PFL - Haskell Coursework

Licenciatura em Engenharia Informática e Computação

Group T12_G07

1. Group Members and Participation

Diogo Miguel Fernandes Ferreira - up202205295

Worked on :

1. *cities*
2. *areAdjacent*
3. *distance*
4. *adjacent*
5. *pathDistance*
8. *shortestPath*
9. *travelSales*

Total Contribution : **50%**

Diogo Soares Rocha - up201606166

Worked on :

6. *rome*
7. *isStronglyConnected*
8. *shortestPath*
9. *travelSales*

Total Contribution : **50%**

2. shortestPath function

- **Function definition**

shortestPath computes all shortest paths connecting the two cities given as input and if there are multiple paths with the same minimum distance, it returns all of them.

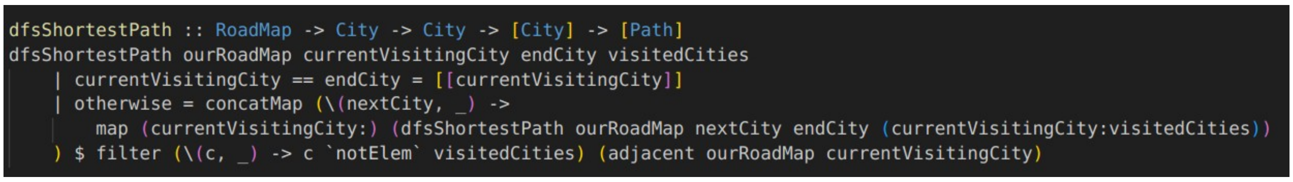
- **Algorithm implemented**

For this function, we implemented a DFS to generate all possible paths between the two cities, later taking advantage of the *pathDistance* function to filter out the minimum distance.

- **Data Structures used**

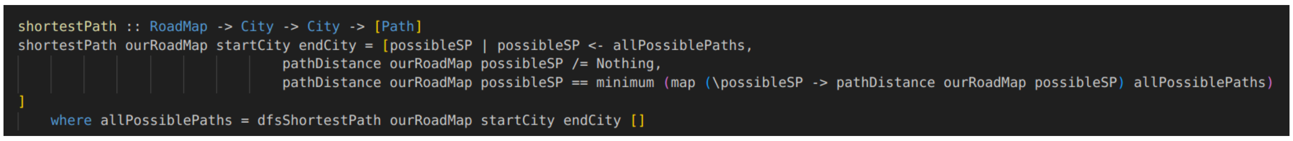
As for data structures, we used the already existing ones, **RoadMap**, which serves as an adjacency list, storing a pair of cities and their associated distance. Also **Path**, a list that represents the cities along a specific route.

- **Algorithm breakdown**



```haskell
dfsShortestPath :: RoadMap -> City -> City -> [City] -> [Path]
dfsShortestPath ourRoadMap currentVisitingCity endCity visitedCities
    | currentVisitingCity == endCity = [[currentVisitingCity]]
    | otherwise = concatMap (\(nextCity, _) ->
        map (currentVisitingCity:) (dfsShortestPath ourRoadMap nextCity endCity (currentVisitingCity:visitedCities))
    ) $ filter (\(c, _) -> c `notElem` visitedCities) (adjacent ourRoadMap currentVisitingCity)
```

We use the *dfsShortestPath* helper function to perform a DFS from the starting city to the destination city with the objective of building a list of possible paths. The function recursively explores each unvisited city, storing them in the *visitedCities* as it moves along to the final city.



```haskell
shortestPath :: RoadMap -> City -> City -> [Path]
shortestPath ourRoadMap startCity endCity = [possibleSP | possibleSP <- allPossiblePaths,
                                                pathDistance ourRoadMap possibleSP /= Nothing,
                                                pathDistance ourRoadMap possibleSP == minimum (map (\possibleSP -> pathDistance ourRoadMap possibleSP) allPossiblePaths)
]
    where allPossiblePaths = dfsShortestPath ourRoadMap startCity endCity []
```

After getting this list, we use a previously implemented function called *pathDistance* to compute the total distance for each path returned by *dfsShortestPath.*
We compare all paths to the minimum total distance path, in order to ensure that only and all shortest paths are returned.

- **Edge Cases**

Valid edge cases for our problem would be:

  - **No existing Path**

If no valid path exists, our function returns an empty list.

  - **Multiple Shortest Paths**

If there are multiple shortest paths, the function returns all of them.

- **Time Complexity**

For large graphs, this solution struggles because of the amount of paths being generated by the DFS.

3. travelSales function

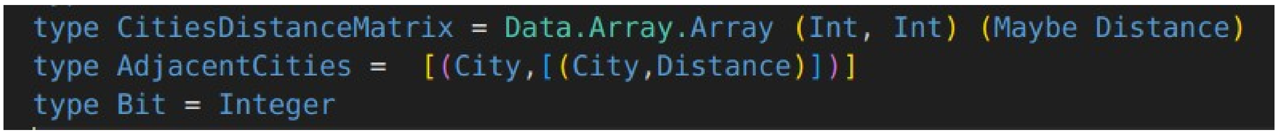
- **Function definition**

Given a *roadMap*, returns a solution of the Traveling Salesman Problem (TSP). It finds the shortest route that visits each city one time and returns to the starting city.

- **Algorithm implemented**

The *travelSales* function uses recursion to explore all possible routes that visit each city exactly once and return to the starting city. We keep track of which cities have been visited by using *bitmasking*, which allows us to check sets of visited cities with efficiency. To solve this problem, we used **this video** to give us a better understanding of the problem, this being an Hamiltonian Cycle.

- **Data Structures used**



```haskell
type CitiesDistanceMatrix = Data.Array.Array (Int, Int) (Maybe Distance)
type AdjacentCities =  [(City,[(City,Distance)])]
type Bit = Integer
```

- **CitiesDistanceMatrix**

A 2D array used to store a pair of cities and their respective distance.

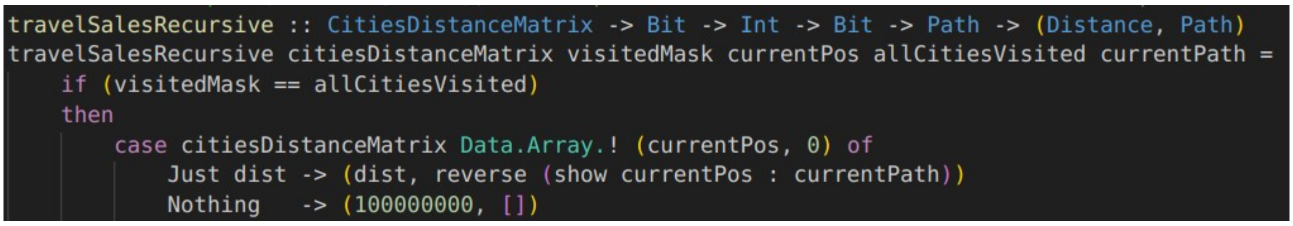
- **AdjacentCities**

A list of tuples giving us the city and a list of cities connected to it and their respective distance.

- **Bit**

A bitmask used to keep track of which cities have been visited in the current path.

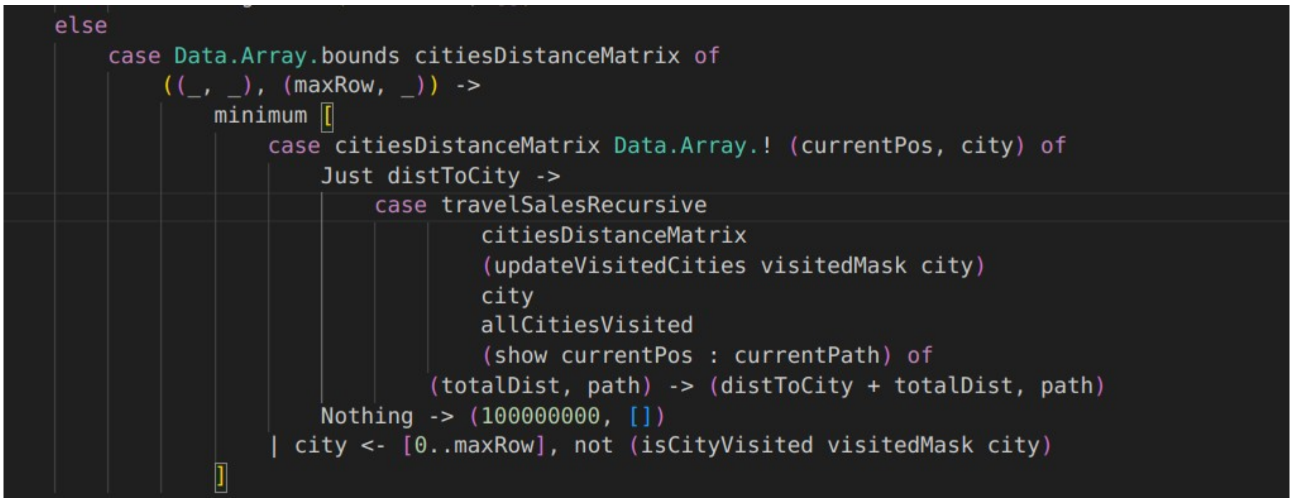
- **Algorithm breakdown**



```haskell
travelSalesRecursive :: CitiesDistanceMatrix -> Bit -> Int -> Bit -> Path -> (Distance, Path)
travelSalesRecursive citiesDistanceMatrix visitedMask currentPos allCitiesVisited currentPath =
    if (visitedMask == allCitiesVisited)
    then
        case citiesDistanceMatrix Data.Array.! (currentPos, 0) of
            Just dist -> (dist, reverse (show currentPos : currentPath))
            Nothing   -> (100000000, [])
```

We start by defining our base case, it occurs when **all cities have been visited**. This is checked by comparing *visitedMask*, a bitmask to keep track of visited cities, to *allCitiesVisited*, a bitmask with all bits set to 1, indicating every city is visited.
In this case, the function attempts to return to the starting city to complete the cycle.

- **If there is a route** back to the starting city, our function adds the distance to the total path length and completes the route.
- **If no route exists** back to the starting city, our function assigns a high cost to the path and returns an empty list, rejecting the path.



```haskell
    else
        case Data.Array.bounds citiesDistanceMatrix of
            ((_, _), (maxRow, _)) ->
                minimum [
                    case citiesDistanceMatrix Data.Array.! (currentPos, city) of
                        Just distToCity ->
                            case travelSalesRecursive
                                    citiesDistanceMatrix
                                    (updateVisitedCities visitedMask city)
                                    city
                                    allCitiesVisited
                                    (show currentPos : currentPath) of
                                (totalDist, path) -> (distToCity + totalDist, path)
                        Nothing -> (100000000, [])
                    | city <- [0..maxRow], not (isCityVisited visitedMask city)
                ]
```

If there are still unvisited cities, the function proceeds with the recursive exploration.
It iterates over all the cities to find neighboring cities to the current city and for each city, our function looks up the distance from *currentPos* to *city* using *citiesDistanceMatrix* stored value.

For each valid route (*Just distToCity*), the function makes a recursive call, updating :

- ***visitedMask***

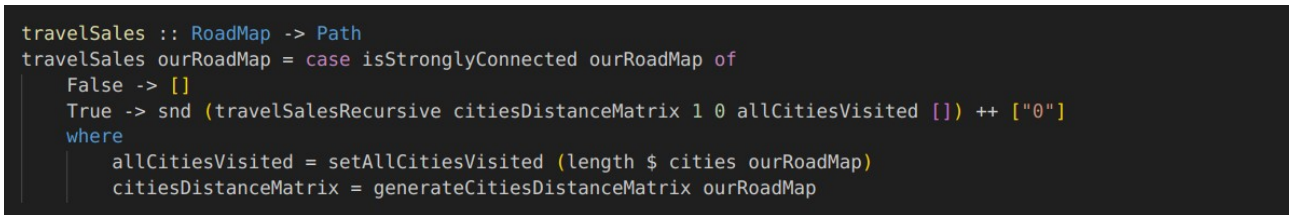
By setting the city's corresponding bit to 1
- ***currentPos***

By setting *city* as the new current city for the next recursion
- ***currentPath***

By adding the current city to the *currentPath*

After the recursive call, *distToCity* + *totalDist* adds the distance from *currentPos* to *city* to the total path distance, keeping track of accumulated distance.

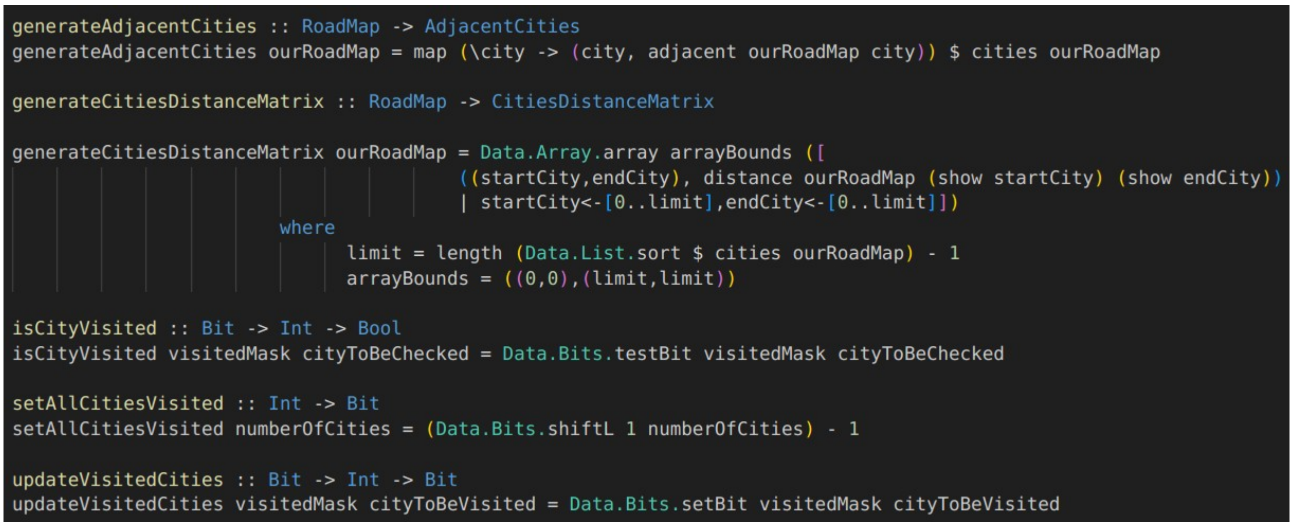
The recursive call returns a list of all possible distances and paths from *currentPos* to the unvisited cities, and with the *minimum* function we can select the shortest path from these options.



```haskell
travelSales :: RoadMap -> Path
travelSales ourRoadMap = case isStronglyConnected ourRoadMap of
    False -> []
    True -> snd (travelSalesRecursive citiesDistanceMatrix 1 0 allCitiesVisited []) ++ ["0"]
    where
        allCitiesVisited = setAllCitiesVisited (length $ cities ourRoadMap)
        citiesDistanceMatrix = generateCitiesDistanceMatrix ourRoadMap
```

We first check if all cities are connected to each other, by using the *isStronglyConnected* function defined before.

- **If not**, we return an empty list because the problem isn't solvable.
- **If they are**, we apply our helper function to get the path with the shortest distance.



```haskell
generateAdjacentCities :: RoadMap -> AdjacentCities
generateAdjacentCities ourRoadMap = map (\city -> (city, adjacent ourRoadMap city)) $ cities ourRoadMap

generateCitiesDistanceMatrix :: RoadMap -> CitiesDistanceMatrix

generateCitiesDistanceMatrix ourRoadMap = Data.Array.array arrayBounds ([
                                        ((startCity,endCity), distance ourRoadMap (show startCity) (show endCity))
                                        | startCity<-[0..limit],endCity<-[0..limit]])
                        where
                              limit = length (Data.List.sort $ cities ourRoadMap) - 1
                              arrayBounds = ((0,0),(limit,limit))

isCityVisited :: Bit -> Int -> Bool
isCityVisited visitedMask cityToBeChecked = Data.Bits.testBit visitedMask cityToBeChecked

setAllCitiesVisited :: Int -> Bit
setAllCitiesVisited numberOfCities = (Data.Bits.shiftL 1 numberOfCities) - 1

updateVisitedCities :: Bit -> Int -> Bit
updateVisitedCities visitedMask cityToBeVisited = Data.Bits.setBit visitedMask cityToBeVisited
```

All helper functions used to facilitate the solution :

- ***generateAdjacentCities***

Generates list of tuples giving us the city and a list of cities connected to it and their respective distance.

- ***GenerateCitiesDistanceMatrix***

Generates matrix responsible for storing the distance of a pair of cities.

- ***IsCityVisited***

Using *Data.Bits.testBit* to check if the city's corresponding bit is 1.

- ***setAllCitiesVisited***

Using *Data.Bits.shiftL* to generate bitmask of all the cities, with all bits turned to 1.

- ***updateVisitedCities***

Using *Data.Bits.setBit* to update a city's corresponding bit to 1.

- **Edge Cases**

Valid edge cases for our problem would be:

  - **Disconnected graph**

If the roadmap is not strongly connected, the function returns an empty list since a valid TSP path does not exist.

  - **No path back to start**

If there is no valid path from the final city back to the starting city after visiting all the cities, the function assigns a very high value to the distance so it is an undesirable choice.

- **Time Complexity**

Since our function explores every possible permutation of city visits, the time complexity is **O(n!).**