Instituto Superior de Engenharia do Porto

23-01-2022

ESINF Report

LAPR3 INTEGRATIVE PROJECT

SPRINT 4

Group 96:

- ✓ Miguel Morais (1181032)
- ✓ Rodrigo Rodrigues (1191008)
- ✓ Tiago Ferreira (1200601)
- ✓ Mariana Lages (1200902)
- ✓ Eduardo Sousa (1200920)
- ✓ Miguel Jordão (1201487)

Índice

Class Diagram Analysis

For this sprint, we added new methods to be able to satisfy the ESINF User Stories. We will start by the PortCentrality class. Starting with the method getCentralityOfNPorts(), which shows the ports that have the greatest centrality. Their centrality is defined by the number of shortest paths that pass through them. The method sortMap() sorts a LinkedHashMap by descending order.

The class ShortPaths has two methods called seaPath() and landPath(), one calculating the shortest path between two locals by sea and the other calculates the shortest path between two locals by land. The class also has a method called calculateBestPath(), which calculates the best path between two points inputted by the user. Finally, there is the permutation method which permutates a list of Vertex.

Finally, there is the MostEfficientCircuit class, which has the efficientCircuit, that calculates the best circuit by having a starting point city. The backTrackCircuit does a backtrack to the starting point, without repeating the same cities that the circuit visited before.

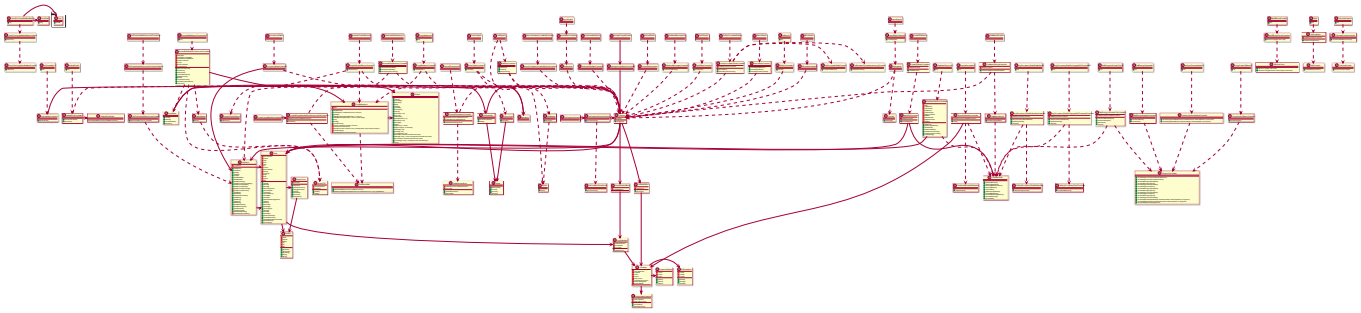

Figure 1 - Geral Class Diagram

US401 Complexity Analysis

The getCentralityOfNPorts method requires a previously built graph and a "N" number, which is introduced by the user. This number, theoretically, can't be less than 0, but in case that happens, the method complexity is $\boldsymbol{O}(\mathbf{1})$ since it returns a String warning that the number can't be negative and must be above 0. When the number introduced is valid, the method will traverse all the vertices of the graph and calculate the shortest paths of each one of them, using the shortestPaths method implemented in the Algorithms class. Then, it will traverse each of the paths and each vertex of each path and, if the vertex is a port, it is added to a map. After doing this procedure for all paths and vertices, we sort the map in descending order of value, making sure that the ports with the greatest centrality are in the top and the ones with the lowest centrality are in the bottom. Then, it prints the "N" more critical ports in the console. The complexity of the shortestPaths method is $\boldsymbol{O}(\boldsymbol{V^2})$ and, since it is called inside a "for" cycle, we can conclude that the final complexity is $\boldsymbol{O}(\boldsymbol{V^3})$.

US402 Complexity Analysis

The methods of the ShortPaths class require a previously built graph. The class starts with the methods SeaPath() and LandPath(). One calculates the shortest path between two locals by sea and the other calculates the shortest path between two locals by land. Both methods are $\boldsymbol{O}(\boldsymbol{V^2})$.

The calculateBestPath() method calculates the best path between two paths and has a list of vertices. If the list size is 1, the method will have a complexity of $\boldsymbol{O}(\boldsymbol{V^2})$. In this case, the method will call the shortestPath method to make its comparison. If the list size is bigger than 1, its complexity is $\boldsymbol{O}(\boldsymbol{n}!)$. The method will declare two list of vertices, one being the pathTotal and the other being the saveTotalPath, which will be used to save the list pathTotal. It will run a set of LinkedList<Vertex>, adding the vertex of those list to the pathTotal. When the operation ends, the pathTotal will be saved to the saveTotalPath and it will reset its value for the next "for each" cycle.

There is a method called permutation that does permutations on a list of vertices. This method has a complexity of $\boldsymbol{O}(\boldsymbol{n}!)$. It uses the recursion for its permutations.

We are aware that this wasn't the best approach to this problem. Our approach is greedy since we calculate all the shortest paths using all permutations. A better solution was to use the Floyd-Warshall algorithm with an auxiliar matrix that saves the intermediate vertices between the origin and destination vertices. Then, we could reconstruct the best path from the beginning to the end. However, we couldn't make it work since the lack of time we had. So, we opted to use a worst solution to this problem, making sure we could complete this User Story in time.

US403 Complexity Analysis

The efficientCircuit() method requires a previously built graph and a "N" number, which is introduced by the user. This number, theoretically, can't be less than 0, but in case that happens, the method complexity is $\boldsymbol{O}(\boldsymbol{1})$ since it is returning a null list. The method also requires a city input, which will be used as the circuit starting point. The method has a list that will add the cities that the circuit will have. The method starts by calculating the distance of the adjacent vertices of the starting point and it will compare their distances. When it finds the vertex with the shortest distance, the method will use recursion for this distance analysis, but this time using the vertex that had the shortest distance. This recursion will end when the vertex has a degree less or equals to 0. This method is $\boldsymbol{O}(\boldsymbol{V})$ if this recursion happens. When the recursion ends, the method will call the backtrack method: backTrackCircuit().

The backTrackCircuit() is a $\boldsymbol{O}(\boldsymbol{V})$ method as well since it is similar to the efficientCircuit(), but it has a list that checks if the city was already visited by the circuit or not. Each time the distance is compared, the cities will be added to that list, so they won't be added to the next recursions. This method is also $\boldsymbol{O}(\boldsymbol{V})$.

Contributions

- US401
  - Code: Mariana Lages (1200902) and Miguel Jordão (1201487)
  - Tests: Mariana Lages (1200902) and Miguel Jordão (1201487)

    - Complexity Analysis: Mariana Lages (1200902) and Miguel Jordão (1201487)

- US402
    - Code: Eduardo Sousa (1200920) and Miguel Jordão (1201487)
    - Tests: Eduardo Sousa (1200920) and Miguel Jordão (1201487)
    - Complexity Analysis: Eduardo Sousa (1200920) and Miguel Jordão (1201487)

- US403
    - Code: Tiago Ferreira (1200601)
    - Tests: Tiago Ferreira (1200601)
    - Complexity Analysis: Tiago Ferreira (1200601)