PFL-project-1

Group Members:


```
Luis Fernandes     UP:202108770
Karolina Jedraszek UP:202402265
```


Distribution of work

The work was distributed equally (50/50).

1. cities :: RoadMap -> [City]
   Responsible: Luis
2. areAdjacent :: RoadMap -> City -> City -> Bool
   Responsible: Karolina
3. distance :: RoadMap -> City -> City -> Maybe Distance
   Responsible: Karolina
4. adjacent :: RoadMap -> City -> [(City,Distance)] Responsible: Karolina
5. pathDistance :: RoadMap -> Path -> Maybe Distance
   Responsible: Luis
6. rome :: RoadMap -> [City]
   Responsible: Karolina
7. isStronglyConnected :: RoadMap -> Bool
   Responsible: Luis
8. shortestPath :: RoadMap -> City -> City -> [Path]
   Responsible: Luis
9. travelSales :: RoadMap -> Path
   Responsible: Karolina

Process of work for shortestPath

So in order to do the shortestPath function we followed this procedure:
First we check if the roadmap is empty, if yes end the function by returning empty.
Then we get to the actual cases: We now check if the start and target cities provided are not the same, in the case they are we return [start].
On the case they are not we proced to call an helper function called findShortestPaths which takes a list of paths to start evaluating, a empty list where the result will be, a distance where we set to maxBound so when searching for the distance on the first case we dont accidently give a lower number than the case and lastly a new type that we created. This new type is: type AdjList = [(City,[(City,Distance)])], which basically creates a list of tuples where the first argument is

a city1 and the second one is a list of tuples but of cities and their respecitve distance to the city1.
Now this functions works using the BFS type to search for all the paths. It then starts creating the path and calculating the distance, once it reaches the target it then evaluates the distance with the value that was provided before when the function was called, this is only applied to the first path possible, with a distance that is.
Then depending it can go in 2 ways whenever the path has a distance different that the current shortestDist it creates a new list with this path and deletes the other, or simply skips the path entirely, when its lower and higher than the value respectively.
Once all paths are analysed we then return the list with all the paths that cost the lowest.

Process of work for travelSales

Idea: We will use dynamic programming with memoization to solve the traveling salesman problem.

First we are going to fill a memo array with results (values of the shortest path) of recurisve calls for the dp function. Following the formula:

`dp[end][mask] = minimum distance to reach the city 'end' by visiting the subset of cities represented by 'mask' (starting from the start city = 0).`

We will calculate `dp[i][j]` recursively using the formula:

`dp[end][S] = min(dp[end][S], dp[k][S − {end}] + distance[k][end]) for all k in subset S and k ≠ end.`

Of course, we need to consider the base cases:

`dp[end][0]` = minimum distance from the start city to reach the city end without visiting any other cities = distance from end city to the start city.

`dp[end][{end}]` = minimum distance from the start city to reach the city end by visiting only end = distance from end city to the start city.

To calculate the value of the shortest path, we find the minimum of `dp[end][mask] + minDirectDistance (intToCity i cities) (intToCity start cities)` for each end in the cities (except the starting city) where mask represents a subset containing all cities except the start city (lastSubset). We need to include the minDirectDistance here because it occurs twice in or path.

We use a memo array (that stores results for each end and mask from the dp function) to reduce time. Without this, we would call the dp function with the same arguments multiple times, wasting time.

To retrieve the path (findPath function), we first calculate the dp function and, based on its results: the value of the shortest path and the last vertex (end) of this path — we reconstruct the shortest path.

Similarly to the shortestPath function, we use a new data type, AdjList = [(City,[(City,Distance)])], to represent a roadmap. Although it is not as time-efficient as using matrices, it uses less memory.

Additional information: We use Int values to represent each city, so to get the minDirectDistance, we use the intToCity function to convert a city's number to its actual name from the roadmap. Moreover, because the graph is not weighted minDirectDistance i j = minDirectDistance j i.