PFL - Project 1

Project Members:

- Rui Cruz (up202208011): contribution - **100%** - the student carried out all the tasks assigned to him (***cities***, ***distance*** and ***pathDistance*** functions), contributing assiduously to the development of the last tasks and writing the final README.

- Tomás Teixeira (up202208041): contribution - **100%** - the student also successfully completed his tasks (***areAdjacent***, ***adjacent*** and ***rome*** functions), and also helped solve the last functions and write the README.

Shortest Path Problem

In this problem, we are tasked with finding all shortest paths between two cities in a given ***RoadMap***. We chose an approach that evaluates all possible paths between the start and end cities, selecting those with the minimum distance. This method allows us to capture multiple shortest paths, if they exist, as a purely greedy approach would not reliably do so.

Algorithm Description

The algorithm begins by checking if the starting city is the same as the ending city:

```
shortestPath xs start end
  | start == end = [[start]]
  | otherwise = searchShortestPaths [([start], 0)] []
```

If ***start*** and ***end*** are identical, the shortest path is simply **[start]**, as no traversal is necessary. This avoids unnecessary exploration. If ***start*** and ***end*** differ, the algorithm invokes ***searchShortestPaths*** to explore all paths between them. This function receives two lists:

- Unexplored paths ***(list)***: A list of paths yet to be completed, initialized with the starting city and a distance of 0. The function systematically explores and extends these paths.
- Paths from origin to destination ***(res)***: A list that stores completed paths that reach the target city along with their distances.

The structure of these lists is (Path, Distance) pairs, enabling each path to track its cumulative distance.

Base Case

The base case of ***searchShortestPaths*** triggers when there are no more unexplored paths (***list*** is empty), meaning all possible routes have been evaluated:

```
searchShortestPaths [] res = [p | (p, d) <- res, d == minDist]
  where minDist = if null res then 0 else minimum [d | (_, d) <- res]
```

At this point, the function filters ***res*** to return only the paths with the minimum distance. If ***res*** is empty, ***minDist*** is set to 0, returning an empty result.

Recursive Case

In the recursive case, the algorithm takes the first path in the unexplored list (***(path, dist):list***), where ***path*** represents the cities visited so far and ***dist*** is the cumulative distance:

```
searchShortestPaths ((path, dist):list) res
  | current == end = searchShortestPaths list ((path, dist) : res)
  | otherwise = searchShortestPaths newList res
  where
    current = last path
     neighbors = [(path ++ [adjacentCity], dist + d) | (adjacentCity, d) <- adjacent xs
current, adjacentCity `notElem` path]
    newList = Data.List.sortOn snd (list ++ neighbors)
```

1. **Check Destination:** If the last city in ***path*** (***current***) is the destination (***end***), the path is added to ***res***, and the function proceeds to the next path in ***list***.

2. **Expand Path:** If ***current*** is not the destination, the algorithm calls ***adjacent*** to find neighboring cities, extending ***path*** to each unvisited neighbor. Each new path, including its updated distance, is added to the unexplored paths (***newList***), sorted by distance. This sorting step prioritizes shorter paths but does not terminate the search when a path reaches ***end***, allowing all minimum-length paths to be identified.

Auxiliary Data Structures

To implement ***shortestPath***, we used two primary auxiliary data structures:

- ***res:*** Stores completed paths to the destination. This list ensures we capture all minimum-length paths rather than stopping after the first one found.

- ***newList:*** Holds paths to be explored, sorted by cumulative distance. Although this sorting doesn't make the algorithm strictly greedy, it allows the function to process shorter paths first, increasing efficiency.

These structures facilitate an exploration of paths in a way that captures all shortest paths, meeting the problem's requirements for thoroughness and completeness.

Traveling Salesman Problem

In this problem, we aim to find the shortest path that visits all cities exactly once in a given ***RoadMap*** and returns to the starting city. This function, ***travelSales***, implements a dynamic programming approach with bitmasking to efficiently solve the problem. If the roadmap is empty, disconnected, or does not allow a complete tour, the function returns an empty path.

Algorithm Description

The algorithm starts with basic checks to ensure it is possible to find a complete path:

```
travelSales xs
  | null allCities = []
  | not (isStronglyConnected xs) = []
  | dpArray Data.Array.! (initialMask, startPos) >= infinity = []
  | otherwise = constructPath initialMask startPos
```

Initial Checks

- If ***allCities*** is empty, there are no cities to visit, so an empty path is returned.

- If the roadmap is not strongly connected (***isStronglyConnected xs***), it's impossible to visit all cities from any starting point, so the function returns an empty path.

- The third check ensures that a complete path exists by verifying that the shortest computed tour is less than ***infinity***. If it's not, an empty path is returned.

Auxiliary Data Structures and Functions

The algorithm uses several auxiliary data structures and functions to manage city indexing, distances, and recursive calculations. These are essential to efficiently implement dynamic programming with bitmasking.

City Index Mapping:

- ***cityToIndex:*** Maps each city to a unique integer index. This is necessary for bitmasking, as each city can then be represented by a specific bit position.

- ***indexToCity:*** The inverse of ***cityToIndex***, mapping each index back to its corresponding city. This allows us to retrieve city names when reconstructing the path.

```
cityToIndex = zip allCities [0..]
indexToCity = zip [0..] allCities
```

toIdx and toCity: Helper functions that use the mappings above.

- ***toIdx:*** Converts a ***City*** to its corresponding index.
- ***toCity:*** Converts an index back to its corresponding ***City***.

```
toIdx :: City -> Int
toIdx city = case lookup city cityToIndex of
        Just idx -> idx
        Nothing -> 0

toCity :: Int -> City
toCity idx = case lookup idx indexToCity of
        Just city -> city
        Nothing -> ""
```

Distance Array (distArray):

- This 2D array holds the distances between each pair of cities, where ***distArray (i, j)*** represents the distance from city ***i*** to city ***j***. If two cities are not directly connected, the distance is set to ***infinity***.

- The ***distArray*** structure allows for quick lookups of distances without recalculating them each time.

```
distArray = Data.Array.array ((0, 0), (n-1, n-1))
  [((i, j), case distance xs (toCity i) (toCity j) of
        Just d -> d
        Nothing -> infinity)
  | i <- [0..n-1], j <- [0..n-1]]
```

Memoization Array (dpArray):

- ***dpArray*** is a 2D array that stores the minimum distance for subsets of cities, represented by bitmasks. This allows ***dp*** to store previously computed distances for subsets of cities, avoiding redundant calculations.

- Each entry ***dpArray (mask, pos)*** holds the minimum distance to complete the tour from a subset of cities (***mask***) ending at city ***pos***.

```
dpArray = Data.Array.array ((0, 0), ((2^n)-1, n-1))
   [((mask, pos), dp mask pos) | mask <- [0..(2^n)-1], pos <- [0..n-1]]
```

Recursive Distance Calculation

The ***dp*** function uses ***dpArray*** to compute the minimum distance required to visit all cities in a subset and end at a specified position:

```
dp :: Int -> Int -> Distance
dp mask pos
   | mask == (2^n)-1 = distArray Data.Array.! (pos, 0)
   | otherwise = minimum [if Data.Bits.testBit mask next then infinity
                          else distArray Data.Array.! (pos, next) + dpArray Data.Array.!
(Data.Bits.setBit mask next, next)
                          | next <- [0..n-1]]
```

Function logic:

- If ***mask*** includes all cities, ***dp*** returns the distance back to the starting city.

- Otherwise, it iterates over each possible next city (***next***). If ***next*** has not been visited (checked using ***Data.Bits.testBit***), it calculates the minimum distance to visit ***next*** from ***pos*** and recursively continues to compute the distance with the updated subset (***Data.Bits.setBit mask next***).

Path Reconstruction

After filling ***dpArray***, the ***constructPath*** function reconstructs the shortest path by tracing back the choices stored in ***dpArray***:

```
constructPath :: Int -> Int -> Path
constructPath mask pos
   | mask == (2^n)-1 = [toCity pos, startCity]
   | otherwise = toCity pos : constructPath newMask bestNextCity
  where
        possibleNextCities = [(next, distArray Data.Array.! (pos, next) + dpArray
Data.Array.! (Data.Bits.setBit mask next, next))
                | next <- [0..n-1], not (Data.Bits.testBit mask next)]
       (bestNextCity, _) = Data.List.minimumBy (\(_, x) (_, y) -> compare x y)
possibleNextCities
   newMask = Data.Bits.setBit mask bestNextCity
```

Function logic:

- When all cities are visited (***mask*** includes all cities), ***constructPath*** completes the tour by returning to the ***startCity***.

- Otherwise, it chooses the next city (***bestNextCity***) that minimizes the tour length, then recursively calls ***constructPath*** to continue building the shortest path.

This implementation of the TSP problem efficiently finds the shortest tour using dynamic programming with bitmasking. Key auxiliary data structures, including ***distArray*** and ***dpArray***, enable quick lookups and minimize redundant calculations. The helper functions (***toIdx***, ***toCity***, etc.) facilitate seamless transitions between city names and indices, which is essential for the bitmasking technique used in ***dp***.