Faculdade de Engenharia da Universidade do Porto

Assignment on Functional Programming

TP1

Programação Funcional e Lógica (PFL)

Class 2 - Group 3

Developed by:

Alexandre Costa	up202207499
Sofia Gonçalves	up202205020

Index

1. Shortest Path Implementation

1.1 Auxiliar functions

Function `createAdjList`: Builds an adjacency list for the road map. It iterates over each road (c1, c2, d) in the RoadMap, and addNeighbor adds the city and its connections with distances.

Function `createAllDistancesArray`: Initializes an array with the minimum distances for each city from the starting city. The distance for the starting city is 0, while all other cities are set to maxBound (infinite distance). This array will be updated with shorter distances as Dijkstra's algorithm progresses.

Function `getAdjacentCities`: Retrieves the adjacent cities and their distances for a specific city from the AdjList.

Function `getAllPath`: Reconstructs the shortest paths going from the destination city back to the starting city.

Function `relaxAdjacentCities`: Relaxes adjacent cities for a given city by examining each adjacent city in the adjacency list. Checks if an adjacent city has been visited or if a shorter path exists. If a shorter path is found, it updates the MinHeap with the new distance.

Function updateDistances: Updates the distances if the new distance is less than the current distance; if it's equal, it adds the cities to the existing predecessors.

1.2 Heap functions

Function `createHeap`: Creates an empty MinHeap with a specified capacity. Initializes an array with each position holding a default tuple with maximum distance to facilitate comparisons in the heap.

Function `insert`: Inserts a new element into the MinHeap. If there is capacity, it adds the new element at the end of the array and then calls bubbleUp to maintain the min-heap property.

Function `bubbleUp`: Maintains the min-heap property by bubbling up the element at a given index until the heap property is satisfied. Swaps elements if the child's distance is less than the parent's distance, adjusting positions in the array.

Function `extractMin`: Removes and returns the element with the smallest distance (root of the heap) from the MinHeap. Replaces the root with the last element, decreases the heap size, and calls bubbleDown to restore the heap property.

Function `bubbleDown`: Restores the min-heap property by bubbling down the element at a given index. Chooses the smaller child to swap with, if necessary, until the heap property is satisfied, effectively moving the element down the heap.

Function `findMinForCity`: Searches the MinHeap for the minimum distance entry for a specific city. Returns the element with the smallest distance for that city if found, or Nothing if the city is not in the heap.

1.3 Main function description

The `shortestPath` function is designed to find all possible shortest paths between cities in a given RoadMap, using a modified version of Dijkstra's algorithm. It begins by setting up essential data structures for better performance. First, it creates an adjacency list from the RoadMap, enabling quick access to neighboring cities and distances between them. Then, it initializes an array to hold the shortest-known distances and predecessor cities from the start city to all other cities. Initially, the distance to the start city is set to 0, while distances to all other cities are set to a very high value (maxBound), indicating they are initially unreachable. A min-heap is also initialized to manage unvisited cities, with the start city added first with a distance of 0. This min-heap enables efficient retrieval and updating of the city with the shortest-known distance at each step.

With these structures set up, `shortestPath` invokes a recursive function, `dijkstra`, which implements the modified Dijkstra's algorithm to capture all possible shortest paths. In each iteration, dijkstra extracts the city with the shortest distance from the heap. For this city, it examines each of its unvisited neighbors,

calculating potential new distances from the start city. If the calculated distance to a neighboring city is shorter than previously recorded, it updates the array to reflect this shorter path and clears any previous paths to store only the new shortest path. If the new distance equals the existing shortest distance, it adds the current city as an additional predecessor, thereby capturing all shortest paths.

This approach continues until all reachable cities have been processed, at which point the distances array will hold the shortest distance to each city along with all possible predecessor cities that form the shortest paths. Following `dijkstra`, the `getAllPaths` function is called to backtrack through the array, reconstructing every possible shortest path from the start city to the destination by exploring each predecessor. Starting from the destination, `getAllPaths` recursively follows each predecessor until reaching the start city, generating all shortest paths.

1.4 Complexity and structures used

Adjacency List: The adjacency list is an efficient way to represent the graph, especially for sparse graphs where not all cities are directly connected. The adjacency list only stores the actual connections, making it space-efficient with O(E) storage where E is the number of edges. In terms of operation efficiency, accessing the neighbors fast, as each city's adjacency list entry directly points to its connected cities. This results in `O(E)` time complexity for looking up neighbors across all nodes, contributing significantly to the overall performance of the algorithm.

Array for Distances: The distances and predecessor cities are stored in a fixed-size array (Data.Array.Array Int ([City], Distance)) indexed by city ID. This array allows constant-time access (`O(1)`) to retrieve and update both the shortest-known distances and the list of predecessor cities for each city from the start city. When a shorter path to a city is found during the relaxation process, the distance can be updated immediately, ensuring efficient performance without any search overhead.

Min-Heap for Unvisited Cities: With the use of a min-heap the algorithm efficiently tracks and retrieves the city with the shortest known distance, reducing the complexity

for retrieving the closest city. Each time a city is processed, finding and removing the closest city is an `O(log V)` operation. Additionally, inserting or updating adjacent cities into the heap is also `O(log V)`, resulting in significant performance improvements compared to maintaining a sorted list.

So, adding all the complexities, we get `O((E+V) * log V)` for the entire algorithm.

2. Travel Sales Implementation

2.1 Adjacency matrix functions

Function `createAdjMatrix`: Creates an adjacency matrix with the distances between cities based on the provided RoadMap. It uses `createEmptyMatrix` to start with an empty matrix, with the size according to the number of cities, and applying `addEdgeMatrix` to add the corresponding distance between two cities in both directions to the matrix, given the Edge.

2.2 Table functions

Function `createTableMatrix`: Creates an empty table to store the results of the dynamic programming solution, initializing all entries to represent that no distances or paths have been calculated yet.

Function `setEntryTable`: Calculates and returns the entry of the table, a tuple with the distance and path, for a given city and subset of cities.

Function `fillTable`: Fills the table entries, filtering only valid subsets for each city.

2.3 Auxiliar functions

Function `intToSubset`: Converts a bitmask integer into a list by extracting the indices of the bits set that represent the cities included in the subset.

Function `subsetToInt`: Converts a list into a bitmask integer by setting the corresponding bits for each city on the subset. This integer is used as a coordinate in the matrix, specifically in the second dimension that corresponds to the subset of visited cities, since the matrix indices rely on integer values.

Function `createSubset`: Creates a subset without a specific city. This is useful because on the TSP formula, we generate subsets of cities that excluded a particular city *(s\{c})*.

Function `sumDist`: Computes the sum of the distance accumulated so far and the distance between two given cities.

Function `minim`: Finds the path with the shortest distance from a list of possible paths. The helper function `mini` performs the comparison for each pair.

2.4 Main function description

The `travelSales` main function receives a roadmap and returns the optimal path that visits all cities exactly once and returns to the starting city, as a solution to the Traveling Salesman Problem (TSP).

$$f(i,s) = \min_{c \in s} \{d_{i,c} + f(c, s \setminus \{c\})\}$$

Fig 1. - TSP dynamic programming formula

The data structures used are a list of all cities in the RoadMap (`allCities`), the starting city (`startCity`), an adjacency matrix with the distances between all city pairs of the RoadMap (`matrix`) and an empty table matrix with integer coordinates representing (city, subset) and entries consisting of (distance, path) (`emptyTable`). The `startInt` is used as an index of the city in the matrix.

The `fillTable` function populates the entries necessary to solve the problem using dynamic programming. It generates a list of tuples (city, subset) in `AllSubsets`, starting with the ones with fewer elements and progressing to the larger, as the calculated entries for smaller subsets are going to be used to solve the larger ones. Then it filters the valid subsets, since the only tuple with the `startCity` is the one with the subset of all the other cities expect the start, the entry that we want on `travelSales`, and for every other city, their respective subsets cannot include the city itself or the starting city to prevent revisiting cities. Finally, the function updates the

table with the calculated entries for each valid (city, subset) pair using the `setEntryTable` function.

In the `setEntryTable` function:

- When the subset is empty, it means there are no more cities left to visit. So it calculates the final segment of the path, setting the entry with the distance from the current city (i) back to the starting city, which is looked up from the adjacency matrix, and a path that includes the current city and the starting city to return.
- Otherwise, finds the shortest path up to the current city (i), returning a tuple with the distance and the city (i) added to the front of the path list.
    - The `pathList` stores a list of (path, distance) tuples based on the subset. For each city c in the subset ($c \in s$), gets the corresponding entry ($f(c, s \setminus \{c\})$) from the `table` using `eachPath`, which takes each city c and their respective subset ($s \setminus \{c\}$), created with `createSubset`.
    - Then it adds for each path the distance from the current city i to c ($d_{i,c}$), which is handled by the `sumDist` function ($d_{i,c} + f(c, s \setminus \{c\})$).
    - After all distances are calculated, the algorithm selects the path with the minimum total distance ($\min \{d_{i,c} + f(c, s \setminus \{c\})\}$), using the `minim` function.

The tuple `(optDist, optPath)` gets the entry from the table for the startCity and its associated subset of remaining cities ($f(i,s)$). If the distance is Nothing, it means that the roadmap doesn't have a TSP path and the function returns an empty list. Otherwise, `travelSales` retrieves the optimal path.

2.5 Structures used and complexity

In this context, n is the number of cities.

Adjacency matrix: is an n × n matrix and each entry stores the distance between a pair of cities. Therefore, this structure has space complexity of `O(n^2)`, regardless of the number of cities, and the time complexity for accessing the distance is `O(1)`. The main reason for choosing this data structure is the fast-access to the distance between

two cities, a feature frequently needed in this algorithm to efficiently calculate path costs.

Table matrix: is an n × (2^n) matrix, since the coordinates are defined by the cities and all possible subsets of cities. Each entry contains a tuple that stores both the distance and the corresponding path taken. While this structure has a space complexity of `O(n * 2^n)`, which can be significant, it allows constant time access to any entry of `O(1)`.

The combination of the number of entries of the table used (`n * 2^n`) and the steps needed to compute each entry (`n`) results in the overall time complexity of the algorithm is `O(n^2 * 2^n)`. While not all entries are filled during the execution of the algorithm, the worst-case analysis remains.

3. Contributions

Group members:

- Alexandre Costa (50%): The main functions I worked on were cities, areAdjacent, distance, adjacent, pathDistance and shortestPath.
- Sofia Gonçalves (50%): The main functions I worked on were rome, isStronglyConnected and travelSales.