Haskell Practical Assignment

This **README** provides a comprehensive guide to the implementation of the two main functions in this project: *shortestPath* and *travelSales*. Each function is discussed in detail, including the algorithms used and auxiliary data structures.

Shortest Path Function: *shortestPath*

The *shortestPath* function calculates the shortest path between two cities in the roadmap, represented as a `RoadMap`. The implementation uses **Dijkstra's** algorithm to efficiently find the shortest path from the starting city to the destination city.

Data Structures Used

In the making of this function, several data structures were used to optimize the computation of the shortest path. These structures are essential for storing intermediate results, managing city distances, and efficiently updating paths during the search.

1. *Adjacency List* (*AdjList*)

The ***adjacency list*** is used to represent the graph as a collection of cities, each associated with a list of neighboring cities and the distances to them. This structure was chosen for its efficiency in accessing neighbors and their distances, crucial for **Dijkstra's** algorithm.

- **Implementation**: Represented as a list of tuples `(City, [(City, Distance)])`.
- **Functionality**: Provides efficient access to all cities directly reachable from a given city, facilitating traversal and exploration in algorithms such as Dijkstra and TSP.
- **Creation**: The *toAdjList* function constructs the adjacency list from the roadmap, calling the *adjacent* function to retrieve the neighbors of each city.

```
toAdjList :: RoadMap -> AdjList
toAdjList roadmap = [(city, adjacent roadmap city) | city <- cities roadmap]
```

2. *Priority Queue* (*PQueue*)

The ***priority queue*** is a central structure in the Dijkstra algorithm, where it maintains the cities ordered by their current shortest known distances. This allows the algorithm to always expand the city with the shortest path next.

- **Implementation**: Represented as a list of tuples `(Distance,City)`.
- **Functionality**: Cities and their distances are inserted in sorted order, ensuring that retrieval of the shortest path is constant.

3. *Distance Table* (*DistMap*)

The ***distance table*** keeps track of the minimum known distance from the starting city to each city in the roadmap. This ensures that each city's shortest path distance is stored and updated only when a shorter path is found.

- **Implementation**: Represented as a list of tuples `(City,Distance)`.
- **Functionality**: Allows for quick lookup of the shortest known distance to a city, enabling efficient updates during the algorithm.

4. *Previous City Table* (*PrevMap*)

The ***previous city table*** records the last city in the path before each city, which is essential for reconstructing the shortest path at the end of the algorithm.

- **Implementation**: Represented as a list of tuples `(City,[City])`.
- **Functionality**: Allows the reconstruction of paths by tracing back from the destination to the start.

Exception Handling Functions

The *searchPrevMap* and *searchDistMap* functions are utility handlers designed to prevent runtime errors when looking up values in the tables, particularly when a city might be missing from the data. These functions ensure the algorithms can handle missing entries gracefully by returning default values.

```
searchPrevMap :: City -> PrevMap -> [City]
searchPrevMap city prevMap = case lookup city prevMap of
    Just prevCities -> prevCities
    Nothing -> []
```

- **Purpose**: Find the previous cities associated with a given City in prevMap.
- **Default Behavior**: Returns an empty list if the city is not found, allowing the algorithm to proceed without errors due to missing data.

```
searchDistMap :: City -> DistMap -> Distance
searchDistMap city distMap = case lookup city distMap of
    Just currentDist -> currentDist
    Nothing -> 1000000000000000
```

- **Purpose**: Retrieves the distance for a given City from distMap.
- **Default Behavior**: Returns a large number (treated as "infinity") if the city is absent, indicating that the city is unreachable in the current path context.

Rebuilding Paths: *buildPaths*

The *buildPaths* function is responsible for reconstructing the shortest paths from the destination city back to the starting city. It recursively builds the paths by backtracking from the destination to the start, using the previous city table to determine the next city in the path.

```
buildPaths :: City -> City -> PrevMap -> [Path]
buildPaths start city prevMap
    | city == start = [[start]]
    | otherwise = [city:path | prev <- searchPrevMap city prevMap, path <-
buildPaths start prev prevMap]
```

- **Purpose**: Recursively builds the shortest paths from the destination city back to the starting city.
- **Base Case**: If the current city is the starting city, the path is complete, and the function returns a list containing the starting city.
- **Recursive Case**: For each previous city, the function appends the current city to the path and recursively builds the path from the start to the previous city.

Dijkstra's Algorithm: *dijkstra* and *updateNeighbors*

The *dijkstra* function is the main function in the **Dijkstra's** algorithm implementation. It encapsulates the core logic of the algorithm, which is supported by the updateNeighbors function. The *updateNeighbors* function updates the priority queue, distance map, and previous map for the neighbors of the current city.

Dijkstra's Algorithm: *dijkstra*

The *dijkstra* function receives the adjacency list, priority queue, distance map, and previous city table as arguments. It iterates through the priority queue, expanding the city with the shortest known distance and calling *updateNeighbors* to update its neighbors. When the priority queue is empty, it means all cities have been visited, and the algorithm terminates, returning the previous city table.

```
dijkstra :: AdjList -> PQueue -> DistMap -> PrevMap -> PrevMap
dijkstra _ [] _ prevMap = prevMap
dijkstra adjList ((dist, city):queue) distMap prevMap = dijkstra adjList newQueue'
newDistMap' newPrevMap'
    where
        neighbors = case lookup city adjList of
            Just list -> list
            Nothing -> []
        currentDist = searchDistMap city distMap
        (newQueue', newDistMap', newPrevMap') = updateNeighbors neighbors queue
distMap prevMap
```

- **Purpose**: Executes Dijkstra's algorithm to find the shortest paths from the starting city to all other cities in the graph.
- **Base Case**: If the priority queue is empty, the function returns the previous city table.
- **Recursive Case**: The function selects the city with the smallest distance from the priority queue, calls the *updateNeighbors* to update the distances to its neighbors, and recursively processes the remaining cities in the queue.

Updating Neighbors: *updateNeighbors*

The *updateNeighbors* function receives the neighbors of the current city, the priority queue, the distance map, and the previous city table. It iterates through the neighbor's list and compares the new distance to the neighbor with the previous known distance. If the new distance is shorter, all three tables are updated with corresponding values. If the new distance is equal to the previous distance, the previous city is appended to the list of previous cities for the neighbor. If the new distance is longer, no updates are made, and the loop continues. When the neighbor's list is empty, the function returns the updated priority queue, distance map, and previous city table.

```
updateNeighbors :: [(City, Distance)] -> PQueue -> DistMap -> PrevMap -> (PQueue,
DistMap, PrevMap)
updateNeighbors [] queue distMap prevMap = (queue, distMap, prevMap)
updateNeighbors ((neighbor, weight):ns) queue distMap prevMap
    | newDist < prevNeighborDist = updateNeighbors ns newQueue newDistMap
newPrevMap
    | newDist == prevNeighborDist = updateNeighbors ns queue distMap
newPrevMapEqual
    | otherwise = updateNeighbors ns queue distMap prevMap
        where
            newDist = currentDist + weight
            prevNeighborDist = searchDistMap neighbor distMap
            newQueue = Data.List.insertBy (\(d1, _) (d2, _) -> compare d1 d2)
(newDist, neighbor) queue
            newDistMap = (neighbor, newDist) : filter ((/= neighbor) . fst)
distMap
            newPrevMap = (neighbor, [city]) : filter ((/= neighbor) . fst) prevMap
            newPrevMapEqual = (neighbor, city : searchPrevMap neighbor prevMap) :
filter ((/= neighbor) . fst) prevMap
```

- **Purpose**: Updates the priority queue, distance map, and previous map for the neighbors of the current city.
- **Base Case**: If there are no more neighbors to process, the function returns the updated priority queue, distance map, and previous map.

- **Recursive Case**: The function updates the distances to each neighbor, maintains the shortest known distances, and recursively processes the remaining neighbors:
    - If the new distance is shorter, the priority queue, distance map, and previous map are updated with the new values.
    - If the new distance is equal to the previous distance, the previous city is appended to the list of previous cities for the neighbor.
    - If the new distance is longer, no updates are made.

Shortest Path Caller: *shortestPath*

The shortestPath function receives a roadmap, a start city, and an end city as arguments. It initializes the necessary data structures and calls the *dijkstra* function to find the shortest paths from the start city to all other cities. Finally, it calls the *buildPaths* function to reconstruct the shortest path from the start city to the end city.

```
shortestPath :: RoadMap -> City -> City -> [Path]
shortestPath roadmap start end = map reverse $ buildPaths start end foundPaths
    where
        initialQueue = [(0, start)]
        initialDistMap = [(start, 0)]
        initialPrevMap = [(start, [])]
        foundPaths = dijkstra (toAdjList roadmap) initialQueue initialDistMap
initialPrevMap
```

- **Purpose**: Organize and call the necessary functions to find the shortest path between two cities in the roadmap.
- **Functionality**: Initializes the priority queue, distance map, and previous city table, then calls *dijkstra* to find the shortest paths. Finally, it reconstructs the shortest path from the start city to the end city.

Traveling Salesman Problem (TSP) Function: *travelSales*

The *travelSales* function calculates the shortest path that visits all cities in the roadmap exactly once and returns to the starting city. The implementation uses a **dynamic programming** with bit masking approach to solve the TSP efficiently. This algorithm was implemented based on this video by William Fiset.

Data Structures Used

In the making of this function, several data structures were used to optimize the computation of the TSP path. These structures are essential for storing intermediate results, managing city distances, and efficiently updating paths during the search.

1. *Adjacency Matrix* (*AdjMatrix*)

The ***adjacency matrix*** is used to represent the graph as a 2D array of distances between cities. This structure allows for efficient lookup of distances between any two cities, essential for the dynamic programming approach to the TSP.

- **Implementation**: Represented as a 2D array of `Maybe Distance`.
- **Functionality**: The matrix has dimensions `n x n`, where `n` is the number of cities, and each cell stores the distance between two cities if they are connected.
- **Creation**: The *toAdjMatrix* function constructs the adjacency matrix from the roadmap, populating the distances between connected cities and setting the rest to `Nothing`.

```
toAdjMatrix :: RoadMap -> AdjMatrix
toAdjMatrix roadmap = emptyMatrix Data.Array.// connections Data.Array.// reversedConnections
    where
        n = length (cities roadmap)
        emptyMatrix = Data.Array.array ((0,0), (n - 1,n - 1)) [((i, j), Nothing) | i <- [0..n-1], j <- [0..n-1]]
        connections = [((read city1 :: Int, read city2 :: Int), Just d) | (city1, city2, d) <- roadmap]
        reversedConnections = [((read city2 :: Int, read city1 :: Int), Just d) | (city1, city2, d) <- roadmap]
```

2. *Memoization Table* (*MemoTable*)

The ***memoization table*** stores the intermediate results of the TSP algorithm, which are used to avoid redundant calculations and improve the efficiency of the dynamic programming approach. This table is essential for storing the shortest path lengths for each city and visited set of cities.

- **Implementation**: Represented as a 2D array of `Maybe Distance`.
- **Functionality**: The matrix has dimensions `n x 2^n`, where `n` is the number of cities, and each cell stores the shortest path length for a city and a set of visited cities. The first dimension represents the current city, and the second dimension represents the set of visited cities.

- **Creation**: The *initializeMemoTable* function constructs the memoization table, initializing the base cases for the dynamic programming approach.

```
initializeMemoTable :: AdjMatrix -> MemoTable
initializeMemoTable adjMatrix = emptyMatrix Data.Array.// baseCases
    where
        n = length adjMatrix
        emptyMatrix = Data.Array.array ((0,0), (n - 1, 2^n - 1)) [((i, j),
Nothing) | i <- [0..n-1], j <- [0..2^n-1]]
        baseCases = [((i, 1 `shiftL` i), Just (adjMatrix Data.Array.! (i,
0))) | i <- [0..n-1]]
```

Auxiliary Functions

The *notIn*, *combinations*, *customMatrixLookup*, and *arrayToPath* functions are auxiliary functions that provide support for the main TSP algorithms functions. These functions are used to check if a city is not in a set, generate combinations of cities, perform custom lookups in the adjacency matrix, and convert an array to a path.

```
notIn :: Int -> Int -> Bool
notIn i subset = (1 `Data.Bits.shiftL` i) Data.Bits..&. subset == 0
```

- **Purpose**: Check if a city is not in a given subset of cities.
- **Functionality**: Performs bitwise operations to determine if the city's bit number is not set to 1 in the subset.

```
combinations :: Int -> Int -> [Int]
combinations 0 _ = [0]
combinations i n
    | i > n = []
    | otherwise = combinations i (n - 1) ++ [x Data.Bits..|. (1 `Data.Bits.shiftL`
(n - 1)) | x <- combinations (i - 1) (n - 1)]
```

- **Purpose**: Generate all possible combinations of cities for a given subset size.
- **Functionality**: Recursively generates all possible combinations of cities for a given subset size N, with I bits set to 1.

```
customMatrixLookup :: Data.Array.Array (Int, Int) (Maybe Distance) -> Int -> Int -
> Distance
customMatrixLookup matrix i j = case matrix Data.Array.! (i, j) of
    Just d -> d
    Nothing -> 1000000000000000
```

- **Purpose**: Find the distance between two cities in a matrix.
- **Default Behavior**: Returns the found distance or a large number (treated as "infinity") if the distance is not found.

```
arrayToPath :: Data.Array.Array Int Int -> Path
arrayToPath array = [show i | i <- Data.Array.elems array]
```

- **Purpose**: Convert an array to a path of cities.
- **Functionality**: Converts an array of city indices to a valid Path structure.

Rebuilding the Path: *recoverPath*, *recoverPathAux*, and *findNextCity*

The *recoverPath*, *recoverPathAux*, and *findNextCity* functions are responsible for reconstructing the found TSP path from the memoization table. They start from the last city visited and recursively backtrack to the starting city, following the shortest path lengths stored in the memoization table.

Process Handler: *recoverPath*

The *recoverPath* function initializes the path array and calls the *recoverPathAux* function to recursively recover the path from the destination city to the starting city.
Finally, it converts the path array to a valid path structure.

```
recoverPath :: AdjMatrix -> MemoTable -> Int -> Int -> Path
recoverPath matrix memo start n = arrayToPath (initialPathArray Data.Array.//
recoveredPathIndices)
    where
        initialIndex = start
        initialState = (1 `Data.Bits.shiftL` n) - 1
        initialPathArray = Data.Array.array (0, n) [(i, -1) | i <- [0..n]]
Data.Array.// [(0, start), (n, start)]

        recoveredPathIndices = recoverPathAux (n - 1) initialIndex initialState []
```

- **Purpose**: Organize the path reconstruction process.
- **Functionality**: Initializes all necessary data structures used in the path recovery process and calls the *recoverPathAux* function to recover the path from the destination city to the starting city. Finally, calls the *arrayToPath* function to convert the array to a valid `Path` structure.

Recovery Main Loop: *recoverPathAux*

The *recoverPathAux* function is the main recursive function that backtracks from the destination city to the starting city, following the shortest path lengths stored in the memoization table. It uses the *findNextCity* function to determine the next city in the path based on the previous city and the current state.

```
recoverPathAux :: Int -> Int -> Int -> [(Int, Int)] -> [(Int, Int)]
recoverPathAux i lastIndex state acc
    | i < 1 = acc
    | otherwise = recoverPathAux (i - 1) newIndex newState ((i, newIndex) : acc)
    where
        newIndex = findNextCity (-1) [0..n - 1] lastIndex state
        newState = state `Data.Bits.xor` (1 `Data.Bits.shiftL` newIndex)
```

- **Purpose**: Recursively backtrack from the destination city to the starting city to recover the shortest path.
- **Base Case**: If the current index is less than 1, the function returns the accumulated path.
- **Recursive Case**: The function determines the next city in the path using the *findNextCity* function and recursively processes the remaining cities.

Next City Finder: *findNextCity*

The *findNextCity* function determines the next city in the path based on the previous city and the current state. It iterates through the possible cities, excluding the starting city and those already visited, to find the city with the shortest path length to the previous city.

```
findNextCity :: Int -> [Int] -> Int -> Int -> Int
findNextCity best [] _ _ = best
findNextCity best (j:js) index state
    | j == start || notIn j state = findNextCity best js lastIndex state
    | otherwise =
        let prevDist = customMatrixLookup memo best state + customMatrixLookup
matrix best lastIndex
            newDist = customMatrixLookup memo j state + customMatrixLookup matrix
j lastIndex
        in if best == -1 || newDist < prevDist
            then findNextCity j js lastIndex state
            else findNextCity best js lastIndex state
```

- **Purpose**: Find the next city in the path based on the previous city and the current state.
- **Base Case**: If there are no more cities to check, the function returns the best city found.
- **Recursive Case**: The function iterates through the possible cities, excluding the starting city and those already visited, to find the city with the shortest path length to the previous city.

TSP DP w/Bit Masking Algorithm: *solveTSP* and child functions

The *solveTSP* function is the main function in the TSP algorithm implementation. It encapsulates the core logic of the algorithm, which is supported by the *updateMemo*, *updateSubset*, *updateNext*, and *calcMinDist* functions.

TSP Parent Function: *solveTSP*

The *solveTSP* function is the main function in the TSP algorithm implementation. It encapsulates the core logic of the algorithm in its child functions, which are responsible for updating the memoization table, subsets, and calculating the minimum distance.

```
solveTSP :: AdjMatrix -> MemoTable -> Int -> Int -> Int -> MemoTable
solveTSP matrix memo start n r
    | r > n = memo
    | otherwise = solveTSP matrix updatedMemo start n (r + 1)
        where
            updatedMemo = updateMemo memo (combinations r n)
```

- **Purpose**: Handle the outermost loop of the TSP algorithm and call the necessary functions to update the memoization table.
- **Base Case**: If the current number of cities is greater than the number of existing cities, the function returns the memoization table.
- **Recursive Case**: The function calls the *updateMemo* function to update the memoization table with the current subset size and recursively processes the next subset size.

Subset Handler: *updateMemo*

The *updateMemo* function is responsible for handling the calling of the *updateSubset* function for each subset of cities.

```
updateMemo :: MemoTable -> [Int] -> MemoTable
updateMemo memo [] = memo
updateMemo memo (subset:subsets)
    | notIn start subset = updateMemo memo subsets
    | otherwise = updateMemo (updateSubset memo subset [0..n - 1]) subsets
```

- **Purpose**: Handle the updating of the memoization table for each subset of cities.
- **Base Case**: If there are no more subsets to process, the function returns the updated memoization table.
- **Recursive Case**: The function calls the *updateSubset* function to update the memoization table for the current subset and recursively processes the remaining subsets. If the starting city is not in the subset, the function skips the subset.

Subset Updater: *updateSubset*

The *updateSubset* function has a similar structure to the *updateMemo* function, but it is responsible for updating the memoization table for a specific subset of cities.

```
updateSubset :: MemoTable -> Int -> [Int] -> MemoTable
updateSubset memo _ [] = memo
updateSubset memo subset (next:nexts)
    | next == start || notIn next subset = updateSubset memo subset nexts
    | otherwise = updateSubset (updateNext memo subset next) subset nexts
```

- **Purpose**: Update the memoization table for a specific subset of cities.
- **Base Case**: If there are no more cities to process, the function returns the updated memoization table.
- **Recursive Case**: It iterates over the existing cities and checks if the next city is the starting city or not in the subset before calling the *updateNext* function to update the memoization table. If the city is not the starting city or not in the subset, the function skips the city.

Next City Updater: *updateNext*

The *updateNext* function is responsible for receiving the minimum distance from the *calcMinDist* function and updating the memoization table with the new distance.

```
updateNext :: MemoTable -> Int -> Int -> MemoTable
updateNext memo subset next = memo Data.Array.// [((next, subset), Just minDist)]
    where
        state = subset `Data.Bits.xor` (1 `Data.Bits.shiftL` next)
        minDist = calcMinDist memo state next [0..n-1] maxBound
```

- **Purpose**: Update the memoization table for the next city in the subset.
- **Functionality**: Call the *calcMinDist* function to calculate the minimum distance for the next city and update the memoization table with the new distance. It also updates the state of the subset by toggling the bit corresponding to the next city.

Minimum Distance Calculator: *calcMinDist*

The *calcMinDist* function calculates the minimum distance for the next city in the subset by looping all possible cities and calculating the distance based on the current state and the previous city.

```
calcMinDist :: MemoTable -> Int -> Int -> [Int] -> Distance -> Distance
calcMinDist _ _ _ [] minDist = minDist
calcMinDist memo state next (e:es) minDist
    | e == start || e == next || notIn e state = calcMinDist memo state next es
minDist
    | otherwise = calcMinDist memo state next es (min minDist sumDist)
        where
            sumDist = customMatrixLookup memo e state + customMatrixLookup matrix
e next
```

- **Purpose**: Calculate the minimum distance for the next city in the subset.
- **Base Case**: If there are no more cities to process, the function returns the minimum distance found.
- **Recursive Case**: The function iterates over the possible cities, excluding the starting city, the next city, and those not in the current state, to calculate the distance based on the current state and the previous city. It updates the minimum distance if a shorter path is found.

TSP Algorithm Caller: *travelSales*

The *travelSales* function is the function that starts the TSP algorithm. It initializes the necessary data structures and calls the *solveTSP* function to find the shortest path that visits all cities in the roadmap exactly once and returns to the starting city.

```
travelSales :: RoadMap -> Path
travelSales roadmap
    | not (isStronglyConnected roadmap) = []
    | otherwise = recoverPath matrix memo start n
        where
            start = 0
            n = length (cities roadmap)
            matrix = toAdjMatrix roadmap
            memo = solveTSP matrix (initializeMemoTable matrix start n) start n 3
```

- **Purpose**: Organize and call the necessary functions to solve the Traveling Salesman Problem.
- **Functionality**: Initializes the adjacency matrix, memoization table, and starting city, then calls the *solveTSP* function to find the shortest path that visits all cities exactly once and returns to the starting city. Finally, it recovers the path from the memoization table. Notably, the function checks if the roadmap is strongly connected before proceeding with the TSP algorithm.

This project was developed by Afonso Neves