IMPORTANT NOTE (PLEASE READ):

Much like in US13, for the purposes of simplifying and streamlining the development of this US, our team (G123) decided to create a custom Class, an abstract data type, that we use throughout the project to store the information about the shortest paths from any vertice to the "origin vertice" (the vertice to which all shortest paths are calculated, i.e. the Assembly Point". This class was given the name "ShortestRoutes" and its source code is as follows:

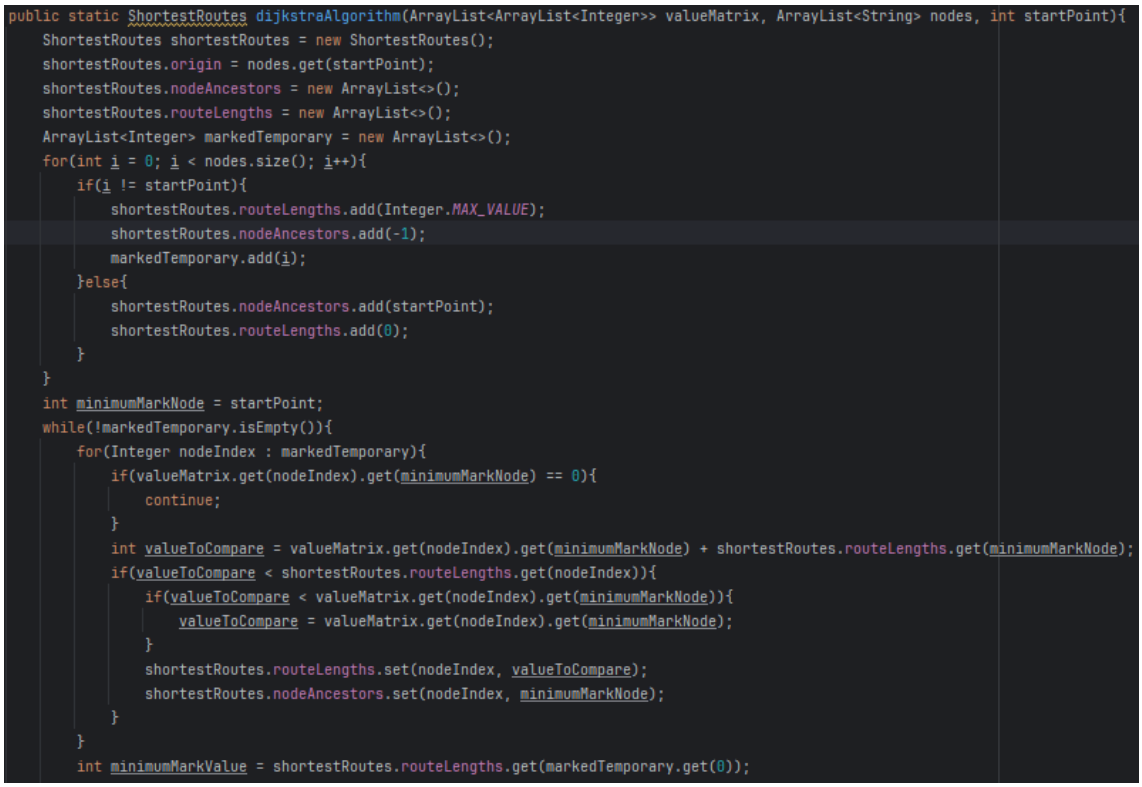
```
final class ShortestRoutes {  11 usages
    public String origin;  3 usages
    public ArrayList<Integer> nodeAncestors;  19 usages
    public ArrayList<Integer> routeLengths;  18 usages
}
```

This class is entirely just a set of variables, in particular the nodeAncestors and routeLengths lists, and should in no way be thought of as an element with any real functionality.

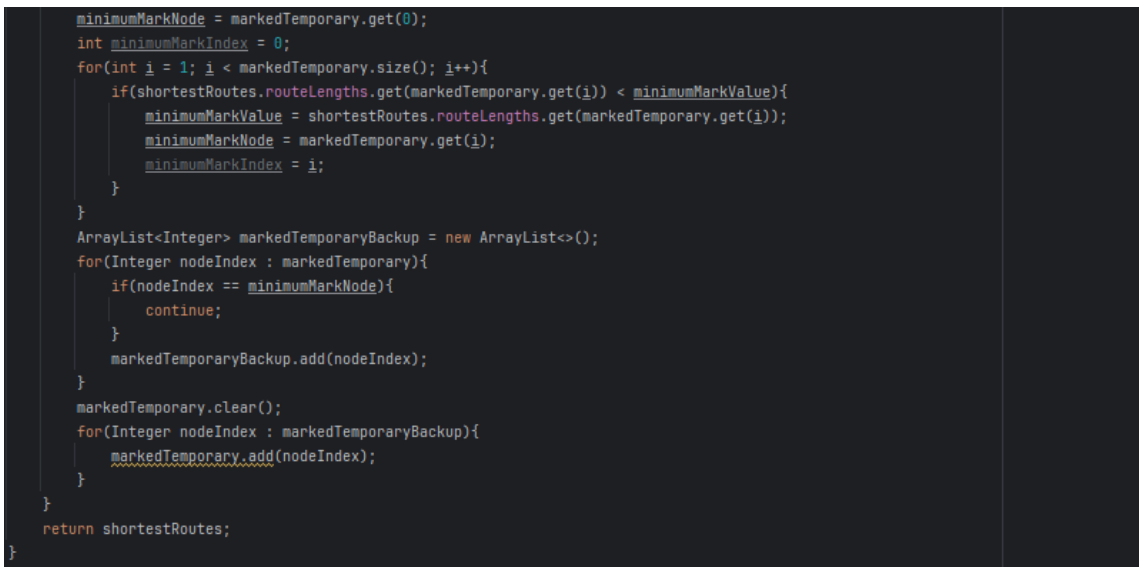
Additionally, this class's "origin" variable, though interacted with at various points in the program, is never interacted with in a way that affects the time complexity beyond any other explained factor. So, in the pseudocode interpretations present in US19, all mentions to this "origin" variable are completely removed. This is irrelevant and should not cause confusion to the reader.

With that in mind, we proceed to showcasing the procedures put in place to implement the algorithms for this US.

Below is our team's implementation of the Dijkstra Algorithm.



```
public static ShortestRoutes dijkstraAlgorithm(ArrayList<ArrayList<Integer>> valueMatrix, ArrayList<String> nodes, int startPoint){
    ShortestRoutes shortestRoutes = new ShortestRoutes();
    shortestRoutes.origin = nodes.get(startPoint);
    shortestRoutes.nodeAncestors = new ArrayList<>();
    shortestRoutes.routeLengths = new ArrayList<>();
    ArrayList<Integer> markedTemporary = new ArrayList<>();
    for(int i = 0; i < nodes.size(); i++){
        if(i != startPoint){
            shortestRoutes.routeLengths.add(Integer.MAX_VALUE);
            shortestRoutes.nodeAncestors.add(-1);
            markedTemporary.add(i);
        }else{
            shortestRoutes.nodeAncestors.add(startPoint);
            shortestRoutes.routeLengths.add(0);
        }
    }
    int minimumMarkNode = startPoint;
    while(!markedTemporary.isEmpty()){
        for(Integer nodeIndex : markedTemporary){
            if(valueMatrix.get(nodeIndex).get(minimumMarkNode) == 0){
                continue;
            }
            int valueToCompare = valueMatrix.get(nodeIndex).get(minimumMarkNode) + shortestRoutes.routeLengths.get(minimumMarkNode);
            if(valueToCompare < shortestRoutes.routeLengths.get(nodeIndex)){
                if(valueToCompare < valueMatrix.get(nodeIndex).get(minimumMarkNode)){
                    valueToCompare = valueMatrix.get(nodeIndex).get(minimumMarkNode);
                }
                shortestRoutes.routeLengths.set(nodeIndex, valueToCompare);
                shortestRoutes.nodeAncestors.set(nodeIndex, minimumMarkNode);
            }
        }
        int minimumMarkValue = shortestRoutes.routeLengths.get(markedTemporary.get(0));
```



```
        minimumMarkNode = markedTemporary.get(0);
        int minimumMarkIndex = 0;
        for(int i = 1; i < markedTemporary.size(); i++){
            if(shortestRoutes.routeLengths.get(markedTemporary.get(i)) < minimumMarkValue){
                minimumMarkValue = shortestRoutes.routeLengths.get(markedTemporary.get(i));
                minimumMarkNode = markedTemporary.get(i);
                minimumMarkIndex = i;
            }
        }
        ArrayList<Integer> markedTemporaryBackup = new ArrayList<>();
        for(Integer nodeIndex : markedTemporary){
            if(nodeIndex == minimumMarkNode){
                continue;
            }
            markedTemporaryBackup.add(nodeIndex);
        }
        markedTemporary.clear();
        for(Integer nodeIndex : markedTemporaryBackup){
            markedTemporary.add(nodeIndex);
        }
    }
    return shortestRoutes;
}
```

As you can see this algorithm requires 2 images to properly display in this document. However, that is the complete implementation of the algorithm.

It should be noted that, in the pseudocode interpretation in US19, the last two loops have been simplified to a simple "remove" operation: As explained there, this "remove" operation, whilst primitive in mathematical theory, is not primitive in Java: In order to avoid using non-primitive operations in our Java implementation, we emulated a remove operation as seen in the screen prints above.

It should also be noted that, as explained in US19, this algorithm makes use of the ShortestRoutes class defined earlier that the pseudocode avoids mention of.