TP1: Haskell Coursework

> Functional and Logic Programming (L.EIC024) 2024/2025
> Bachelor in Informatics and Computing Engineering
> António Mário da Silva Marcos Florido (Regent and practical classes teacher)

Group T02_G08

- Guilherme Duarte Silva Matos (up202208755@up.pt)
- João Vítor da Costa Ferreira (up202208393@up.pt)

Group Contributions

Member	%	Task Assignment
Guilherme Matos	45%	Warning fixing relating to the usage of `head`, streamlining the code, auxiliary functions, `cities`, `rome`, `isStronglyConnected`, `travelSales`
João Ferreira	55%	`adjacent`, `areAdjacent`, `distance`, `pathDistance`, `dijkstra`, priority queue implementation, project testing

`shortestPath` Implementation

The `shortestPath` function was implemented using the **Dijkstra's** algorithm to find the shortest path between two cities in a roadmap. Below is the pseudocode of the algorithm used in the implementation:

```
shortestPath(roadmap, city1, city2):
    Mark all nodes as unvisited
    Initialize all the distances to infinity
    Set the distance to the starting node to 0
    Create a priority queue `q` and insert all the nodes with
their distances
    While `q` is not empty:
        Pop the node `u` with the smallest distance from `q`
        Mark `u` as visited
        For each unvisited neighbor `v` of `u`:
            relax(roadmap, u, v)

relax(roadmap, u, v):
    distU = distance from the starting node to `u`
    distV = distance from the starting node to `v`
    weight = distance between `u` and `v`
    If distV >= distU + weight:
        Set the distance of `v` to distU + weight
        Set the previous node of `v` to `u`
```

The implementation uses two auxiliary data structures:

- A priority queue to store the node yet to be processed, ordered by the distance from the starting node. Implemented using a list of tuples with linear time search;

  > It is worth mentioning that the priority queue could be implemented using a binary heap with logarithmic access times to improve the time complexity of the algorithm, at the cost of simplicity.

- A list (with linear access time) of auxiliary variables for each city to store:
  - The distance from the starting node;
  - A list of the previous nodes to reconstruct the path;
  - If the node was already processed.

`travelSales` Implementation

This project uses Dynamic Programming for the TSP. Dynamic programming allows for the reduction of the number of calculations by caching the answer to a substet to the problem. This property allows for the reduction of work in comparison to a brute-force implementation, positively affecting the time complexity of the function.

Mathematically, it can be defined as such:

$$c_{i,\varnothing} = w_{i,n} \quad i \neq n$$

$$c_{i,S} = min_{j \in S}[w_{i,j} + c_{j,Sj}] \quad i \neq n, i \notin S$$

A short overview of the relevant functions to the implementation:

```
    table(roadMap, lastIndex, mask, path):
        This function represents the dynamic programming table
of two dimensions, of size (n-1, 2^n -1), leveraging Haskell's
memoization of function calls.
        This function is the tool that calculates the cost and
path of the TSP problem.

    travelSales(roadMap):
        This function calls table, solving the TSP. It uses the
roadMap to calculate all the other arguments for the table
function.
```