PFL Project 1

Class 11 Group 8

L.EIC 24/25

Leonor Couto up202205796@fe.up.pt

Rodrigo de Sousa up202205751@fe.up.pt

Introduction

This project focuses on the design and implementation of a graph-based model representing a network of roads , which consists of interconnected cities. By utilising Haskell's functional programming capabilities, we developed a robust framework to manipulate and analyse roadmaps, enabling various operations such as determining city connectivity, calculating distances, and finding optimal paths for a travelling salesperson.

Our graph model, defined as undirected, allows for the representation of bidirectional roads, with tuples denoting connections and distances for clarity. Additionally, we tackled complex challenges like computing shortest paths and solving the Travelling Salesman Problem (TSP).

The functions were done by both students Leonor and Rodrigo, mostly done simultaneously but also by delegating some tasks in auxiliary functions. That being said, the weight of the contributions by Leonor is 50% and 50% by Rodrigo.

shortestPath

In this function we had to implement a way to get all possible paths between two given cities that would wield the shortest distance. We implemented a breadth-first search algorithm to help with that.

Algorithm Description

The first part of our algorithm is to find all possible paths and to do so we used a BFS algorithm:

```
bfsPaths :: RoadMap -> City -> City -> [Path]
```

This auxiliary function receives a RoadMap and two cities, returning a list of the paths that go from the first city to the second. It works as a normal BFS where it stores in a queue the next cities that are adjacent to the one that we are currently "visiting". It uses the previously created `adjacent` function and adds the cities to the queue. It iterates through the whole queue marking cities as visited until the current city is the destination, then it adds the whole path that had been visited to the list of paths and tries again until we have a list of all paths.

After getting all the paths, we then calculate the distance for all paths with the previously defined function `pathDistance` to get the distance for all paths and with that get the minimum distance. Then we apply a filter in the list where all paths with distances bigger than the minimum are removed and we end up with the list of all shortest paths possible between two cities.

Data Structures Used

This function leverages a combination of lists and tuples to effectively represent and manage both the road map and the paths between cities. The road map is organised as a list of tuples, with each tuple containing two cities and their associated distance. This structure facilitates efficient lookups and traversal of connections.

To explore all possible paths from the starting city to the destination city, the function employs a breadth-first search (BFS) strategy, storing paths as lists of cities. Each path's validity is assessed by calculating its distance through the `pathDistance` function, which utilises the Maybe type to gracefully handle cases where a path may be invalid, specifically when cities within the path are not directly connected.

This design enables the function to identify all valid paths and filter them to find the shortest ones.

travelSales

In this function we made a solution for the Travelling Salesman Problem (TSP) for the roadmap given. The goal is to find the shortest possible route that allows a salesman to visit each city exactly once and return to the starting city. In this implementation, we again used a breadth-first search (BFS) approach to explore potential paths through a given roadmap represented as a list of cities and their distances, with this function:

```
bfsTravel :: [Path] -> [Path]
```

The algorithm ensures that only valid paths are considered by verifying that all cities are reachable from the starting point. In the beginning of the function we verify that the road map is strongly connected, because if it isn't there can’t be a path going through all the cities.

Algorithm Description

To identify the optimal route, the algorithm generates paths incrementally. It examines each city’s neighbours, verifying that the paths remain valid throughout the exploration process, with the `cityNeighbors` auxiliary function. Upon forming a complete path that visits all cities once and returns to the origin, the algorithm calculates the total distance. This process continues until all potential paths are explored, ultimately selecting the shortest as the solution to the TSP, in the same way that we chose the shortest path in the previous section.

Data Structures Used

Data structures are utilised for efficient path traversal and connectivity checks. Adjacency lists facilitate quick access to neighbouring cities, while lists manage and extend current paths during the search for an optimal route. This combination supports the BFS methodology, ensuring a systematic exploration of all possible routes while maintaining operational clarity and efficiency.