Adela Solorio A01637205

Integrated Activity 2

For this activity we were asked to help a company that wants to venture into internet services by implementing algorithms that could help them find optimal solutions for various concerns they have.

1. **First concern**

The first concern is that they want to find the optimal way to wire with optical fiber connecting neighborhoods in such a way that information can be shared between any two neighborhoods, giving as an input a square matrix representing the network with the distances in kilometers between the neighborhoods in the city.

To solve this, we implemented Prim's algorithm for Minimum Spanning Trees problems using an adjacency matrix to represent a weighted graph because it returns a way to visit each vertex (neighborhood) with the least cost (least amount (kilometers) of optical fiber used).

We applied this algorithm in a singular function that consists of 3 main parts:

- Initialization: It initializes arrays to
    - track the parent nodes in the MST for each vertex (*parent*).
    - hold the minimum weights to connect each vertex to the MST (*key*).
    - record which vertices have been included in the MST (*inMST*).
- Building of the MST:
    - It starts with the first vertex, setting it as the root.
    - It then iterates and selects the vertex with the smallest key that is not yet included in the MST and updates the *key* values for all adjacent vertex to it, adds it to *inMST*, and adjusts *parent* accordingly.
- Printing of the MST:
    - After getting the MST, it prints each edge (the vertices that compose it) in letters and their corresponding weight.

The time complexity of this implementation is $O(N^2)$, where N is the number of vertices (neighborhoods). This because for each neighborhood, the algorithm iterates

through all N neighborhoods to find the minimum key in each step, having two nested for loops.

Our implementation is efficient for dense graphs, as it directly uses an adjacency matrix. However, for sparse graphs, Prim's algorithm can be optimized to O(E logN) using adjacency lists and a priority queue, where E is the number of edges.

2. **<u>Second concern</u>**

The second concern is that they want to know what is the shortest possible route that visits each neighborhood exactly once and returns to the neighborhood of origin, giving as an input the same square matrix of distances.

To solve this, we implemented Repetitive Nearest Neighbor algorithm for Traveling Salesman Problem because it returns the path that visits each neighborhood and returns to the origin with minimum cost (distance).

We applied this algorithm in 5 functions:

1. *tripTraveler*: Calculates the total cost of a given trip by adding the weights of the edges in the *trip* path. Has a complexity of $O(N)$ where N is the number of nodes.
2. *tripPrinter*: Prints the sequence of nodes in a trip with their indices converted to their corresponding uppercase letters. Has a complexity of $O(N)$ where N is the number of nodes because trough each one of them.
3. *findMin*: Finds the minimum-cost unvisited node that is directly connected to the current node. Has a complexity of $O(N)$ where N is the number of nodes because it scans the row of distances to find the minimum unvisited one.
4. *nearestNeighbor*: Generates a path that visits each node exactly once by moving to the nearest unvisited node from the current node until all nodes are visited. Finally, it returns to the start node. Has a complexity of $O(N^2)$ because it calls *findMin* for each node.
5. *repNearestNeighbor*: Finds the best trip by using the nearest neighbor algorithm starting from each node, comparing the total costs, and selecting the trip with the minimum cost. Has a complexity of $O(N^3)$ because it calls *nearestNeighbor* for each starting node.

Therefore, the time complexity for this implementation is $O(N^3)$ where N is the number of nodes/vertices (neighborhoods). This means that it is not an efficient implementation for large graphs, but it gives an optimal solution.

3. **Third concern**

The third concern is that they want to know the maximum information flow that can go from a starting neighborhood to a final neighborhood, giving as an input a square matrix of data representing the maximum capacity of data transmission between neighborhoods.

To solve this, we implemented Ford-Fulkerson's method with Edmonds-Karp's algorithm to calculate maximum information flow.

We applied this algorithm using two functions:

1. *bfs*: Performs a Breadth-First-Search on the graph to find an augmenting path from the source to the sink. If it finds one, it returns true and fills the parent array with the nodes along the path. Has a complexity of $O(V + E)$ where V is the number of vertices and E is the number of edges in the graph. This because it explores all nodes and edges in the graph.
2. *fordFulkerson*: Repeatedly calls *bfs* to find augmenting paths, finds the flow through the path, and updates the residual graph. The function continues until no more augmenting paths can be found, and it returns the total maximum flow.
   Has a complexity of $O(VE^2)$ because an outer loop runs while *bfs* can find an augmenting path, which can be up to $O(E)$ times.
   Having a total time complexity of $O(VE^2)$ and giving an optimal solution.

4. **Fourth concern**

The fourth concern is that they want to have a way to decide, given a new service contract, which exchange is geographically closest to that new contract, giving as an input the geographic location of several "exchanges" to which new homes can be connected.

The original input should be in the form (x,y), but we were not able to parse it correctly, so the format of the input we receive is x y.

To solve this, we implemented Voronoi Diagram's algorithm using CGAL library to visualize then determine the closest exchange for a new service contract, this because it allows to enter the points for each exchange, and then the points for the homes and tells in which of the Voronoi cells (that belong to an exchange) they are in.

Our implementation uses Delaunay triangulation to divide the set of points into triangles, ensuring that no point is inside the circumcircle of any triangle, used to then compute the Voronoi Diagram that divides a plane into regions, each corresponding to a specific input point (exchange). It uses CGAL's data structures to handle both Delaunay triangulation and the Voronoi diagram, as well as CGAL's handles to navigate and query the Voronoi diagram.

Building the Delaunay triangulation and Voronoi diagram has a complexity of $O(NlogN)$ where N is the number of vertices, the locate operation for each query point takes $O(logN)$, and iterating over the edges of the Voronoi cell (polygon) takes $O(m)$, where m is the number of edges in the cell. The overall complexity is $O(NlogN + qlogN + qm)$, where q is the number of query points.