# TP1 for PFL

**T11\_G11**

## Group Members

* Member 1: Leandro Martins up202208001@up.pt - 50%
* Member 2: Simão Neri up202206370@up.pt - 50%

## Contribution Overview

### Leandro Martins

* + Implemented functions *2*, *4*, *6*, and *8*:
    - *areAdjacent*: Checks if two cities are directly connected
    - *adjacent*: Returns cities adjacent to a given city along with their distances
    - *rome*: Identifies cities with the highest number of connecting roads
    - *shortestPath*: Computes all shortest paths between two cities, handling cases with multiple equal-distance paths.
  + Conducted testing for these functions to ensure accuracy and efficiency.
  + Created and wrote the README file for the project

### Simão Neri

* + Implemented functions *1*, *3*, *5*, *7*, and *9*
    - *cities*: Returns a list of all cities in the roadmap
    - *distance*: Finds the direct distance between two cities, connected
    - *pathDistance*: Computes the total distance of a path, returning `*Nothing*` if any segments aren't directly connected
    - *isStronglyConnected*: Checks if all cities in the roadmap are reachable from each other
    - *travelSales*: Provides a solution for the Traveling Salesman Problem (TSP) if a valid path exists.
  + Implemented functions to convert *RoadMap* into an *Adjacent List*.
* **Both**
  + Tested each other's functions and looked for mistakes.
  + Reviewed the README file.

## Implementation of the *shortestPath* Function

### Key Components of the Implementation:

1. RoadMap Representation:

* The RoadMap is represented as a list of tuples, where each tuple contains two cities and the distance between them.

1. Adjacency List Conversion:

* The function first converts the RoadMap into an adjacency list format. This structure enables efficient look-up of neighboring cities and their distances.

1. Dijkstra's Algorithm

* The algorithm uses a priority queue to explore the shortest paths incrementally. It maintains a list of visited cities and the current path taken to reach each city.
* If the current city is the destination city, the function checks whether the current path's distance is less than or equal to the previously recorded minimum distance. If so, it stores the current path.

1. Path Storage and Management:

* The paths are stored in a results list, which is populated with all the shortest paths found during the search.

1. Return Value:

* The function returns a list of paths, each represented as a list of cities, which represent the shortest routes from the starting city to the ending city.

### Justification of Data Structures

* Adjacency List:
  + The adjacency list is chosen for its efficiency in storing sparse graphs, allowing for quick access to neighboring cities. This is crucial for reducing the time complexity of the algorithm.
* Priority Queue:
  + A priority queue is used to ensure that the algorithm always explores the shortest available path next. This structure supports efficient insertion and extraction of the minimum element, leading to improved performance.

### Complexity Analysis

The time complexity of the *shortestPath* function is O((V + E) log V), where V is the number of vertices (cities) and E is the number of edges (roads). The logarithmic factor is due to the operations performed on the priority queue during the path exploration.

## Implementation of the *travelSales* Function

### Key Components of the Implementation:

1. Component 1:
   * Info 1
2. Component 2:
   * Info 2
3. Component 3:
   * Info 3

### Justification of Data Structures

* Structure 1:
  + Info 1
* Structure 2:
  + Info 2

### Complexity Analysis

The time complexity is something.