COMP333 Assignment 2 Part 1 Report

Name: George Hayward
Student Number: 43692486

Part A) Find Number of Routes

Introduction

The aim of this task was to get the number of paths between every combination of stations on a given network. Following these rules:

- If there was an infinite path (a cycle) between two stations then the number of paths would be -1.
- If there was no way to get between the two stations then the number of paths would be 0.
- If there where actual paths between the two stations then the number would be the total number of paths.

Before I started coding this program, I determined what sets of data I would require to meet the requirements above. Data Sets:

1. All paths between every pair of nodes.
   a. Do this using dynamic programming if possible, to eliminate the need to recalculate every path. This was not done.
2. Get all cycles present in the graph. Then combine all stations which are a part of cycles into a set: cycleList

Then run the rules of the task, if any path contains a station within cycleList then the result will be -1. Otherwise it will be the true value of the number of paths between the two stations.

Implementation

The following implementation section will describe what each part of the program does during its natural flow.

File input:

```
public void processInput(String infile) throws IOException {
```

The processInput method works by going through every line on the input file and splitting the two stations by the blank space. The left side being the origin station and the right side being the destination station. Fill the connection map with this information.

```
HashMap < String, ArrayList < String >> connectionMap;
```

This connectionMap will contain keys that are all the origin stations from the left side and the ArrayList value will hold every station that they origin will go to. This is useful for getting the directed neighbours for each station with ease.

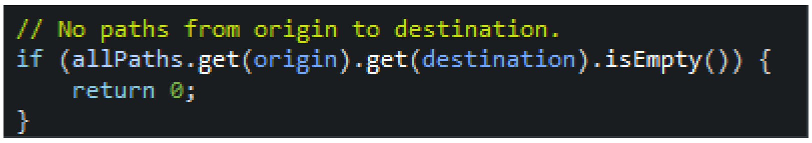
Generate number of routes:

```
public HashMap < String, HashMap < String, Integer >> findNumberOfRoutes() {
```

This method will simply add every combination of two stations to the HashMap including there number of paths, this will return the desired value of the task. One thing to mention is that when adding the actual Integer value for the inner HashMap **getNumberDFS(origin, destination**) is called. This method will generate the appropriate integer value. E.g. -1, 10 or 0.

```
public int getNumberDFS(String origin, String destination) {
```

This method works by calling for the generation of paths and cycles for the network. Then returning the appropriate integer value depending on the specification rules. It is essentially a large conditional method to get the correct answer based on the comparison of datasets. The code has been commented so it is clear what each condition is doing. However, here is an example of a condition which will return 0 for obvious reasons.



```
// No paths from origin to destination.
if (allPaths.get(origin).get(destination).isEmpty()) {
    return 0;
}
```

```
public void getPaths(String origin, String destination, ArrayList < String >
visited, ArrayList < String > pathList) {
```

This is a recursive depth first search approach to getting all paths between the given origin and destination station. It works by adding the paths to a global HashMap variable allPaths. This is a similar data structure to the required return value of the task but instead of the inner HashMap value being an integer, it is all the paths found between the origin / destination. It is stored in a global variable permanently for data comparison as mentioned earlier.

Generate cycle list:

This part of the task required an algorithm to get all the strongly connected components of the graph. There are a few well known algorithms to do this, I decided to go with Korsaraju's Algorithm as it seemed the simplest for me to understand and implement.

Korsaraju's Algorithm – Modified code: https://github.com/mission-peace/interview/blob/master/src/com/interview/graph/StronglyConnectedComponent.java

Korsaraju's algorithm works by running a DFS two times. This first run fills a stack with stations which have been completely explored (no more neighbour paths to explore) by finishing time in reverse order. The second pass of DFS runs a reversed graph implemented by the method:

```
public HashMap < String, ArrayList < String >> getReversedConnections(){
```

which returns the reversed version of the original network. The second DFS then uses the stack generated from the first DFS to find strongly connected components individually. All the strongly connected components are stored in a variable SCCList.

Once Korsaraju's Algorithm has retrieved all strongly connected components, all stations from SCC's that are greater than size 1 (because SCC's can be single leaf nodes) are added to the returned value or in this programs case cycleList. cycleLisst is an arrayList which contains every single station which is a part of a cycle (or SCC in this instance).

The program then finishes off as mentioned in **Generate number of routes** by running the data sets of allPaths and cycleList through conditional statements to give the returned hashMap the appropriate inner integer value.

Note

Unfortunately, I did not have time to implement the dynamic programming part of the getting paths algorithm. However, I will give a brief explanation of how such algorithm would work.

Start by getting paths between two stations. Then for every station on that path (from origin to destination e.g. A>B>C>D>E), add those paths also from the next station (after pruning the origin station) to the destination station (e.g. Add B>C>D>E, C>D>E, D>E to their respective origin / destination hashMap locations). This will greatly reduce the number paths needed to be generated.

Part B) Compute Min Devices

Introduction

The aim of this task was to find the minimum amount of power cut to an undirected electrical network to disconnect two given stations. For the rest of this report I will be referring to the power as weight, stations as vertices and segments as edges.

The task is a modified max flow – min cut problem. The modification being there are no directed edges and the vertices (besides the target two vertices) also have weight.

Implementation

File input:

```
public void processInput(String infile) throws IOException {
```

The two vertices that need to be separated are first put into variables source and sink. The naming type is useful for the implementation of the min flow – max cut algorithm.
All non-source / sink edges are then added to a HashMap<String, Integer> called stations, the integer values are the weights of that edge.
All paths between edges are added to a HashMap<String, HashMap<String, Integer>> called segments. This works the same way as station but with the inner HashMap for connecting two edges.
All station strings are added to the stationIndexer arrayList<String> for later use.

Helper methods:

```
public int[][] generateGraph() {
```

This method provides a clean way of generating an adjacency matrix for a given network. It works by iterating over two (one nested) for loops of stationIndexer size. It then fills the values of the matrix's i/j location with the distance (if it exists) between stationIndexer.get(i) and stationIndexer.get(j).

For example, the distance between stationIndexer.get(i) and stationIndexer.get(j) may be:
Lidcombe and Redfern = 50.
Or, if the indexers land on the same station vertex it will be:
Lidcombe and Lidcombe = 23. Because 23 was the vertex weight.

Minimum Cut:

This implementation was sourced and modified from https://www.geeksforgeeks.org/minimum-cut-in-a-directed-graph/ and works like so:

```
    public Integer computeMinDevice() {
```

1. Run Ford-Fulkerson algorithm and retrieve the final residual graph

2. Find a path from Source to Sink in that residual graph
   ```
   public boolean bfs(int rGraph[][], String origin, String destination,
   HashMap < String, String > parent) {
   ```
   This path is stored in parent. A HashMap which is used to get the last visited node on a path. For example, starting at sink. Parent.get(sink) will return the node before arriving at sink on the path. This is used for altering the found path with the new flow. If the method returns false then there are no more paths from source to sink on the graph.

3. All edges which are from a reachable vertex to a non-reachable vertex are minimum cut edges. An example of this might be A-B and the value of that edge might be 5. So, the minimum cut value will be 5.

4. Get the minimum cut and apply it to the path's edges. Matrix example:
   ```
   matrix[stationIndexer.indexOf(station)][stationIndexer.indexOf(parentStation)] =
   matrix[stationIndexer.indexOf(station)][stationIndexer.indexOf(parentStation)] +
   pathFlow;
   ```
   Also add that minimum cut to the finalResult e.g. Add 5 from previous example.

5. Reduce the flow of the path based on the minimum cut made. There entire path will be reduced by that minimum cut including the vertices. Matrix example:
   ```
   matrix[stationIndexer.indexOf(station)][stationIndexer.indexOf(station)] =
   matrix[stationIndexer.indexOf(station)][stationIndexer.indexOf(station)] -
   pathFlow;
   ```

6. Check if that path from source to sink is still possible on this augmented graph i.e. Is there a path from source to sink with all weights being greater than 0? If it is, continue steps 3-5. If it isn't start back at step 2 until no paths can be found.

7. Return finalResult with the minimum amount of edge reduction for the Source and Sink to be disconnected on the graph.