PFL - TP1

Group T11_G10

Afonso Castro @up202208026
Pedro Santos @up202205900

Index

Introduction

This project was developed by Afonso Castro (up202208026) and Pedro Santos (up202205900). Both group members worked in all the required tasks together, meaning their participation in the project is 50% each.

shortestPath

The shortestPath function has the goal of finding all the shortest paths between two specified cities in a given roadmap.

Here, we decided to use a breadth-first search (BFS) algorithm, which we found appropriate as it explores all paths level by level.

Another reason for this choice of algorithm is related to the fact that it is a clear algorithm easy to understand when compared to other solutions, such as the Dijkstra algorithm.

Algorithm Description Overview

The function receives a RoadMap, and the two cities the user wishes to know the shortest path between, which will be presented in the output as a list of paths (as there might exist more than one possible result).

Firstly, if the two cities in the input are the same, the function returns only one path, with the single city. If they are different, we start a BFS with the starting city that adds a new “step” in each existing path, without ever repeating cities in a path (loops would just add to the total distance of the path, without achieving anything). After extending all the existing paths, we separate them into two lists: allValidPaths and remainingPaths.

A path is transferred to allValidPaths if it already reaches the final destination, otherwise it is kept in the remainingPaths. If there are no more paths in remainingPaths, we check allValidPaths and return the shortest path(s), otherwise we extend remainingPaths again in a new iteration.

Data Structures

In the development of this function we used lists for various types of purposes:

- **Paths List** - Being the main data structure used, this list of paths (where each path is also a list of cities), allows for easy extension and exploration of the many different routes.
- **Filtering Lists** - The use of the list allValidPaths (paths that reach the destination city) and the list remainingPaths (paths that need further exploration), helped us keep the code clear and straightforward, allowing us to access the path data without complex data structures.
- **Distances** - This list, as the name suggests, helps us store distances calculated for each path, which is obviously important when filtering paths based on their total distances.

travelSales

The travelSales function has the goal of finding (one of) the shortest paths that visit every city in a given roadmap returning to the starting city at the end.

Here, we decided to use a very similar algorithm to the shortestPath function (BFS), with a few changes.

Like it was said before, the BFS implementation is straightforward and easier to understand when compared to other algorithms, like backtracking or dynamic programming, which could introduce unnecessary complexity.

Algorithm Description Overview

The function receives a RoadMap, if this roadmap is not strongly connected the function returns an empty list as it is impossible to visit all cities in that situation.

After this initial check we start a BFS to create paths, similarly to what we did in the shortestPath algorithm. In this function however we extend each path until it is a “completeCycle”, meaning that the paths must have the same length as the amount of cities in the RoadMap + 1 (the starting city is repeated at the end), while also having the first and last cities be the same. After all the paths are completeCycles, we once again filter them by the minimum distance, and arbitrarily return one of them (the first of the list in our case)

Data Structures

In the development of this function we used lists for various types of purposes:

- **Paths List** - Being the main data structure used (just like in the shortestPath function), this list of paths (where each path is also a list of cities), allows for easy extension and exploration of the many different routes.
- **Filtering Lists** - The use of the list completeCycles (filtered paths that form a complete TSP cycle) and the list incompletePaths (paths that didn't yet completed a cycle at that moment), helped us organize the code, keeping it clear and straightforward, allowing us to better manage the paths in the BFS.
- **Distances** - This list, as the name suggests once again, helps us store distances calculated for each path, which is obviously important when filtering paths based on their total distances.