Juan Sebastian Botero - A00394729
Felipe Velasco - A00380555

Documentation TI2-
Computación y estructuras discretas

Ideas:

1. Word Ladder Game:

We brainstormed ideas for a game involving finding the shortest path between words. Representing words as nodes and connections as edges in a graph. Players change one letter at a time to navigate and find the shortest path.

2. Network Connections:

We discussed a program used to find the best network routing with routers as nodes and connections as edges in a graph. Using Dijkstra's or Bellman-Ford algorithms.

3. Flight finder:

We thought about a program that could calculate the best route from one city to another taking into consideration all the connections necessary to get to the destination and how long it will take, or how much it will cost, or the distance to find the best option for the client.

4. Social Network Friend Analysis:

Analyzing social networks and people as nodes and relationships as edges in a graph. This led to finding influential individuals, identifying communities, and recommending friends to people that might know them.

5. Maze Solver:

We thought about a maze solver program. Mazes can be represented as graphs with cells as nodes and adjacent connections as edges. Graph traversal algorithms could be used to solve the maze.

6. Subway System:

After watching a video on the internet about a bacteria that finds the most efficient way to get its food, and how researchers in Japan used it to model their subway system, we thought about doing it with graphs.

These ideas emerged through collective brainstorming as we explored various domains, envisioning how graphs could create engaging and challenging scenarios.

ENGINEERING METHOD:

Problem context:

In the context of an airline, inefficiencies in travel paths have been causing issues for both the airline and its customers. Passengers are experiencing long flights and multiple connections, leading to extended travel times and inconvenience. To address this problem, the airline is seeking a system that enables them and their pilots to be more efficient and find the shortest path between cities. (By shortest way we are talking about shorter distance, and therefore shorter time)

Problem identification:

Identification of needs and symptoms:

- Flight operators and pilots require a reliable and fast way to determine the shortest route between airports to optimize flight paths.
- The airline doesn't have a software solution to address this situation.
- The solution should provide a clear and easily understandable visualization of the optimal route, helping the airline, flight operators and pilots plan their routes better.

Problem definition:

The airline is in need of a software solution that will help them to effortlessly identify and print the shortest route between airports. This solution should use the power of algorithms and data analysis to calculate the most efficient path, considering distance and reducing travel time. The software should provide an easy to use interface that displays the identified route clearly, ensuring flight operators and pilots can easily comprehend and follow the recommended path.

Collection of information:

In order to solve the airline's shortest route problem, we gathered information from various sources to understand some valuable information and relevant algorithms needed in our implementation:

1. BFS (Breadth-First Search):

BFS is a vertex-based technique used to find the shortest path in a graph. It follows a first-in-first-out approach using a Queue data structure. When a vertex is visited, it is marked, and its adjacent vertices are visited and stored in the queue. GeeksforGeeks provides a detailed explanation of BFS and its implementation [1].

2. DFS (Depth-First Search):

DFS is an edge-based technique that uses a Stack data structure. It involves two stages: pushing visited vertices into the stack and popping vertices if there are no unvisited neighbors. Comparing it to BFS, DFS has its own advantages and use cases. Further information on DFS can be found on GeeksforGeeks [1].

3. Dijkstra's Shortest Path Algorithm:

Dijkstra's algorithm is used to find the shortest paths from a source vertex to all other vertices in a graph. It generates a Shortest Path Tree (SPT) with the source as the root. By maintaining two sets, one containing the vertices included in the SPT and the other containing the remaining vertices, the algorithm selects the vertex with the minimum distance from the source at each step. GeeksforGeeks offers a comprehensive explanation and implementation details for Dijkstra's algorithm [2].

4. Kruskal's Algorithm:

Kruskal's algorithm is a greedy approach used to find the minimum cost spanning tree in a graph. It selects the smallest weight edges that do not create a cycle in the constructed Minimum Spanning Tree (MST) so far. This algorithm is widely used for finding efficient routes or network connections. For more information and a detailed explanation of Kruskal's algorithm, GeeksforGeeks is a reliable resource [3].

5. Prim's Algorithm:

Prim's algorithm is another greedy approach to finding the Minimum Spanning Tree (MST) of a graph. It starts with an empty spanning tree and iteratively adds edges to connect the MST. By maintaining two sets of vertices, one containing the included vertices and the other containing the remaining vertices, Prim's algorithm selects the minimum weight edge from the edges connecting the two sets at each step. GeeksforGeeks provides comprehensive insights into Prim's algorithm [4].

By referring to these sources, we can gain a solid understanding of the algorithms mentioned and their potential application in solving the airline's shortest route problem. This information will guide us in selecting the most appropriate algorithm or combination of

algorithms to develop a software solution that efficiently finds and prints the shortest routes for the airline's needs.

References:
[1] GeeksforGeeks: BFS and DFS in Graph. Available at: https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/
[2] GeeksforGeeks: Dijkstra's Shortest Path Algorithm - Greedy Algo-7. Available at: https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/
[3] GeeksforGeeks: Kruskal's Minimum Spanning Tree Algorithm - Greedy Algo-2. Available at: https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/
[4] GeeksforGeeks: Prim's Minimum Spanning Tree (MST) - Greedy Algo-5. Available at: https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5

Idea Generation:

We had a brainstorming session where we explored various ideas to address the airline's shortest route problem in relation to the problem context.

1. Graph with DFS and Dijkstra Algorithm:

Our first idea involves creating a graph representation of the airline's connection cities, with nodes representing the cities and edges representing the distance between them. By implementing a Depth-First Search (DFS) and Dijkstra's Algorithm, we can find the shortest route between two cities. The resulting route would be printed on the console, providing flight operators and pilots with the necessary information to optimize their flight paths.

2. Graphical Interface:

We considered improving the solution by adding a graphical interface where users would be able to interact with the system, adding or removing stations and routes. By utilizing DFS and Dijkstra's Algorithm, the system would calculate the shortest route based on the updated graph. The graphical interface would provide a user-friendly platform for managing and visualizing the route information.

3. Graph with BFS, Prim Algorithm, and Console Instructions:

As an alternative approach, we proposed creating a graph representation of the airline's connecting cities, again using nodes for cities and edges for the routes. By applying Breadth-First Search (BFS) and Prim's Algorithm, the system would generate a list of nodes

and edges, printed on the console, serving as instructions for users to reach their desired destination. This idea provides a clear set of steps to follow, ensuring efficient travel routes.

4. Database with City and Route Information:

Another idea we thought of was building a database containing all the cities and their respective routes to other cities. The system would then generate a list of cities and routes for the user, providing them with a suggested itinerary to follow. This approach focuses on utilizing data to optimize travel paths without the need for complex algorithmic calculations.

5. Graph with BFS and Dijkstra Algorithm:

We also considered using a graph representation with nodes representing cities and edges representing connections. By implementing BFS and Dijkstra's Algorithm, we could find the shortest route between cities. Similar to the first idea, the resulting shortest route would be printed on the console, assisting flight operators and pilots in their decision-making.

6. Graph with DFS, Kruskal Algorithm, and Graphical Interface:

Our final idea involved utilizing a graph representation of the map, with nodes representing cities and edges representing connections. By combining DFS and Kruskal's Algorithm, users would be able to add or remove stations and routes through a graphical interface. This interactive approach would allow for flexibility in managing the network of stations and optimizing travel paths.

These ideas emerged from our brainstorming session, considering the airline's specific problem context. Each idea aims to create a graph algorithm and user-friendly interfaces to provide efficient and optimized solutions for the airline's shortest route problem.

Evaluación y selección de la mejor solución:

After reading the prompt from the work, we found out that option 4 is not viable because we need to implement a graph algorithm and a database is not what we need. We also discarded idea 2 because we don't have enough time to implement a good graphic interface that will look good and be user friendly. We decided not to include one and just use simple commands by console. Lastly, we decided to remove option 6 because of the graphic interface as well.

After this we were left with this options and we evaluated them better:

1. DFS - Dijkstra Algorithm - Console print:

This idea combines the efficiency of Dijkstra's algorithm for finding the shortest distances between nodes and the speed of DFS traversal for discovering adjacent connections. By applying Dijkstra's algorithm, we can accurately calculate the shortest path between two nodes. The DFS traversal, on the other hand, quickly identifies connections between adjacent nodes by constructing subtrees. The resulting path, for example: "City1 -> City2 -> City3," is printed in the console, providing a clear and concise representation of the shortest route.

3. BFS - Prim Algorithm - Instruction prints:

In this approach, we leverage Prim's algorithm to identify the edges with the lowest weights between nodes. By utilizing the BFS traversal, we ensure efficient exploration of connections between nodes. Instead of directly printing the entire path, this idea focuses on generating instruction prints. These instructions guide the user step-by-step, providing precise details on which nodes to visit, helping them navigate through the optimal route.

5. Graph with BFS and Dijkstra Algorithm:

Combining the strengths of BFS and Dijkstra's algorithm, this idea aims to find the shortest distances between nodes. Dijkstra's algorithm calculates the direct distances between nodes, ensuring the shortest path. Simultaneously, the BFS traversal method efficiently explores connections between nodes. The resulting path, tailored to the user's request, is then printed in the console, displaying only the necessary information for the journey. This approach provides a concise and focused solution to the airline's shortest route problem.

Rubric to evaluate the best idea:

Criteria A: Faster and better mastered:

1. [2] Faster to search for adjacency.
2. [1] Slower to look for adjacency.

Criteria B: Accuracy in searching for routes:

1. [2] Always find the shortest path between peer nodes.
2. [1] Find a shorter path for the entire graph

Criteria C: Impression of results:

1. [3] Displays results in a user-understandable way.
2. [2] Prints the correct route.
3. [1] Displays the route in a difficult to understand way.

Options\Criteria	A	B	C	**Result**
Option 1	2	2	2	**6**
Option 3	2	2	1	**5**
Option 5	1	2	2	**5**

After evaluating our options we decided to go with option 1 DFS - Dijkstra Algorithm - Console print because it is the most efficient and accurate way that we can get our airline to calculate the shortest flights and connections between two cities and print it in an easy to understand way that will not compromise our due date.

Graph TAD:

Graph TAD		
Graph = {Vertex = <vertex>}		
{ inv : x|x ∈ Graph, n(x) > 0}		
Primitive operations:		
Method	Input	Output

AddVertex:	Vertex	Graph
AddEdge:	Vertex x Vertex	Graph
NumOfVertex:	Graph	Int
CheckConnection:	Graph	Boolean

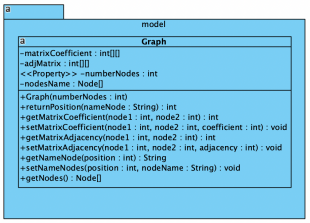
AddVertex(vertex)
Variant
“Is in charge of adding a new vertex to the graph”

AddEdge(vertexA, vertexB)
Variant
“Is in charge of adding a new edge connecting two vertex in the graph”

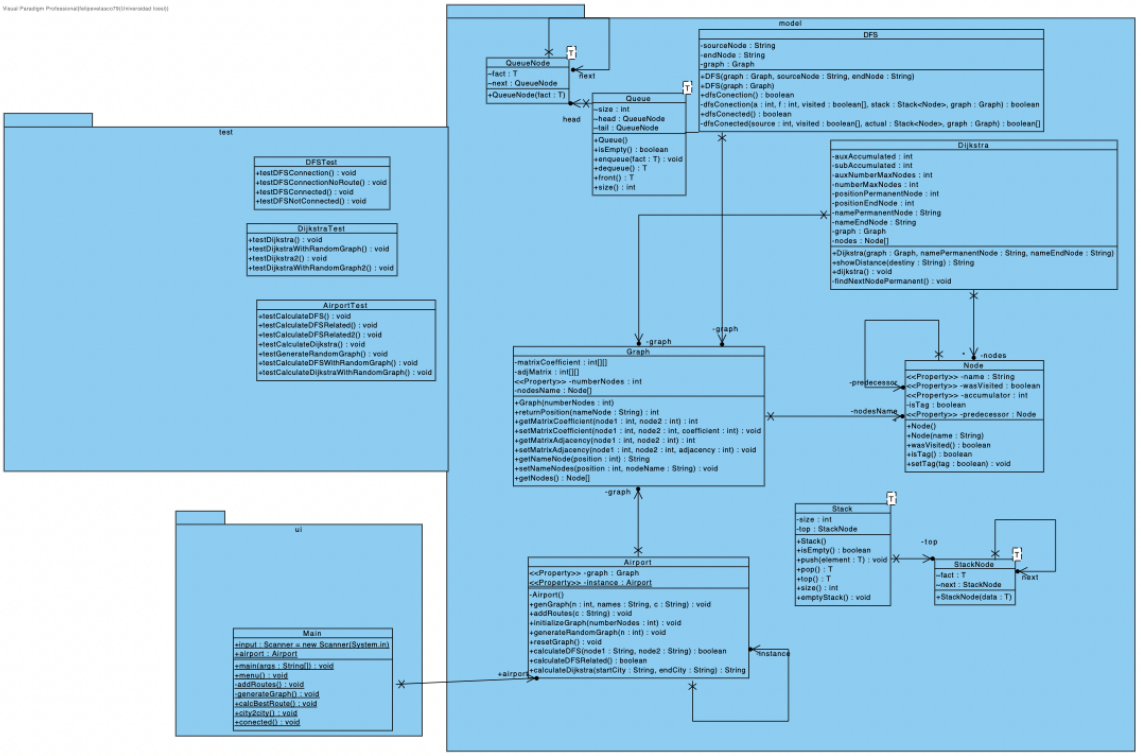
NumOfVertex()
Analyzer
“Is in charge of returning the number of vertex in the graph”

CheckConnection()
Analyzer
“Is in charge of running the graph and checking if it is a connected graph”

Diseño diagramas de clase del TAD:



Diseño de Diagramas de Clase de la Solución del Problema:



Diseño de Casos de Pruebas

Objective: Verify the effectiveness of the methods in the DFS class			
Method	**Scenario**	**Input Values**	**Expected Result**

dfsConection()	Valid route between source and end nodes	- Graph: Generated graph with 6 nodes and connections: "A-B-2/A-C-5/B-D-3/C-D-1/C-E-4/D-E-6/D-F-7/E-F-2" - Source Node: "A" - End Node: "F"	The method returns true indicating a valid route between the source and end nodes.
dfsConection()	No route between source and end nodes	- Graph: Generated graph with 4 nodes and connections: "A-B-1/A-C-1/C-D-1" - Source Node: "A" - End Node: "D"	The method returns false indicating no valid route between the source and end nodes.
dfsConected()	Graph is connected	- Graph: Generated graph with 6 nodes and connections: "A-B-1/A-C-1/B-D-1/C-D-1/C-E-1/D-F-1"	The method returns true indicating that the graph is connected, and all nodes are reachable.

dfsConected()	Graph is not connected	- Graph: Generated graph with 6 nodes and connections: "A-B-1/B-C-1/C-D-1/D-E-1"	The method returns false indicating that the graph is not connected, and there are disconnected nodes.

Objective: Verify the effectiveness of the methods in the Dijkstra class			
Method	**Scenario**	**Input Values**	**Expected Result**
dijkstra()	Valid route between source and end nodes	- Graph: Generated graph with 5 nodes and connections: "A-B-1/A-C-2/B-D-3/C-E-4" - Source Node: "A" - End Node: "E"	The method calculates the shortest path from the source node "A" to the end node "E" and stores the result in the Dijkstra object.
dijkstra() with random graph	Valid route between random source and end nodes	- Graph: Randomly generated graph with 10 nodes - Source Node: "City0" - End Node: "City9"	The method calculates the shortest path from the random source node "City0" to the random end node "City9" and stores the result in the Dijkstra object. If no route exists, the message "There is no route available between City0 and City9 in the random graph." is displayed.

dijkstra()	Valid route between source and end nodes	- Graph: Generated graph with 6 nodes and connections: "A-B-2/A-C-5/B-D-3/C-D-1/C-E-4/D-E-6/D-F-7/E-F-2" - Source Node: "A" - End Node: "B"	The method calculates the shortest path from the source node "A" to the end node "B" and stores the result in the Dijkstra object.
dijkstra() with random graph	Valid route between random source and end nodes	- Graph: Randomly generated graph with 8 nodes - Source Node: "City0" - End Node: "City7"	The method calculates the shortest path from the random source node "City0" to the random end node "City7" and stores the result in the Dijkstra object. The result is displayed.

Objective: Verify the effectiveness of the methods and the implementation of the Airport Class			
Method	**Scenario**	**Input Values**	**Expected Result**
calculateDFS()	Valid route between source and end nodes	- Graph: Generated graph with 5 nodes and connections: "A-B-1/A-C-2/B-D-3/C-E-4" - Source Node: "A" - End Node: "D"	The method calculates whether there is a valid route between the source node "A" and the end node "D". Returns true if a route exists, otherwise false.

calculateDFS()	Invalid end node	- Graph: Generated graph with 5 nodes and connections: "A-B-1/A-C-2/B-D-3/C-E-4" - Source Node: "A" - End Node: "o"	The method returns false as the end node "o" does not exist in the graph.
calculateDFSRelated()	Graph with connected nodes	- Graph: Generated graph with 5 nodes and connections: "A-B-1/A-C-2/B-D-3/C-E-4"	The method determines whether all nodes in the graph are connected. Returns true if all nodes are connected, otherwise false.
calculateDFSRelated()	Graph with disconnected nodes	- Graph: Generated graph with 5 nodes and connections: "A-B-1/A-C-2/C-E-4"	The method determines whether all nodes in the graph are connected. Returns false as nodes "D" and "E" are disconnected.
calculateDijkstra()	Valid route between source and end nodes	- Graph: Generated graph with 5 nodes and connections: "A-B-1/A-C-2/B-D-3/C-E-4" - Source Node: "A" - End Node: "D"	The method calculates the shortest path from the source node "A" to the end node "D" using Dijkstra's algorithm. Returns the string "La mejor ruta es: A -> B -> D".

generateRandomGraph()	Randomly generated graph with connected nodes	- Number of Nodes: 10	The method generates a random graph with 10 nodes and connections. Returns true if all nodes are connected, otherwise false.
calculateDFS()	Valid route between random source and end nodes	- Graph: Randomly generated graph with 10 nodes - Source Node: "City0" - End Node: "City5"	The method calculates whether there is a valid route between the random source node "City0" and the random end node "City5". Returns true if a route exists, otherwise false. The result is displayed.
calculateDFS()	Invalid end node with a random graph	- Graph: Randomly generated graph with 10 nodes - Source Node: "City0" - End Node: "City9"	The method returns false as the end node "City9" does not exist in the random graph. The message "There is no route available between City0 and City9 in the random graph." is displayed.