

PFL-Project1 : Gaph Theory

Participants

- Bruno Miguel Ataide Fortes(202209730)
- Rodrigo Lourenco Ribeiro(202206396)

Participation

Each of us contributed 50% of the required work. We divided the tasks so that each student implemented the two most challenging algorithms. The remaining functions were distributed among us based on our individual strengths and preferences.

Auxiliar Functions

`removeDuplicates`

A recursive function that checks if the first element in a list is already present in the remaining elements.

- *If the element is present, the recursive call proceeds with the rest of the list, excluding the first element.*
- *If the element is not present, it is added to the result, and the recursive call processes the rest of the list.*

`distPath`

This function takes a `RoadMap` and a `Path`, returning a list of distances between adjacent cities in the path.

Example: Given `[city1, city2, city3]`, the output would be `[dist12, dist23]`.

- *First, it creates a list of tuples with adjacent cities.*
- *Then, using `map`, it calculates the distance between each pair of cities in the tuples.*

`roadMapRec`

A recursive function that takes four parameters: `roadMap`, a list of cities, the current maximum number of adjacent cities, and an accumulator with cities that have the same number of adjacent cities.

- **Base case:** When the list of cities to check is empty, the function returns the accumulator.
- **Recursive case:**
 - It calculates the number of adjacent cities for the first city, `len`.
 - If `num < len`, the accumulator is set to the current city.
 - If `num > len`, the accumulator remains unchanged, and the current city is ignored.
 - Otherwise, the current city is added to the accumulator.

Main Functions

Function 1

This function uses an auxiliary function to remove all duplicates from a list.

- To get all cities, we first create a list of all cities by taking the first (index 0) and second (index 1) values from each tuple in the `RoadMap`, using the `concatMap` function.
- We then apply the duplicate removal function to ensure each city appears only once.

Function 2:

A function that return true if 2 cities are adjacent,. For two cities to be adjacent, they must appear together in the same tuple within the RoadMap To achieve this we use the any function ,passing an auxiliar function that check if The first city in the tuple (index 0) matches one of the cities we are checking (city1), and the second city in the tuple (index 1) matches the other city (city2). Since the graph is undirected, we also need to check the reverse.

Function 3

A function that calculates the distance between two cities.

- The distance between `city1` and `city2` can be found by retrieving the third value in the tuple `(city1, city2, Dist)` or `(city2, city1, Dist)`.

Function 4

A function that returns all cities adjacent to a given city.

- *To achieve this, we use list comprehension to gather all cities that have the input city as one of the elements in their pair.*

Function 5

This function takes two attributes: RoadMap and city.

- *It checks if the path is null or contains only one value.*
- *It verifies if there are any adjacent cities that do not have a route to others; if this is the case, it returns nothing.*
- *Otherwise, it sums all the values in the list of distances calculated in distPath. The map is used to transform all 'Just' values into integers.*

Function 6

This function calculates the list of cities that have more adjacent cities Basically it just uses the auxiliar function roadMapRec

Function 7

In this function, we check if the graph is strongly connected.

- *A strongly connected graph is one in which every vertex is reachable from every other vertex.*
- *To achieve this, we implement an auxiliary function using depth-first search (DFS).*
- *We also use the cities function, if the length of the cities obtain in the graph is equal do to cities obtain in the (DFS), we can say that is stringly connected.*

function 8:

Shoertestpath

To implement the shortest path we used the bounding and pruning algorithm, since Bounding and pruning are strategies used to enhance the efficiency of search algorithms, especially when searching for optimal solutions like the shortest path in graphs.

Overview:

- The **shortestPath** function is designed to find all possible paths between two specified cities in a given graph(RoadMap) and return the shortest path with their respective distances.
- **Bounding**: The algorithm keeps track of the best (shortest) distance found so far. This distance is used to prune paths that cannot possibly be shorter, avoiding unnecessary exploration.
- **Pruning**: The function eliminates paths that exceed the known shortest distance, allowing for faster execution by reducing the search space.

Auxiliar functions

`dfsshortestparh`

The dfsShortestPath function effectively implements a depth-first search algorithm to find all paths from a source city to a destination city while employing bounding and pruning techniques to enhance efficiency. By checking distances and ensuring that cycles do not occur, it systematically explores the graph to discover the shortest path.

- First it checks if the current cumulative distance exceeds the best-known shortest distance. If the condition is true, the function immediately returns the `allpaths` list, effectively pruning the search. If we already found a shorter path, we can stop the search because, we won't find a better path.
- Then we check if the current city is the same as the destination city. If they are the same, the function evaluates the distance of the current path: If `dist` is less than `googdistances`, it means we found a new shorter path, so it creates a new path including the current city and its distance, and returns it as a single-item list. If `dist` is not less than `googdistances`, it still adds the current path to `allpaths`, which allows us to keep track of all paths even if they aren't the shortest.
- If the destination has not been reached, the function needs to explore adjacent cities to continue the search. So we iterate over the list of adjacent unvisited cities, and for each unvisited adjacent city, it recursively calls `dfsShortestPath` to update the current path to include the city just visited, and to update the cumulative distance by adding the distance to the edge leading to the unvisited city.
- We use list comprehension to generate a list of adjacent cities to the current city that have not been visited in the current path. To do that we iterate through all adjacent cities obtained from the `adjacent` function and filter out any city that is already in the current path. This ensures that no cycles are created.
- We create a variable that holds the best-known shortest distance found so far among all paths. To do that we first see if `allpaths` is empty, and we initialize `googdistances` with the maximum possible integer value. This allows any newly found path to be shorter. If `allpaths` is not empty, it retrieves the distance of the first path found, to get the distance component from the first tuple in `allpaths`.

Main function

`shortestpath`

- The function first checks if the source city is the same as the destination city. If so, it returns a list containing a single path since the cities are the same there is no path.
- Then it checks if there are any paths found by calling `dfsShortestPath`. If no paths are found, it returns an empty list. If paths are found, it retrieves and returns all paths found by the `dfsShortestPath` function.

Function 9:

TSP

The algorithm used is based on a **dynamic programming** strategy that tracks the state of visited cities using a bitmask with `n` bits, where `n` is the number of cities in the graph.

Overview

- The bitmask allows the algorithm to efficiently track which cities have been visited. For example, if the first three cities are visited, the bitmask would be `1110000`.

- The algorithm also uses an **adjacency matrix** to store the distance between each pair of cities, which makes it easier to access distances between any two cities directly.

Auxiliary Functions

The `tsp` function relies on several auxiliary functions, explained below:

`isConnected`

This function is essential because it uses **Depth-First Search (DFS)** to verify if, from a given starting city, all other cities in the graph are reachable. This check is critical since a TSP solution is only possible in a fully connected graph.

- **DFS:**
 - **Base Case:** If the current city is already in the `visited` list, DFS returns the `visited` list unchanged, preventing redundant visits and avoiding infinite loops, especially in cyclic graphs.
 - **Recursive Case:** If the city has not been visited:
 - Adds the city to the `visited` list.
 - Recursively applies DFS to each neighboring city of the current city, using `foldl`:
 - `foldl` iterates over each neighbor.
 - Calls DFS for each neighbor, using the updated `visited` list as the accumulator.

`tspAdjAux`

This function calculates the TSP path using the adjacency matrix and the bitmask.

- **Base Case:** When all cities have been visited (indicated by a bitmask where all bits are set to 1), `tspAdjAux` checks if there is a connection from the final city back to the starting city. If a connection exists, it returns the distance of this complete path; if not, it returns a very large value (representing infinity) to indicate an invalid path.
- **Recursive Case:** For incomplete paths, the function attempts to visit each unvisited city:
 - **Generate Possible Paths:** For each unvisited city, it calculates the distance between the current city and the next unvisited city using the adjacency matrix.
 - **Recursive Call:** The function then recursively calls itself, marking this next city as visited and updating the path. The `currCity` is updated to this next city in preparation for the next recursive step.

Main Function

1. **Graph Connectivity Check:** Before calling `tspAdjAux`, the main function uses `isConnected` to ensure that the input `RoadMap` is fully connected. A TSP solution is impossible for disconnected graphs.

2. **Preparation of Initial Values:**

- Converts the `RoadMap` to an adjacency matrix (`adjM`).

- o Initializes variables like `visited` (marking the start city), `allVisited` (a bitmask representing all cities as visited), and `currCity` (starting at city 0).