PFL Project 1 - Haskell

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, by implementing some operations needed to manipulate roadmaps.

This project was collaboratively developed by **Francisco Afonso (up202208115)** and **Alexandre Ramos (up202208028)**, with an **equal** division of work **(50% / 50%)**.

Each of us took responsibility for implementing and thoroughly testing various functions while ensuring rigorous code review. We both validated each other's work to maintain accuracy and adherence to project requirements. Notably, almost **every function was developed together**, either **in person** or via the **Live Share** feature in Visual Studio Code, fostering a collaborative coding environment.

This collective effort allowed us to produce a solution that reflects a balanced contribution from both developers.

In this README file, we will also provide a walkthrough of the `shortestPath` and `travelSales` functions, explaining their implementation and functionality.

shortestPath

- This function calculates all possible shortest paths between two cities in a roadmap. It utilizes the Breadth-First Search (BFS) approach to efficiently explore and identify paths with the minimum distance, avoiding cycles.
- The overall time complexity is approximately **O(V * b^V)**, where V represents the number of cities, and b denotes the branching factor, which accounts for the exponential growth in paths. This complexity arises from the BFS approach, which explores all possible paths while tracking visited cities and calculating cumulative distances to find the optimal route.

Auxiliary Data Structures

- **Visited Paths List**: Keeps track of all shortest paths found to avoid duplicates.
- **Queue ([(Path, Distance)])**: A list of (Path, Distance) pairs for BFS traversal.

Key Components

1. **BFS Helper Function** (`bfs`)

   Core recursive function to perform **BFS** for finding all shortest paths from `start` to `finish`.

   - When `queue` is empty, it returns `paths`, representing all unique paths that reach `finish` with the shortest possible distance.
   - If the path's last city (`current`) matches `finish`, `bfs` evaluates whether it's the shortest path found:
     - If no shortest path has been set (`minDist = Nothing`), it adds this path and sets `minDist` to the current distance.
     - If a minimum distance exists (`minDist = Just m`) and matches the current path's distance, it adds the path to `paths`.
   - Otherwise, `bfs` extends the path by exploring neighboring cities not yet visited in the path, creating new potential paths to add to `queue` for further exploration.

   Justification: BFS ensures each shortest path is found before exploring longer paths. By checking minDist at each match, it efficiently returns all minimal paths without extra computations.

2. **Auxiliary Adjacent Function** (`adjacent`)

   Function that we used as helper to retrieve neighboring cities and their distances from the roadmap for a given city.

3. **Shortest Path(s) Main Function** (`shortestPath`)

   Initializes and calls the BFS helper function with the starting city, return a list of all possible shortest paths from `start` to `finish`.

   - If `start` is equal to `finish`, `shortestPath` immediately returns `[[start]]`, as the shortest path is simply the city itself.
   - Otherwise, it calls `bfs` with an initial `queue` containing the start city and a cumulative distance of `0`.

travelSales

- Our solution addresses the Traveling Salesman Problem (TSP) by calculating the shortest possible path that visits each city in a given roadmap once, returning to the starting city. It employs dynamic programming with bitwise operations and an adjacency matrix representation to minimize the travel distance.
- The algorithm has a time complexity of **O(V^2 + 2^V)**, where n is the number of cities, achieved by tracking visited cities via bit masking and recursively calculating distances to build the optimal path.
- The solution consists of several helper functions and a main function (travelSales) that orchestrates the TSP calculation. Each function has a specific role in building the adjacency matrix, managing the bitwise mask for visited cities, and computing the minimum distance path using recursion. Here's a detailed breakdown:

Auxiliary Data Structures

- **Adjacency Matrix**: Enables O(1) access to distances between cities for frequent distance lookups between pairs of cities.
- **Bit Masks**: Offers efficient memory usage and allows for quick checking and updating of visited cities, optimizing the recursive exploration of paths.
- **Visited** : An integer bitmask used to track which cities have been visited. Each bit position in the integer represents a city (e.g., if city 0 is visited, its corresponding bit in Visited will be set to 1). This enables efficient checking and marking of cities during pathfinding.
- **infinite**: A constant representing an infinitely large distance. Used in the adjacency matrix to signify unreachable or invalid routes between cities.

Key Components

1. **Adjacency Matrix Representation** (`tspMatrix`)

   Converts the roadmap to an adjacency matrix for efficient distance lookups. Each cell (i, j) holds the distance between cities i and j, and the matrix bounds are set based on the number of cities. Example matrix layout for gTest2.

	0	1	2	3
0	0	10	15	20
1	10	0	35	25
2	15	35	0	30
3	20	25	30	0

2. **Bit Masking for Tracking Visited Cities**

   The algorithm uses bit masks to track which cities have been visited, allowing for efficient set operations via bitwise manipulation.

   - `allCitiesVisited` generates a bitmask with all bits set to 1, representing the state where all cities are visited.
   - `isVisited` checks if a particular city has been visited by performing a bitwise AND (.&.) operation.
   - `visitCity` marks a city as visited in the bitmask using bitwise OR (.|.).

   Example:

   - `allCitiesVisited` for four cities (binary 1111) indicates all cities have been visited.
   - `isVisited` can confirm if a city (e.g., 0) is already marked in the bitmask, while `visitCity` updates the mask with a new visited city.

3. **Recursive Path Calculation** (`findShortestPath`)

   This function calculates the shortest path recursively, updating the bitmask as each city is visited.

   - **Base Case**: If all cities are visited, it checks for a return path to the starting city (0). If no path exists, it returns a high distance value to indicate an invalid route.
   - **Recursive Case**: It iterates through unvisited cities, calculates the distance to each, and recursively calls itself with the updated bitmask. It collects distances and paths from each call and selects the path with the shortest distance using minimum.

   Example:

   - Starting from city 0, the function explores all unvisited cities, backtracking as necessary, ensuring it finds the minimum path.

4. **Main TSP Solver** (`travelSales`)

   - Checks if the roadmap is strongly connected. If not, it returns an empty path.
   - Converts the roadmap into an adjacency matrix using `tspMatrix`.
   - Initializes the visited bitmask with only the starting city marked.
   - Calls `findShortestPath` with these initial values and appends the starting city ("0") to the resulting path to complete the tour.