Assignment 2

Part I

Introduction

The first part of assignment 2 had us solve two provided problems. A template and a number of JUnit tests were provided. Both problems were described as related to the train network that we used in *assignment 1*. However, these problems could be solved independently and did not rely on the first assignment. Within this document, I will describe my approach to solving the problems, the problem that they relate to, and any problems I had when completing them.

Problem A

The first problem described a situation in which we needed to be able to determine the number of paths between two stations. We were provided a CSV data file and needed to read in the data. The data file gave us a number of station-to-station pairs that indicated an undirected connection between the stations on each line.

Therefore, the first task in this part was to read the information in the CSV file into an appropriate data structure that would allow me to perform an algorithm that would determine the required information. After some thought, I believed that having the information in a `HashMap` that was mapped to another `HashMap` would allow me to hold the station-to-station information as well as store the number of ways to get to each station. Therefore, my data type was initialised as following:

```
HashMap<String, HashMap<String, Integer>> paths;
```

Considering that when reading in the data file we only receive adjacent pairings between stations, I initialised the `HashMap` by giving the pairing an integer value of 1 if they were adjacent. This made sense as if they are adjacent, they have the one way to one another. Furthermore, I found it useful to place the station name into an `ArrayList` that would have a unique listing of each of the stations. I determined I could use this to get the number of stations and also iterate over each station in the problem.

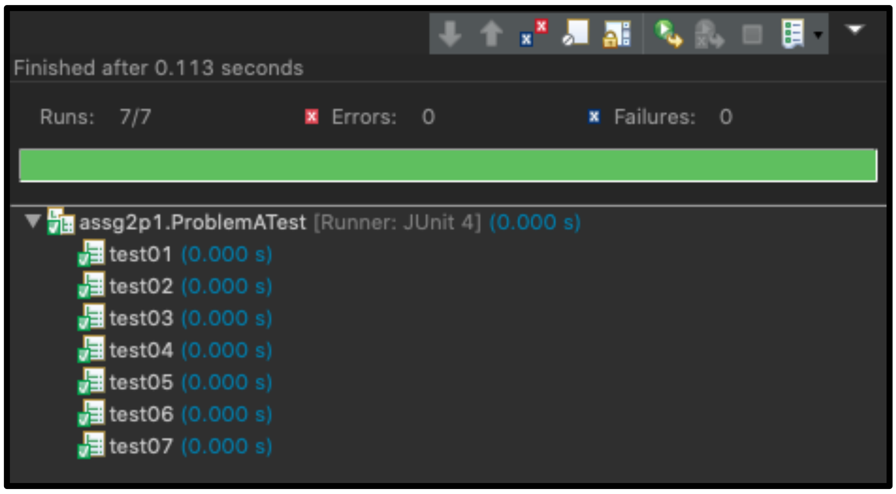
After reading in the information, I began to write out my algorithm. After analysing and researching the type of problem, I discovered that it is similar to the "all-pairs shortest path" problem. However, instead of finding the shortest path, we are trying to find the number of paths. I believed that with some changes, I could use this algorithm to determine the correct output.

Using the template provided by *GeeksForGeeks* [1], I was able to start implementing. Given the way the implementation tries to operate, the first thing I had to do was complete the adjacency list by setting non-adjacent pairings to 0. To do so, I simply iterated through each pair of stations and if they were not in the `HashMap`, I added their pairing to the `HashMap` with a value of 0 as there were no paths between them (yet). I moved this functionality up into the process data as it made sense to do it in this function rather than in the main function.

The next part of the algorithm was simply the *Floyd-Warshall* triple loop. The idea behind this set of nested loops is to pick one by one each node, updating all the shortest paths that contain the picked node as an intermediate node in the shortest path. However, we were not trying to find the shortest path but in fact the number of paths there. So, instead of updating it as a new shortest path, I was able to calculate the number of paths that had been there previously using this as an intermediate node. To do so, I multiplied the two parts of the new path. If they were above zero, we could add this value to the current number of paths as there must be this many new paths to this node, using the selected node as an intermediate.

Finally, the last thing that we needed to do to complete this problem was ensure that any positive cycles be set to -1. As described, if we have any positive cycles, we can always find a new path to destination node. To detect cycles, I simply added a loop that checked if the path to a selected node from that same node was greater than 0. If it was greater than zero, then we know that it must have been able to travel back to itself. Then, we continue with the same Floyd-Warshall strategy and update any path that used that node as an intermediate to be -1 as they must have used this in at least one of their paths.

Upon completing this, I ran the tests. After they all passed, I determined that my solution was correct. The completion of the tests can be seen in *figure 1*.



```
Finished after 0.113 seconds

Runs:   7/7          Errors:   0          Failures:   0

▼ assg2p1.ProblemATest [Runner: JUnit 4] (0.000 s)
    test01 (0.000 s)
    test02 (0.000 s)
    test03 (0.000 s)
    test04 (0.000 s)
    test05 (0.000 s)
    test06 (0.000 s)
    test07 (0.000 s)
```

Figure 1: Screenshot of completed tests for Problem A.

Problem B

The second problem described a situation in which we needed to be able to determine the minimum number of devices that could fail in a network before we could see the path between two provided nodes be unable to communicate. The information we receive is a number of stations and the number of devices that can operate between each pairing of stations as well as the number of devices at each station itself. We should be able to use this information to determine the minimum number of devices that can cause an outage between two given stations.

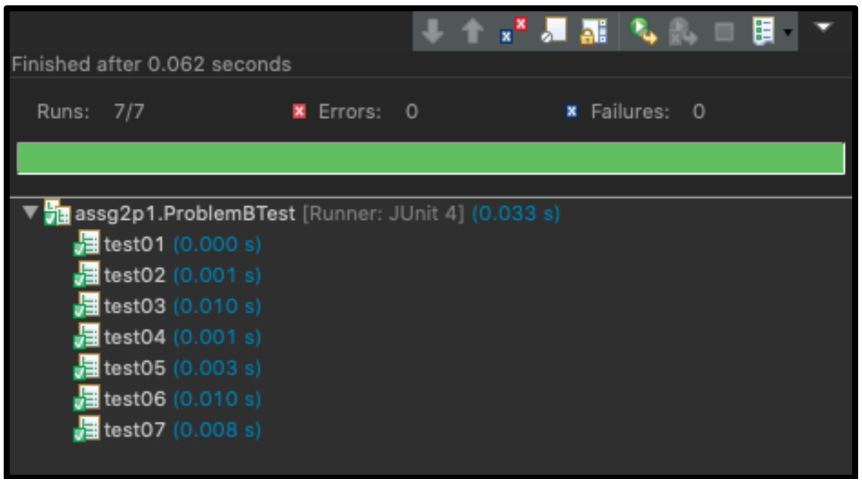
The first part of this problem was reading the input in. I set up some variables to track this information. The first four things we read in from the input are the starting station, end station, the number of vertices, and the number of edges. Therefore, I set these up in variables and read them in. The next information that we read in was about the stations and the number of devices that each held. I used a `HashMap` to hold this information. I then set up a graph that would take in all the information about each edge and contain the connections between stations. Considering the problem describes an undirected graph, the graph had to have connections going in both directions. Both directions had the same number of devices. To complete the matrix that I would use, I then iterated through the remaining pairs and added to the map any connections that weren't there. I set these to 0 as this represents they have no connection between them.

Upon studying the problem, I recognised that it is based off the *maximum flow problem*. Using this knowledge, I was able to use *GeekForGeeks*' implementation of Ford-Fulkerson's algorithm [2] as a starting point for this problem. This solution begins by setting up a residual graph that we will use to find the maximum flow in each path. This is set up by creating a copy of the graph that was set up when reading in the data. I used the `HashMap rGraph` to store this. Next, I needed to set up a way of tracking the parent of each node that has been found to have a path. This is stored in a `HashMap` that has a string representing a station and is mapped to the 'parent' node upon finding a path from start to end. To initialise, I simply placed all the station names in there and mapped them to an empty string.

The next part of this algorithm continues to iterate while there is a path from the start to the end stations. It should be noted that this part required the creation of a simple *BFS* helper method. I will not describe its functionality in detail as it was only a standard *BFS* implementation that returned true if the residual graph had a path from start to end. However, it should be noted that the *BFS* modified the `parent` map when finding a path.

We then iterate through until there are no more paths. On each iteration, the loop uses the parents to get the minimum path flow backwards, updating the residual graph with this minimum flow once it has discovered. The idea is that while there is some path between the two stations, we must also have more flow that we can pass through the network. Once this has completed, we have a maximum flow between the start and end nodes. So, if we were only considering the edges we would know the minimum number of devices that would cause an outage between two provided stations.

To consider the nodes, I used the minimum-cut problem that extends the Ford-Fulkerson algorithm for inspiration [3]. This utilises a *DFS* helper method to find all the nodes that have been visited in the graph. Then, we iterate over each pair and if the first node has been visited, the second one has not and the edge between them exists (i.e. is not 0), we use the minimum number of devices of the two edges, or the connecting edge. By then adding this to a running total, by the max flow in the graph.



```
Finished after 0.062 seconds

Runs:   7/7          Errors:   0          Failures:   0

▼ assg2p1.ProblemBTest [Runner: JUnit 4] (0.033 s)
    test01 (0.000 s)
    test02 (0.001 s)
    test03 (0.010 s)
    test04 (0.001 s)
    test05 (0.003 s)
    test06 (0.010 s)
    test07 (0.008 s)
```

Figure 2: Screenshot showing problem B tests passing.

After completing the implementation, I was able to run it against the test cases. As seen in *figure 2*, all tests pass in 0.033s. This result can vary due to variance but never exceeds 0.05s.

References

[1] "Floyd Warshall Algorithm | DP-16", GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/. [Accessed: 06- Nov- 2019].

[2] "Ford-Fulkerson Algorithm for Maximum Flow Problem", GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/. [Accessed: 06- Nov- 2019].

[3] "Find minimum s-t cut in a flow network", GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/minimum-cut-in-a-directed-graph/. [Accessed: 06- Nov- 2019].