PFL-Project1 :

Graph Theory

Participants

Bruno Miguel Ataide Fortes(202209730)

Rodrigo Lourenco Ribeiro(202206396)

Participations

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

Shortestpath

To implement the shortest path, we used the bounding and prunning algorith, 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

dfsshortestpath

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 wont 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 Updates the current path to include the city just visited, and to updates 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 iterates through all adjacent cities obtained from the adjacent function and filters 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 inicialize 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 the cities are 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.

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 “currCit” is updated to this next city in preparation for the next recursive step.

Main Function

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.

Preparation of Initial Values:

- Converts the `RoadMap` to an adjacency matrix (`adjM`).
- Initializes variables like `visited` (marking the start city), `allVisited` (a bitmask representing all cities as visited), and `currCity` (starting at city `0`).