# PFL - Haskell Coursework

Licenciatura em Engenharia Informática e Computação

October 2024

## 1 Work Groups

Groups of **two (2) students** enrolled in the **same practical class**. Exceptionally, and only if necessary, a group of three per class may be accepted. The groups are chosen in the activity available for this purpose on Moodle. Groups are already created, named *Txx_Gyy*, where *Txx* represents the practical class (from T01 to T16), and *Gyy* represents the group within the class (from group G01 to G14). Groups from each practical class should organize themselves into these groups. **Registration is mandatory for all students taking the assignment.** Unregistered students will have a grade of 0.

## 2 Evaluation and Deadlines

This assignment counts for 25% of the final grade. The evaluation focuses on implemented features, the quality and efficiency of the code and respective comments, the readme file, and participation in the assignment and presentation. Grades may differ between group members.

The work must be submitted by **20:00 of the 3rd of November**, 2024, in the activity to be made available for this purpose on Moodle, with demonstrations carried out during the week of November 4th, 2024. **The demonstrations should be scheduled with the teacher of each practical class.**

## 3 Submission guidelines

Each group must deliver a **.zip file with exactly two files**: a .hs file with the whole source code, called **TP1.hs**, and a .pdf file with a brief README of the project, called **README.pdf**. Both the source code file and the README.pdf file must be at the archive root level. The archive must not contain any directories or other files.

The submitted file must be zipped as a .zip file (not another archive format). The file name must be:

**PFL_TP1_Txx_Gyy.zip**

where Txx_Gyy is the name of the group (example: PFL_TP1_T06_G02.zip).

### 3.1 Source code file

Assignments will be **subjected to an automated battery of tests**, so you **must adhere to the stated file and function naming conventions**. The inability to test the developed code will result in penalties in the evaluation.

The following rules must be followed carefully:

- The implemented assignment must run under GHC, version 9.10.1.
- All the work must be developed in a **single** file named TP1.hs. If you want to separate your code, do so with comments.
- The work must be developed using the **starter TP1.hs file** provided in the course's Moodle page. **The datatype definitions and function signatures of this file must not be modified.**
- All of the functions are initially defined as `undefined` (which is a reseved keyword in Haskell). If you do not implement a version of a function asked in the assignment, leave it as is (i.e. as `undefined`).
- All code must be **properly commented**: each function should include the **type declaration** (signature) as well a brief description of the **function's goal** and the **meaning of the arguments**.
- You may import the following three GHC modules: `Data.List`, `Data.Array` and `Data.Bits`. You may import all, some or none of them. The import statements must be the first lines of the code (i.e. do not include lines with comments before the import statements). Each import statement, which must be in exactly a single line of code, must have exactly the following format: `import qualified <name of the module>`. For instance, if a group wants to use all three modules, the first 3 lines of their source code file are exactly (in no particular order):

```
import qualified Data.List
import qualified Data.Array
import qualified Data.Bits
```

### 3.2 README file

The README file should contain the following sections:

- Identification of the group members, contribution of each member (in percentages adding up to 100%) and a brief description the tasks each one performed.
- Explanation of how the `shortestPath` function was implemented, including a justification of why certain auxiliary data structures were selected and used, and a description of the algorithm(s) used.
- A similar section to the previous one for the `travelSales` function.

# 4 Assignment description

The goals of this project are to define and use appropriate data types for a graph representing a country, composed of a set of interconnected cities.

## 4.1 Types

The type for graphs used as input of all the functions to be implemented is:

```
type RoadMap = [(City,City,Distance)]
```

which must use the following auxiliary types:

```
type City = String
type Distance = Int
type Path = [City]
```

According to these definitions, the vertices of the graph are the cities and each edge is a tuple containing the name of the cities it connects and the distance of the road connecting them. Assume that all distances are represented as integer numbers. Consider that the graph is **undirected**.

## 4.2 Functions

Implement the following operations needed to manipulate roadmaps. Unless we explicitly ask for a particular order, in functions where the output is a list, any order of its elements will be accepted.

1. `cities :: RoadMap -> [City]`, returns all the cities in the graph.
2. `areAdjacent :: RoadMap -> City -> City -> Bool`, returns a boolean indicating whether two cities are linked directly.
3. `distance :: RoadMap -> City -> City -> Maybe Distance`, returns a `Just value` with the distance between two cities connected directly, given two city names, and `Nothing` otherwise.
4. `adjacent :: RoadMap -> City -> [(City,Distance)]`, returns the cities adjacent to a particular city (i.e. cities with a direct edge between them) and the respective distances to them.
5. `pathDistance :: RoadMap -> Path -> Maybe Distance`, returns the sum of all individual distances in a path between two cities in a `Just` value, if all the consecutive pairs of cities are directly connected by roads. Otherwise, it returns a `Nothing`.
6. `rome :: RoadMap -> [City]`, returns the names of the cities with the highest number of roads connecting to them (i.e. the vertices with the highest degree).

7. `isStronglyConnected :: RoadMap -> Bool`, returns a boolean indicating whether all the cities in the graph are connected in the roadmap (i.e., if every city is reachable from every other city).

8. `shortestPath :: RoadMap -> City -> City -> [Path]`, computes all shortest paths [RL99, BG20] connecting the two cities given as input. Note that there may be more than one path with the same total distance. If there are no paths between the input cities, then return an empty list. Note that the (only) shortest path between a city `c` and itself is `[c]`.

9. `travelSales :: RoadMap -> Path`, given a roadmap, returns a solution of the Traveling Salesman Problem (TSP). In this problem, a traveling salesperson has to visit each city exactly once and come back to the starting town. The problem is to find the shortest route, that is, the route whose total distance is minimum. This problem has a known solution using dynamic programming [RL99]. Any optimal TSP path will be accepted and the function only needs to return one of them, so the starting city (which is also the ending city) is left to be chosen by each group. Note that the roadmap might not be a complete graph (i.e. a graph where all vertices are connected to all other vertices). If the graph does not have a TSP path, then return an empty list.

10. **This item is done ONLY by groups of three students** The *brute force solution* to the Traveling Salesman Problem (TSP) generates all possible city permutations and calculates each route's length. The shortest route is then returned as the solution to the TSP. The *dynamic programming solution* breaks down the problem into smaller subproblems and stores the results of these subproblems to avoid redundant calculations.

    Define two functions:

    (a) `travelSales :: RoadMap -> Path`

    (b) `tspBruteForce :: RoadMap -> Path`

    solving the TSP using dynamic programming and brute force, respectively. Compare both versions in terms of efficiency and time complexity. Write this comparison in a new section of the README file.

The time complexity of the algorithms will be taken into account in the coursework grades. Therefore, to achieve more efficient solutions, especially in the last two functions, students are advised to use the data structures from the GHC modules that can be imported and to convert the graph to a more convenient representation. Any other data structure needed to improve the code's efficiency (such as a heap) must be implemented by the students. Some example of graph representations include:

1. Adjacency list representation

```
type AdjList = [(City,[(City,Distance)]]
```

2. Adjacency matrix representation

```
type AdjMatrix = Data.Array.Array (Int,Int) (Maybe Distance)
```
[1]

3. Pointer representation

```
data AdjPointers = Place City  [(AdjPointers, Distance)]
```

The pointer representation is very space efficient but its operations are a bit more complicated and time inefficient.

When comparing between the adjacency list and the adjacency matrix representations, the efficiencies of their manipulating functions depend on the number of cities and roads. Informally, when there is a large number of roads, the map is said to be dense, and when there are few roads, it is sparse. In general, the matrix representation is better with dense maps and the adjacency list representation with sparse maps.

# References


[BG20] Richard Bird and Jeremy Gibbons. *Algorithm Design with Haskell*. Cambridge University Press, 2020.

[RL99] Fethi Rabhi and Guy Lapalme. *Algorithms: a functional programming approach*. Addison-Wesley, 2 edition, 1999.


[1] In Haskell, arrays are not part of the Standard Prelude but provided as the Array library module, so before using any array-related function, this library should be imported using the directive **import Data.Array**. The type of an array is denoted as **Array a b** where **a** is the type of the index and **b** is the type of the value.