PFL - Haskell Coursework

Created by:

Ângelo Oliveira - up202207798@up.pt

José Costa - up202207871@up.pt

Tasks Performed by each member:

Ângelo Oliveira (50%):

- areAdjacent (task 2)
- adjacent (task 4)
- rome (task 6)
- isStronglyConnected (task 7)
- shortestPath (task 8)

José Costa (50%):

- cities (task 1)
- distance (task 3)
- pathDistance (task 5)
- isStronglyConnected (task 7)
- travelSales (task 9)

ShortestPath Explanation:

The `shortestPath` function implements Dijkstra's algorithm to find the shortest paths from a source city to a target city within a road map represented as a graph.

Overview of the Implementation

Data Structures Used

- **MinHeap:** A priority queue implemented as a list of tuples (City, Distance). This structure allows for efficient retrieval of the city with the minimum distance, which is crucial for Dijkstra's algorithm. The heap enables efficient access to the next city to process based on the shortest known distance.
- **CityDistances:** A list of tuples storing distances from the starting city to each city. It keeps track of the shortest distance discovered so far for each city during the algorithm's execution.
- **CityParentNodes:** A list of tuples mapping each city to its predecessor cities. This structure is used to reconstruct all possible paths from the start city to the end city after the distances have been determined.

Algorithm Description

Dijkstra's Algorithm: The core algorithm used here is Dijkstra's algorithm, which efficiently finds the shortest path in a weighted graph with non-negative weights (our case). The algorithm works by maintaining a set of vertices whose shortest distance from the source is known and repeatedly selects the vertex with the smallest known distance to explore its neighbors.

Initialization:

The algorithm begins by assigning a tentative distance value to every vertex. For the initial vertex (the starting city), this value is set to zero, while all other vertices are initialized to infinity, indicating that they are initially unreachable.

A priority queue is used to keep track of vertices to explore, starting with the source vertex.

Main "Loop":

While there are still vertices to explore in the queue:

- The vertex with the smallest tentative distance is extracted from the queue (this is the current vertex).
- All unvisited neighbors of the current vertex are examined.
- For each neighbor, the algorithm calculates the potential distance from the source vertex to the neighbor via the current vertex.
- If this calculated distance is less than the previously recorded distance for that neighbor, the algorithm updates the neighbor’s distance and records the current vertex as a predecessor.

Path Reconstruction:

Once the algorithm has processed all vertices, it can reconstruct the shortest paths from the source to each vertex by backtracking through the recorded predecessor information.

Functions:

- **shortestPathDijkstra:** Orchestrates the overall process by initializing distances and invoking the core dijkstra function.
- **initializeDijkstra:** Sets up the initial distances, queue, and predecessor tracking for the starting city.
- **dijkstra: Implements the main logic of the algorithm, recursively exploring neighbors, updating distances, and predecessors.
- **relaxNeighbors and relaxNeighbor:** Handle the process of updating distances for neighboring cities based on the current city being processed.
- **getAllShortestPaths:** Facilitates path reconstruction by backtracking through the predecessor nodes once the algorithm has finished running.

Key Concepts:

- **Relaxation:** A critical step where the algorithm checks if a shorter path to a neighboring city can be found through the current city, leading to updates in the distances and predecessor relationships.
- **Backtracking:** After the main loop, the implementation uses recorded predecessors to construct all possible shortest paths from the source city to the target city, which is useful for scenarios where multiple paths of equal length exist.

Justification of Data Structures

MinHeap: The use of a min-heap is critical for performance in Dijkstra's algorithm. It allows for efficient extraction of the city with the smallest distance and insertion of new distances. Although lists are used for simplicity, in practice, a more sophisticated data structure (like a binary heap) would provide better performance. Even though we couldn't implement a heap fully in nature, we made it in a way , such that all the operations used for searching have linear time complexity, since we keep our minHeap sorted at all times.

CityDistances and **CityParentNodes**: These lists are effective for managing distance and predecessor information. While other structures like dictionaries could be used for faster lookups, the tuple lists provide simplicity in maintaining order and iterating through the cities.

TravelSales Explanation:

The `travelSales` function is designed to solve the Traveling Salesman Problem (TSP) using two distinct approaches: a brute-force method that computes all possible paths and their corresponding distances, yielding the smallest one, and a dynamic programming approach that utilizes a list to represent visited cities, akin to bit masking.

Overview of the Implementation

Data Structures Used

- **Matrix:** A 2D list that stores the distances between cities. This representation allows for quick access to the distance between any two cities, making it easier to compute path lengths during the exploration phase.
- **VisitedArray:** A list that keeps track of which cities have already been visited during the path exploration. This ensures that each city is visited exactly once in a given path, adhering to the rules of the TSP.
- **CitiesToIDs:** A mapping from city names to unique integer identifiers. This allows for easy conversion between city names and their corresponding indices in the distance matrix.

Algorithm Description

Brute-Force Approach: The first algorithm explores all possible paths through the graph represented by the road map. It recursively generates all paths starting from a given city, attempting to visit all other cities exactly once before returning to the starting point. This exhaustive search guarantees finding the optimal path but may be inefficient for larger graphs due to its factorial time complexity.

Dynamic Programming with List Masking: For larger graphs ($n \geq 8$), we utilize a dynamic programming approach. This algorithm maintains a list to represent the set of visited cities instead of using bit manipulation. This list allows for a more manageable representation of the cities visited during the traversal. By recursively updating this list, the algorithm computes the shortest paths in a more efficient manner than the brute-force approach.

Initialization:

The algorithm begins by initializing several key structures:

- A distance matrix that holds the distances between all pairs of cities.

- A `visitedArray` to track which cities have been visited.
- A mapping from city names to their respective IDs for easier indexing.

Main "Loop":

The core of the algorithm involves a recursive function that explores neighboring cities:

- Starting from the current city, the algorithm examines all adjacent cities.
- For each unvisited neighbor, it recursively explores further, updating the path and total distance.
- When all cities have been visited, it checks if returning to the starting city is possible and, if so, updates the best known path if this path is shorter.

Path Reconstruction:

Once the recursive exploration is complete, the best path found is converted back into a readable format, showing the order of cities to be visited.

Functions:

- **adjacentCityDistances:** Finds all adjacent cities and their respective distances from a given city.
- **totalDistPath:** Calculates the total distance of a given path.
- **convertPath:** Transforms a list of city IDs back into city names.
- **initializeCitiesToIDs:** Sets up a mapping from city names to their corresponding IDs.
- **getDistanceBetweenCities:** Retrieves the distance between two specified cities, accounting for possible directed edges.
- **initializeMatrix:** Constructs the distance matrix based on the input road map.
- **areAllVisited:** Checks if all cities have been visited.
- **setVisited:** Updates the `visitedArray` to mark a city as visited.
- **updateBestPath:** Evaluates and updates the best path found if the current one is shorter.
- **exploreNeighbors:** Recursively explores unvisited neighbors and attempts to find optimal paths.
- **findOptimalPath:** Orchestrates the path exploration, updating the best path as necessary.
- **tspPaths:** Generates all possible paths through the graph, returning only complete cycles.
- **tspShortestPath:** Identifies the shortest path among all possible cycles found.

Key Concepts:

- **Brute-Force Search:** The brute-force algorithm comprehensively explores all potential routes, ensuring that the optimal path is identified, albeit with a higher computational cost for larger datasets.
- **Dynamic Programming:** The dynamic programming approach utilizes memoization techniques to avoid redundant calculations, significantly improving efficiency. The use of a list for visited cities provides clear tracking of the exploration state.
- **Matrix Representation:** Using a matrix for distance representation allows quick lookups of distances between cities, facilitating rapid calculations during path evaluations.

Justification of Data Structures

Matrix: The 2D matrix is crucial for efficient distance retrieval between cities. It allows constant time complexity for distance lookups, significantly speeding up the path evaluation process compared to searching through a list of edges.

VisitedArray: The use of a list to track visited cities provides simplicity and efficiency in marking and checking the status of each city during the exploration process. This structure directly supports the algorithm's requirements for TSP, where each city must be visited exactly once.

CitiesToIDs: This mapping serves to translate between city names and their indices in the distance matrix. It simplifies the implementation by allowing the use of numeric indices for faster access while maintaining the ability to reference city names.

It is worth noting that the algorithm's time complexity is $O(n^{2} * 2n)$ for the dynamic programming approach when n is greater than 8, and O(n!) for the brute-force approach when n is smaller than 8. This complexity arises from the nature of the problem, which involves exploring all possible paths through the graph. We opted for this dual approach to ensure robust performance across different graph sizes, utilizing the brute-force method for $n < 8$ where it yields better results, while leveraging dynamic programming for larger datasets to maintain efficiency.