Assignment 1 – Stage 2

Assignment 1 focuses on the implementation and manipulation of a train network. Using data read in from a CSV file, we have been tasked with completing a number of methods that will enable us to analyse and understand the data provided better. Furthermore, the focus of this assignment is on our ability to understand and write graph algorithms.

Stage 2 focused on a situation described to us where we had to find the minimum cost of an exhaustive search on the given list of stations. After completing the method recursively, I was able to quickly learn the importance of using a dynamic programming approach.

While I was only able to complete the first method, I believe that I was able to get close to completing the second method. I have described my attempted implementation in its appropriate section.

optimalScanCost

In my first attempt, I completed a recursive function that found the value. For reference, it is below. This first solution built off the 'rod cutting' exercise completed in class. The idea of the function is to break up the list into multiple parts, and then calculate the value of each subpart. We can then return all the minimum values of these subparts to calculate what the most optimal exhaustive search count would be given the function guidelines.

This first solution uses a loop to split the given list in different parts. It then uses these two sub-lists to create two recursive calls – one for each list. Throughout this function, a variable is tracking the minimum value for the path. It is initially set to the maximum value `Integer` can hold. Then, within each iteration, we compare it to the total distance of the current route plus the two recursive calls. It then checks to see what is smaller and assigns accordingly. This solution, however, completes far too many recursive calls and when using a large set of stations, can take a very long time to return and complete.

In order to combat this, I implemented the algorithm using a dynamic programming approach. This meant the inclusion of a look up table that would stop us from recalculating recursive calls. With this, instead of going ahead and blindly calling the recursive function, I check if a value exists in the lookup table that matches the recursive call that I was about to make. If there is one, we know that this has already been calculated and we can just use the value from the lookup table instead of recalculating it. If there is no value, however, we complete the recursive call and place the path and the corresponding value into the look up table so that we can use it in future iterations/recursive calls.

optimalScanSolution

While I did not complete this method, I believe that I was on the right track and was getting close. My implementation was focused around the use of a HashMap to store the previous point to that path, or the 'split' node. Each time a recursive call was made, and a decision was found, we could set this value to be the 'split' node and have it mapped to the distance there. This was we can continually track the best value to use.

However, I was unable to get this working and was returning too many values to the HashMap.

Code

```
public int optimalScanCost(ArrayList<String> route) {
    if (route == null || route.size() <= 2) {
        return 0;
    }

    int min_val = Integer.MAX_VALUE;

    for (int i = 1; i < route.size() - 1; i++) {
        ArrayList<String> temp1 = new ArrayList<>(route.subList(0, i+1));
        ArrayList<String> temp2 = new ArrayList<>(route.subList(i, route.size()));
        min_val = Math.min(min_val, findTotalDistance(route) + optimalScanCost(temp
1) + optimalScanCost(temp2));
    }

    return min_val;
}
```