packets that should be sent through upon reconnection (but won’t be due to a implementation error).
2. **User Input causing packet loops:** If a user sends user inputs through the CLI too quickly, the network may enter a state of looping. An implementation flaw means that the network does not understand when packets should ideally expire. As such, packets flood through the network until all nodes have seen the packet. If two packets are being continuously sent, then the topologies will continually update. This could be fixed by timestamps attached to the packets, but this will ultimately not be done in time. To prevent this, user input should await doing contradictory changes to the network (i.e., spamming a node up and down), until the entire network has received the original command. This problem is easily fixed if the wait time specified in the specifications for packets was significantly smaller (i.e., less than a second).
3. **None updated configs.** The ‘change’ command does not update config files. This just wasn’t done in time.

Additionally, whilst not outline in the specification but a poor practice regardless is that the terminals themselves must be shutdown to stop each node program.

**Successes:**

The main success of the final product is that the nodes fully propagate information throughout the network successfully. All nodes will learn the entire network once it has fully propagated through the network. Turning a node ‘down’, up or changing a weight between nodes also fully propagates through the network successfully. As such, the network does function, and can update itself successfully. The routing system is always correct *given* that the nodes receive the correct packets (and don’t suffer from the above outlined issues).

**Example:** Basic example for the given topology:

A blue line with a red dot

Description automatically generated

|  |  |  |  |
| --- | --- | --- | --- |
| **Stage**: | Node A: | Node B: | Node C: |
| Start Up | A black background with white letters  Description automatically generated | A group of white letters on a black background  Description automatically generated | A black background with white letters  Description automatically generated |
| Packet 1 sent: | A black background with white letters  Description automatically generated | A black background with white letters  Description automatically generated | A black background with white letters  Description automatically generated |
| Change C A to 10 | A black background with white letters  Description automatically generated | A black background with white letters  Description automatically generated | A black background with white text  Description automatically generated |
| A DOWN | - |  | A black background with white text  Description automatically generated |
| A UP | A black background with white text  Description automatically generated | A group of white letters on a black background  Description automatically generated | A black background with white letters  Description automatically generated |

In the above example we see all the basic functionalities. The initial flooding of information, then a network weight change, a node going down and then a node coming back up.