# Documentation

WF-Delivery System

## Team:


| Name | Code |
|---|---|
| Óscar Andrés Gómez Lozano | A00394142 |
| Cristian Eduardo Botina Carpio | A00395008 |
| Juan Manuel Marín Angarita | A00382037 |


## Problem Statement

The company "Wings Factory" has expanded its presence worldwide and recently inaugurated several branches in different countries. Among them is their latest construction project in Colombia, which is the achievement they are most proud of, as a total of 3 branches were built, spread across the 3 cities that constitute a significant part of the country.

This business operates as follows: when receiving an order from any branch, the operator must check if it is possible to deliver it to the customer's home from the current establishment. If not, they must notify the customer and reject the delivery. If possible, the system should calculate the most optimal route for the delivery person to send the order in the shortest possible time.

Additionally, the branches regularly send advertisers who travel throughout the city to spread information about their products to the public. It is important that the route be efficient, as this would save the maximum amount of money on fuel and time.

We have been asked to create a program that facilitates the calculation of the routes the company needs to plan daily to improve its efficiency and, therefore, provide better customer service.

## Engineering Method

### Phase 1:

#### 1. Description of the context specifying causes and symptoms.

The company "Wings Factory" has asked for help to create a program capable of defining optimal routes for its operations of delivery of addresses and advertising.

### Possible causes

- Operators spend a lot of time looking on a map to see if incoming orders can be accepted or not.
- Delivery drivers don't follow the most optimal route to finish home deliveries.
- Publicists don't have a defined route to tour the whole sector in the most optimal way.

### Possible symptoms

- The incoming calls for delivery orders are attended to, requiring a significant amount of time.
- Delivery drives spend more time than expected on each route.
- Advertising expenses are higher than expected.

## 2. Concrete and unambiguous identification of the problem.

The company "Wings Factory" requires a program that optimizes its times of operation in the following aspects:

- Determining whether it is possible to fulfill an order, based on the branch it is received at and the destination address.
- Specification of the most optimal route to follow a delivery driver to deliver an order in the shortest possible time.
- Definition of a route in which a publicist goes through all the houses in a sector of the city using the shortest possible time.

## 3. Requirement Specification

| | |
|---|---|
| Client | Wings Factory |
| User | Operators (call-in delivery order operators, delivery driver operators and publicists) |
| Functional requirements | FR01. Create houses<br>FR02. Create connections between houses<br>FR03. Verify path between two houses<br>FR04. Calculate minimum path between two houses<br>FR05. Generate most efficient path that connects all houses |

| | |
|---|---|
| Problem context | "Wings Factory" requires a program to optimize various aspects of its operations. This includes determining order feasibility based on branch and destination, finding the most efficient delivery routes for drivers, and planning time-efficient routes for publicists to cover all houses in a specific city sector. |
| Non-functional requirements | NFR1. The system must be user friendly, so it has to have an easy to use visual interface that shows the city map in a visual way, using pictures to represent the houses.<br>NFR2. The system must be made using the Java programming language and following the git flow principles.<br>NFR3. The program must save the data each time it is closed, and load it every time it is initialized.<br>NFR4. The program must show the most efficient path, highlighting the streets.<br>NFR5. The program will show an animation of the publicist going through each vertex in the calculated efficient path.<br>NFR6. The program must have a fast response as the pcs where it is going to be installed have limited hardware.<br>NFR7. Wings Factory wants to expand its production and scope, so the program also have to be easily scalable. |

| | | | |
|---|---|---|---|
| Name or identifier | FR01 - Create houses | | |
| Abstract | The system must be able to generate a house, each house will have a unique identifier and a type, WingsFactory headquarter or regular house. | | |
| Inputs | Input name | Data type | Condition of select or repetition |
| | houseID | String | It is unique |
| | type | HouseType | Allowed types:<br>- Headquarter<br>- RegularHouse |
| General activities needed to obtain the results | 1. Create a House based on the given attributes<br>2. Store the house in a structure that allows connecting it to others. | | |
| Result or postcondition | The structure that stores the houses is modified with a new house in it. | | |
| Outputs | Output name | Data type | Condition of select or repetition |
| | | | |

| Name or identifier | FR02 - Create connections between houses | | |
|---|---|---|---|
| Abstract | The system must be able to generate a connection between two houses. Each connection is represented as a street and has the time that takes to get from one house to another. | | |
| Inputs | Input name | Data type | Condition of select or repetition |
| | fromId | String | |
| | targetId | String | |
| | duration | double | |
| General activities needed to obtain the results | 1. Search the from house by its id.<br>2. Search the target house by its id.<br>3. Add a connection between two houses, assigning the duration.<br>4. Store the edge in a structure so that it can be accessed later. | | |
| Result or postcondition | The structure that stores the connections is modified with a new connection. | | |
| Outputs | Output name | Data type | Condition of select or repetition |
| | | | |

| Name or identifier | FR03 - Verify path between two houses | | |
|---|---|---|---|
| Abstract | The system must be able to verify if a house is reachable from another, by visiting other houses as intermediates. | | |
| Inputs | Input name | Data type | Condition of select or repetition |
| | fromId | String | |
| | targetId | String | |
| General activities needed to obtain the results | 1. Search from house by its id<br>2. Search target house by its id<br>3. Visit from house neighbors until reaching the target house.<br>4. If all possible houses are visited by its neighbors and the target is not found, return false, else true. | | |
| Result or postcondition | None | | |
| Outputs | Output name | Data type | Condition of select or |

| | | | |
|---|---|---|---|
| | | | repetition |
| | isReachable | boolean | It is true when the second house is reachable, otherwise it is false. |

| | | | |
|---|---|---|---|
| Name or identifier | FR04 - Calculate minimum path between two houses | | |
| Abstract | The system must be able to calculate the minimum duration path between two houses that have a direct or indirect connection. | | |
| Inputs | Input name | Data type | Condition of select or repetition |
| | fromId | String | |
| | targetId | String | |
| General activities needed to obtain the results | 1. Search from house by its id<br>2. Search target house by its id<br>3. Visit each house by its neighbors choosing the minimum duration connections until getting into the target house.<br>4. Return the houses that form the minimum duration path in order. | | |
| Result or postcondition | None | | |
| Outputs | Output name | Data type | Condition of select or repetition |
| | minimumPath | House[] | |

| | | | |
|---|---|---|---|
| Name or identifier | FR05 - Generate most efficient path that connects all houses | | |
| Abstract | The system must be able to calculate the most efficient path that includes all the connected houses, that is, the path that requires less time and visits all houses at once. | | |
| Inputs | Input name | Data type | Condition of select or repetition |
| | startId | String | |
| General activities needed to obtain the results | 1. Search the start house by its id<br>2. Visit each house by its neighbors choosing the minimum duration connections.<br>3. Return the houses that form the minimum duration path in order. | | |
| Result or postcondition | None | | |

| Outputs | Output name | Data type | Condition of select or repetition |
|---|---|---|---|
| | minimumPath | House[] | |

# Phase 2:

## 1. Investigation

### Route optimization algorithm

There are optimization algorithms designed to solve problems related to finding the most optimal path to pass through different points in the shortest possible time. Problems like the "Traveling Salesman Problem" and the "Vehicle Routing Problem" are some of those related to the subject.

Due to the number of route generation combinations that exist when there are many points to visit, the complexity of brute force algorithms that aim to find the shortest path is extremely high.

Today, there are algorithms based on graph structure such as Floyd-Warshall and software that allow finding an approximate solution to this type of problem.

### Graphs theory

Graph theory aims to visually represent abstract datasets using nodes or vertices and the connections or relationships that these nodes can have with other nodes through edges. Thanks to this theory, significant advancements have been made in the analysis of large volumes of data.

In the context of reachability between points, a graph can be used to model the connections between different points in space. Each point is represented as a node, and the connections between the points are represented as edges. For example, if we have a road network with different intersections, each intersection can be a node, and the roads that connect them can be the edges.

### Dijkstra algorithm

The Dijkstra algorithm is an example of a solution to determine the shortest network length in a graph between two points or vertices. Utilizes the arc weights to find the path that minimizes the total value between the origin node and the other nodes in the graph. The specific meaning of the value depends on how the arc weights are defined in the graph. It could represent factors such as time, cost, or distance. The algorithm aims to determine the most efficient route in terms of the assigned weights, taking into account the starting node and the remaining nodes in the graph.

### 2. References

Lillo, A.G. (September, 2022). What Are Route Optimization Models. Retrieved from https://xcloudy.es/que-son-los-modelos-de-optimizacion-de-rutas

Graph Everywhere. (n.d.). Graph theory. Retrieved from https://www.grapheverywhere.com/teoria-de-grafos/#:~:text=Con%20esta%20teor%C3%ADa%20se%20busca,de%20amplios%20vol%C3%BAmenes%20de%20data.

Cassingena Navone, E. (October 24, 2022). Dijkstra's Shortest Path Algorithm - Illustrated and Detailed Introduction. freeCodeCamp. Retrieved from https://www.freecodecamp.org/espanol/news/algoritmo-de-la-ruta-mas-corta-de-dijkstra-introduccion-grafica/#:~:text=El%20algoritmo%20de%20Dijkstra%20encuentra,los%20dem%C3%A1s%20nodos%20del%20grafo.

## Phase 3:

### 1. Idea generation method

The team decided to meet in person to analyze the statement and propose ideas for solving each of the presented problems. These are:

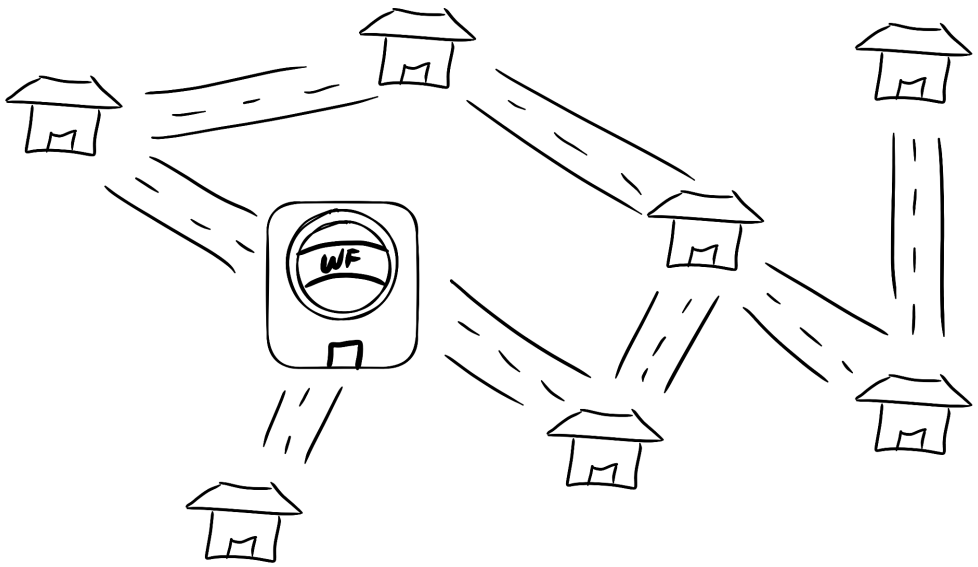
- How can we verify that a house is reachable from a branch?
- If the delivery is feasible, how can we determine the shortest path to reach it?
- How can we calculate the optimal route for an advertiser to visit all the houses in a city?

The ideas for possible solutions are presented for further analysis, with the aim of eventually discarding the least viable ones.

### 2. Ideas raised

Based on the previous research, the team believes that the best structure to store the houses in each city and the branches is a graph, as it allows for easy modeling of the connections between them and analyzing the reachability of each entity with respect to another.

Here is an example of how the graph representation would look in the given context:




Graphs can be implemented using both adjacency lists and adjacency matrices, so the implementation of their structures can be considered.

Now, based on this structure, the following ideas were proposed:

### Scope between two entities

- Brute Force: One approach would be to evaluate every possible path from a given branch to the other houses in its city and stop if the program finds one that connects to the given destination. This way, it would determine if the two entities are reachable to each other.
- BFS Algorithm: This algorithm allows finding the distance between a vertex in a graph and all those reachable from it. So, if after executing it, the target vertex has no distance, it would mean that it is not reachable.
- Validate connection: One could traverse the structure in which the vertices are stored (either an adjacency list or matrix) and check if there is a connection between the branch and the respective house. If no connection is found, it can be assumed that they are not reachable.

### Calculate shortest path

- Choose the shortest edge: From a branch, we can always follow the edge with the lowest weight until the final vertex is found.
- Dijkstra's Algorithm: This algorithm allows finding the minimum distance between a vertex and the rest of the graph, providing the specified route as well.
- Floyd-Warshall Algorithm: By using this algorithm, all paths from one vertex to another are analyzed to find the approximate most optimal or shortest route between them.

### Calculate best path traversing all entities

- DFS Algorithm: Implement the Depth-First Search algorithm to explore all possible paths in the graph and determine if there is a circuit that passes through all the points. This will help identify the optimal route that covers all houses.

- Minimum Spanning Tree: Apply the Minimum Spanning Tree algorithm to find the subset of connections that connects all the points with the minimum total weight.

## Phase 4:

### 1. Discarded ideas

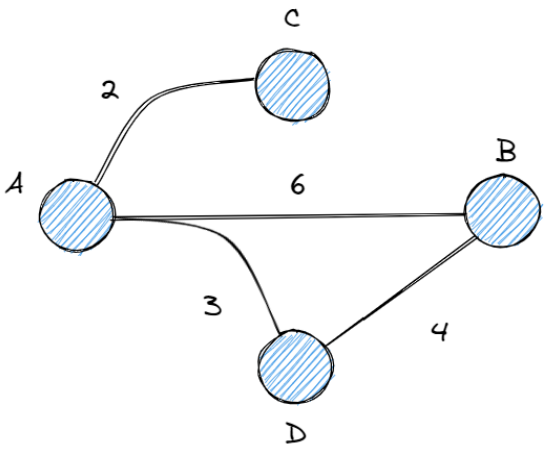
As we ended the formulation of the ideas, we had to reject the ideas that could not be implemented as a solution due to the requirements raised before.

#### Scope between two entities

- Brute force method cannot be implemented as a solution because the more houses are reached by Wings Factory the slowest the program would be, tending to never return an answer even with only 50 houses on the map.

#### Calculate shortest path

- Choosing the shortest path to calculate an efficiency path is not a good option to approach an answer as it is not guaranteed that the lowest value is going to return the most efficient path or even return a path, thinking towards this scenario:




Where we need to go from A-B, the path A-C goes nowhere but its the shortest, the path A-D-B has a weight of 7 and reaches the point B using the lowest values, but is not the most efficient path, finally, the path A-B has a weight of 6 and is the most efficient way to reach B.

### 2. Probable solutions

Now, with the worst ideas discarded, we have to complement the ideas that are going to be considered probable solutions.

#### Scope between two entities

- BFS Algorithm: This algorithm allows finding the distance between a vertex in a graph and all those reachable from it. So, if after executing it, the target vertex has no distance, it would mean that it is not reachable. The complexity of this solution is

O(|V| + |E|), where |V| is the number of vertices and |E| is the number of edges, as we want to know if the house is reachable, it will be a good and easy to use option.

- Validate connection: One could traverse the structure in which the vertices are stored (either an adjacency list or matrix) and check if there is a connection between the branch and the respective house. If no connection is found, it can be assumed that they are not reachable, this idea is stipulated as a combination of various techniques that allocates different ways to reach a node based on the implementation of a graph itself. Will require more time to be implemented and its complexity is not tested.

### Calculate shortest path

- Dijkstra's Algorithm: This algorithm allows finding the minimum distance between a vertice and the rest of the graph, providing the specified route as well. This algorithm would be useful for publicity distribution, as it also provides very good performance when making the most efficient path (its worse case is 0(V^2) when using a matrix, or O((|V| + |E|) log V) using priority queues). Its major contra is when the graph is too dispersed, the making of the shortest path is slower.
- Floyd-Warshall Algorithm: By using this algorithm, all paths from one vertex to another are analyzed to find the approximate most optimal or shortest route between them. It's easy to implement compared with other alternatives and works with every kind of graph, either if the edge is negative, but this solution has a very bad performance that makes it unable to work with big graphs (its worse case is O(|V|^3)).

### Calculate best path traversing all entities

- DFS Algorithm: Implement the Depth-First Search algorithm to explore all possible paths in the graph and determine if there is a circuit that passes through all the points. This will help identify the optimal route that covers all houses. This algorithm is a good option to interpret the paths over the map, either if they are or are not in the city, but this algorithm does not guarantee that it will return the shortest path, also it does have problems when a branch is too big.
- Minimum Spanning Tree: Apply the Minimum Spanning Tree algorithm to find the subset of connections that connects all the points with the minimum total weight. This a really good option as it guarantees an optimal solution to travel in a pondered graph from a certain origin, its major problem is that it doesn’t take on count restrictions, only the weights of the edges.

## Phase 5:

### 1. Criteria evaluation

To select the best ideas that solve the initially presented problems, they are going to be evaluated based on the following criteria:

- Implementation difficulty (being 1 very difficult and 5 very easy)
- Adaptability to the problem context (being 1 very inappropriate and 5 very appropriate)
- Time complexity (being 1 too slow and 5 very fast)

### Scope between two entities

**Idea 1**: BFS Algorithm (Accepted).

- Implementation (4 | 5): Its implementation, regardless of the graph structure, is usually quite simple.
- Adaptability (5 | 5): It perfectly solves the need to determine if a vertex in the graph can be reached from another.
- Time complexity (4 | 5): Its complexity is O(V+E), which means it is linear.

**Idea 2**: Validate connection (Denied).

- Implementation (3 | 5): Its implementation will be slightly more complex as it needs to traverse all the vertices and edges of the graph while constantly validating if the reached vertex is the expected one.
- Adaptability (1 | 5): It does not adapt to the problem since it does not meet the requirement of needing a fast response and it does not consider possible paths between the source and destination, providing a result only if they are directly connected.
- Time complexity (3 | 5): Due to the constant validations it needs to perform, its time complexity is moderate.

### Calculate shortest path

**Idea 1**: Dijkstra's Algorithm (Accepted).

- Implementation (3 | 5): Its implementation requires some time as it involves constant validations to determine if the calculated path weighs more or less than another.
- Adaptability (5 | 5): It perfectly suits the problem's need as it provides information about the most optimal path between two vertices.
- Time complexity (3 | 5): Its complexity also depends on the priority queue used and is generally O((V + E) log V).

**Idea 2**: Floyd-Warshall Algorithm (Denied).

- Implementation (2 | 5): Its implementation is more complex than the previous one as it traverses and calculates distances between all vertices of the graph.
- Adaptability (2 | 5): It does not fit the problem since it computes more distances than required and does not provide the expected path.
- Time complexity (1 | 5): Its time complexity is O(V^3), which is very high to provide the expected response for the problem.

### Calculate best path traversing all entities

**Idea 1**: DFS Algorithm (Denied).

- Implementation (4 | 5): Its implementation, regardless of the graph structure, is usually quite simple.

- Adaptability (2 | 5): It does not adapt to solving the problem since, although it can provide a path that visits all vertices, it does not guarantee that it is the most optimal one.
- Time complexity (4 | 5):  Its complexity is O(V+E), which means it is linear.

**Idea 2**: Minimum Spanning Tree (Accepted).
- Implementation (3 | 5): Its implementation is usually more complex than DFS, as it involves stages of edge sorting and can vary depending on the data structures used.
- Adaptability (5 | 5): Despite its complexity, it always provides an approximate most optimal path that visits all vertices in the graph.
- Time complexity (3 | 5): Its time complexity will depend on the type of MST being implemented, as well as the size of the graph. The most common algorithms for finding the MST are Kruskal's algorithm and Prim's algorithm, with time complexities of O(E log V) and O((V + E) log V) respectively.

## 2. Conclusions

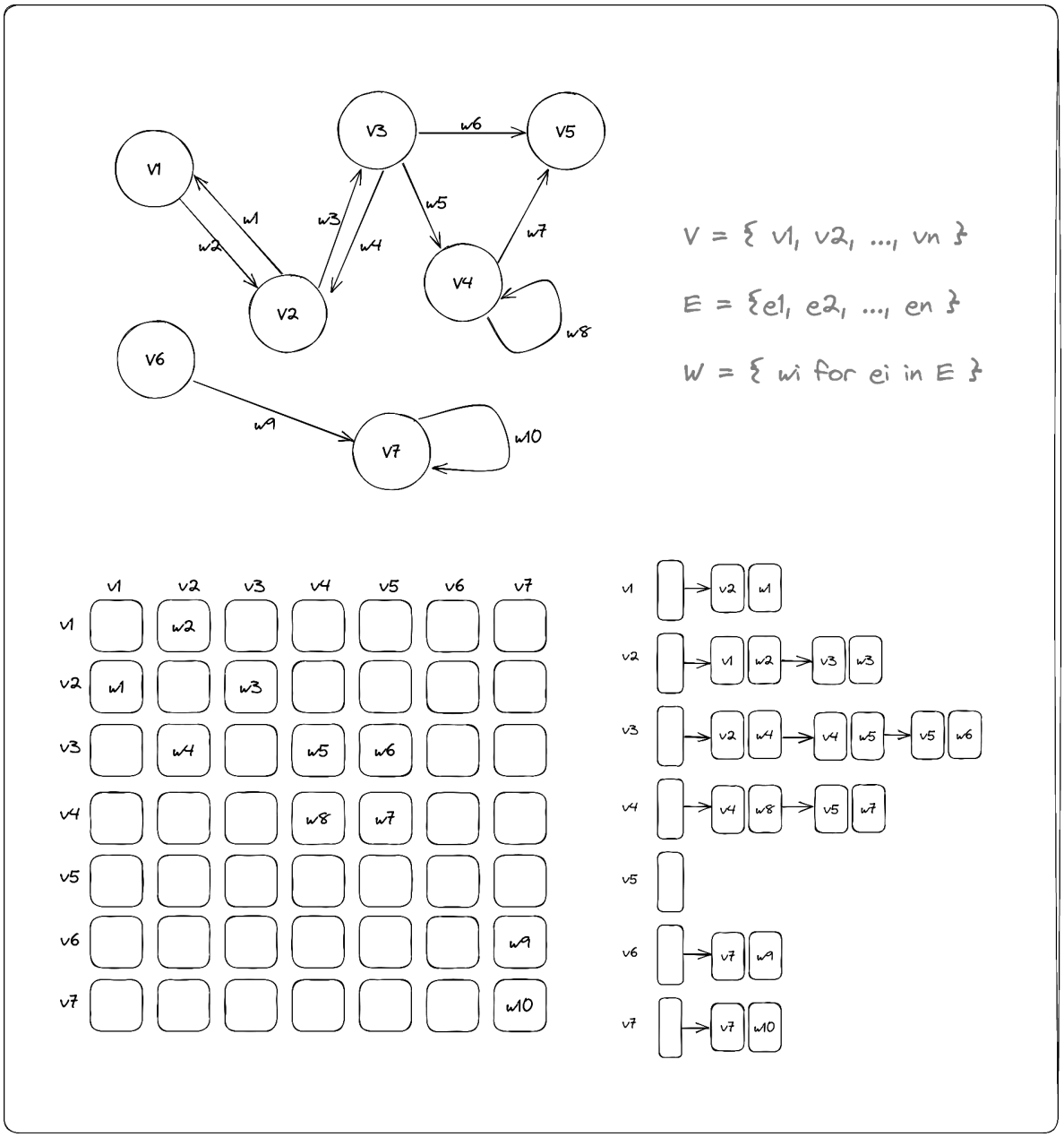
After analyzing the remaining ideas, the following implementations will be considered to solve each problem:

1. Scope between two entities: BFS Algorithm
2. Calculate shortest path: Dijkstra's Algorithm
3. Calculate best path traversing all entities: Minimum Spanning Tree

# Design

## 1. ADT Design

### 1.1 Graph ADT




Invariants

| **Natural Language** | **Logical Language** |
|---|---|
| The graph contains a set of vertices | $Let\ G\ be\ the\ graph, Let\ V\ be\ the\ set\ of\ vertices, \forall v / v \in V, V\ is\ a\ set\ of\ vertices$ |

| Natural Language | Logical Language |
|---|---|
| Each vertex has a unique identifier | $\forall v, u/v, u \text{ are vertices in } G, If\ v \neq u,$ $then\ v.id \neq u.id$ |
| The graph can contain multiple edges between two vertices | $\forall v, u/v, u \text{ are vertices in } G,$ $Let\ E \text{ be the set of edges between } v \text{ and } u,$ $\text{Then }\}$ |
| The graph can be directed or undirected | $\forall v, u/v, u \text{ are vertices in } G,$ $Let\ E \text{ be the set of edges between } v \text{ and } u,$ $If\ G \text{ is directed, then } E \text{ is directed}$ |
| The graph can be weighted or unweighted | $\forall v, u/v, u \text{ are vertices in } G,$ $Let\ E \text{ be the set of edges between } v \text{ and } u,$ $If\ G \text{ is weighted, then } E \text{ is weighted}$ |

### Primitive Operations

| Operation | Input | Output |
|---|---|---|
| `addVertex:` | graph, vertex | void |
| `removeVertex:` | graph, vertex | void |
| `addEdge:` | graph, vertex, vertex, weight | void |
| `removeEdge:` | graph, vertex, vertex | void |
| `hasVertex:` | graph, vertex | boolean |
| `hasEdge:` | graph, vertex, vertex | boolean |
| `getWeight:` | graph, vertex, vertex | weight (or null if edge doesn't exist) |
| `getAdjacentVertices:` | graph, vertex | list of vertices |
| `isEmpty:` | graph | boolean |

---

### addVertex( `graph`, `vertex` )

"Adds a new vertex `vertex` to the graph called `graph`."

{ Pre: `graph` is an initialized graph }

{ Pos: The vertex `vertex` has been added to the graph `graph` }

---

### removeVertex( `graph`, `vertex` )

"Removes the vertex `vertex` from the graph called `graph`."

{ Pre: `graph` is an initialized graph }

{ Pos: The vertex `vertex` has been removed from the graph `graph` }

---

### addEdge( `graph`, `source`, `destination`, `weight` )

"Adds a new edge with weight: `weight` between the vertices: `source` and `destination` in the graph called `graph`."

{ Pre: `graph` is an initialized graph, `source` and `destination` are vertices in `graph` }

{ Pos: An edge with weight `weight` has been added between `source` and `destination` in the graph `graph` }

---

### removeEdge( `graph`, `source`, `destination` )

"Removes the edge between the vertices: `source` and `destination` from the graph called `graph`."

{ Pre: `graph` is an initialized graph, `source` and `destination` are vertices in `graph` }

{ Pos: The edge between `source` and `destination` has been removed from the graph `graph` }

---

### hasVertex( `graph`, `vertex` )

"Checks if the graph `graph` contains the vertex `vertex`."

{ Pre: `graph` is an initialized graph }

{ Pos: `true` if the graph `graph` contains the vertex `vertex`, otherwise `false` }

---

### hasEdge( `graph`, `source`, `destination` )

"Checks if the graph contains an edge between the vertices: `source` and `destination`."

{ Pre: `graph` is an initialized graph, `source` and `destination` are vertices in `graph` }

{ Pos: `true` if the graph `graph` contains an edge between `source` and `destination`, otherwise `false` }

---

### getWeight( `graph`, `source`, `destination` )

"Returns the weight of the edge between the vertices: `source` and `destination` in the graph called `graph`."

{ Pre: `graph` is an initialized graph, `source` and `destination` are vertices in `graph` }

{ Pos: The weight of the edge between `source` and `destination` in the graph `graph`, or `null` if the edge doesn't exist }

---

### getAdjacentVertices( `graph`, `vertex` )

"Returns a list of vertices adjacent to the vertex `vertex` in the graph called `graph`."

{ Pre: `graph` is an initialized graph, `vertex` is a vertex in `graph` }

{ Pos: A list of vertices adjacent to the vertex `vertex` in the graph `graph` }

---

### isEmpty( `graph` )

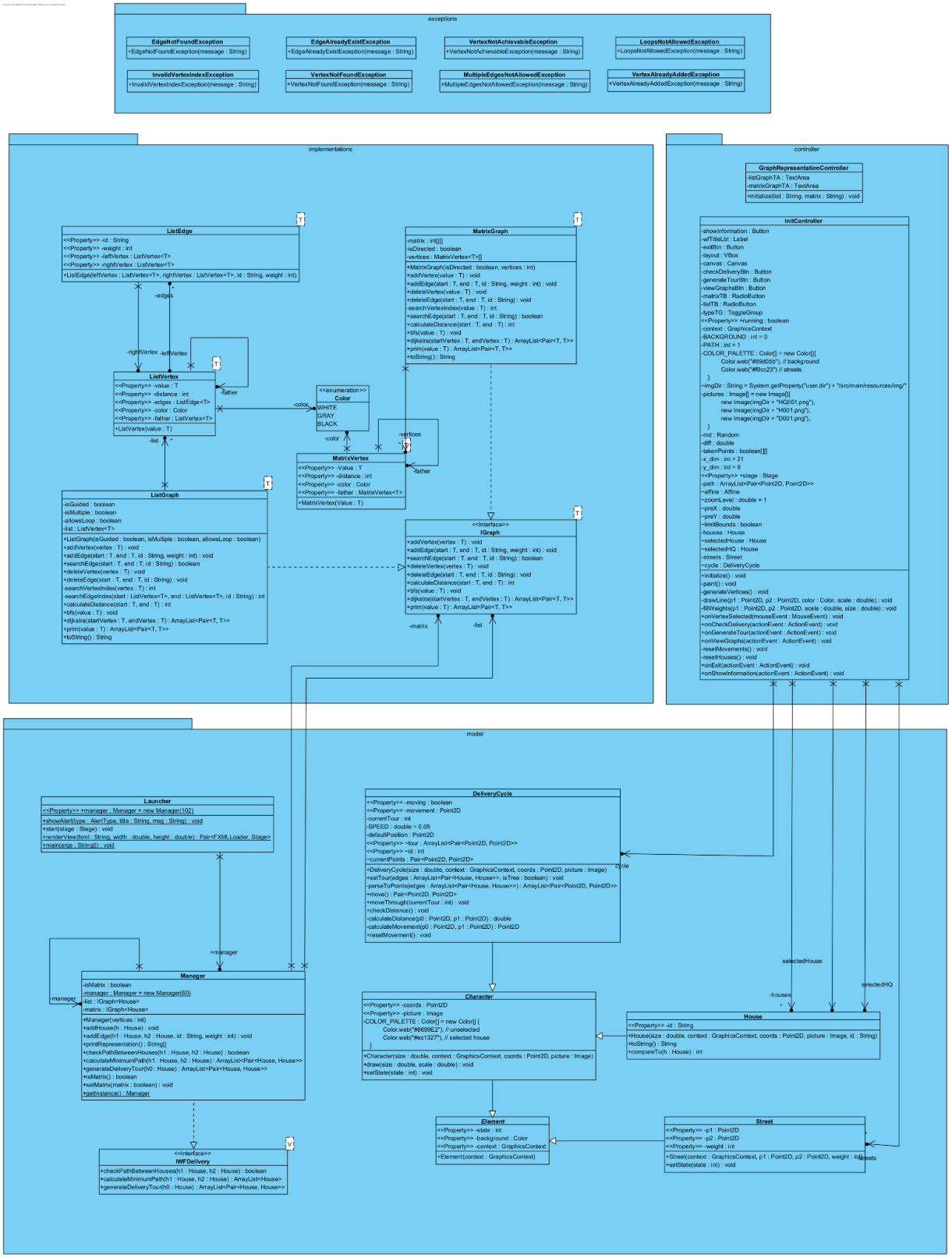
"It checks if the graph: `graph` is empty or not."

{ Pre: `graph` is an initialized graph }

{ Pos: `true` if the graph `graph` is empty, otherwise `false` }

# 2. UML Class Diagram

(Also in the repository doc folder)



# 3. Test Cases

## 3.1 List Graph

### Scenarios

| Name | Class | Scenario |
|---|---|---|
| newGraphSetup1() | TestListIGraph | Initialized empty graph with properties: guided, no multiple and without allowing loops. |
| newGraphSetup2() | TestListIGraph | Initialized graph with properties: guided, no multiple and without allowing loops. |
| newGraphSetup3() | TestListIGraph | Initialized graph with properties: no guided, multiple and allowing loops. |
| newGraphSetup4() | TestListIGraph | Initialized graph with properties: no guided, multiple and without allowing loops. |

### Tests

| **Test Objective**: Test the addVertex method adding one, multiple and duplicated vertices. | | | | |
|---|---|---|---|---|
| **Class** | **Method** | **Scenario** | **Inputs** | **Expected Result** |
| TestListIGraph | addVertexTest | newGraphSetup01() | Vertex 0 | The vertex 0 is added to the graph successfully. |
| TestListIGraph | addMultipleVertex | newGraphSetup01() | Vertices 0 to 7 | Vertices 0 to 7 are added to the graph successfully. |
| TestListIGraph | addDuplicatedVertexException | newGraphSetup01() | Vertex 0 | VertexAlreadyAddedException is thrown when adding a duplicate vertex. |

| **Test Objective**: Test the addEdge method adding one and multiple edges. Also, test graph restrictions. | | | | |
|---|---|---|---|---|
| **Class** | **Method** | **Scenario** | **Inputs** | **Expected Result** |
| TestListIGraph | addEdgeTest | newGraphSetup01() | Vertices 1 and 2, Edge "v1" with weight 4 | The edge "v1" is added between vertices 1 and 2 in the graph successfully. |
| TestListIGraph | addMultiplesEdgesTest | newGraphSetup01() | Vertices 0 to 7 and many edges | All edges are added between the specified vertices in the graph successfully. |

| | | | | |
|---|---|---|---|---|
| TestListIGraph | addEdgeNoMultipleException | newGraphSetup01() | Vertices 1 and 2, Edge "v1" with weight 4 | MultipleEdgesNotAllowedException is thrown when adding a duplicate edge. |

| **Test Objective**: Test the searchEdge method by validating existing and non-existing edges. Also, validate non-existing vertex exceptions. | | | | |
|---|---|---|---|---|
| **Class** | **Method** | **Scenario** | **Inputs** | **Expected Result** |
| TestListIGraph | validateExistingEdge | newGraphSetup02() | Search edge "v1" between vertices 1 and 5 | The edge "v1" exists between vertices 1 and 5 in the graph. |
| TestListIGraph | validateNonExistingEdge | newGraphSetup03() | Search edge "v1" between vertices 1 and 5 | The edge "v1" does not exist between vertices 0 and 5 in the graph. |
| TestListIGraph | validateEdgeBetweenNonExistingVerticesException | newGraphSetup04() | Search edge "v1" between vertices 2 and 8 | VertexNotFoundException is thrown when searching for an edge between non-existing vertices. |

| **Test Objective**: Test the deleteVertex method by deleting one and multiple vertices, including edge existence validations. | | | | |
|---|---|---|---|---|
| **Class** | **Method** | **Scenario** | **Inputs** | **Expected Result** |
| TestListIGraph | deleteVertexTest | newGraphSetup02() | Vertex 5 | The vertex 5 is deleted, and all associated edges are removed. Searching for those edges throws VertexNotFoundException. |
| TestListIGraph | deleteMultipleVertexTest | newGraphSetup02() | Vertices 0, 2 | Vertices 0 and 2 are deleted from the graph without throwing any exceptions. |
| TestListIGraph | deleteVertexWithLoops | newGraphSetup03() | Vertex 0 | The vertex 0 is deleted, and self-loops associated with it are removed. Searching for those edges throws VertexNotFoundException. |

| **Test Objective**: Test the deleteEdge method by deleting one and multiple edges. Also, validate non-existing edges. | | | | |
|---|---|---|---|---|
| **Class** | **Method** | **Scenario** | **Inputs** | **Expected Result** |
| TestListIGraph | deleteEdgeTest | newGraphSetup01() | Vertex 1, Vertex 2, Edge "v1" | The edge between Vertex 1 and Vertex 2 with label "v1" is deleted. Searching for the edge returns false. |

| | | | | |
|---|---|---|---|---|
| TestListIGraph | deleteManyEdgesOnMultipleGraph | newGraphSetup03() | Vertex 1, Vertex 0, Edge "v1" | The edge between Vertex 1 and Vertex 0 with label "v1" is deleted. Searching for the edge returns false. |
| TestListIGraph | deleteNonExistingEdgeException | newGraphSetup02() | Vertex 1, Vertex 0, Edge "v1" | Deleting the edge between Vertex 1 and Vertex 0 with label "v1" throws an EdgeNotFoundException. |

| **Test Objective**: Test BFS method by calculating distance between two vertices. Also, validate non existing vertices. | | | | |
|---|---|---|---|---|
| **Class** | **Method** | **Scenario** | **Inputs** | **Expected Result** |
| TestListIGraph | calculateDistanceOnNonGuidedGraphUsingBfs | newGraphSetup03() | Vertex 3, Vertex 2 | The distance between Vertex 3 and Vertex 2 is calculated using BFS and is equal to 3. |
| TestListIGraph | calculateDistanceOnGuidedGraphUsingBfs | newGraphSetup02() | Vertex 0, Vertex 6 | The distance between Vertex 0 and Vertex 6 is calculated using BFS and is equal to 2. |
| TestListIGraph | calculateDistanceOfNonAchievableVerticesException | newGraphSetup02() | Vertex 4, Vertex 2 | Calculating the distance between Vertex 4 and Vertex 2 throws a VertexNotAchievableException. |

| **Test Objective**: Test Dijkstra's method by calculating the shortest path between two vertices. Also, validate if both vertices are reachable. | | | | |
|---|---|---|---|---|
| **Class** | **Method** | **Scenario** | **Inputs** | **Expected Result** |
| TestListIGraph | dijkstraOnDirectedGraph | newGraphSetup02() | Vertex 6, Vertex 7 | Dijkstra's algorithm is executed on the directed graph, and the result is an ArrayList of pairs representing the shortest path from Vertex 6 to Vertex 7. The result is validated against the expected path [6, 5, 5, 3, 3, 7]. |
| TestListIGraph | dijkstraOnMultipleSimpleGraph | newGraphSetup04() | Vertex 1, Vertex 4 | Dijkstra's algorithm is executed on the multiple simple graphs, and the result is an ArrayList of pairs representing the shortest path from Vertex 1 to Vertex 4. The result is validated against the expected path [1, 3, 3, 0, 0, 4]. |
| TestListIGraph | dijkstraBetweenNonReachableVerticesException | newGraphSetup03() | Vertex 0, Vertex 6 | Executing Dijkstra's algorithm between non-reachable vertices (Vertex 0 and Vertex 6) throws a VertexNotAchievableException. |

| **Test Objective**: Test prim method by creating MST of a graph. Also, validate loops exception. | | | | |
|---|---|---|---|---|
| **Class** | **Method** | **Scenario** | **Inputs** | **Expected Result** |
| TestListIGraph | createMinimumSpanTreeOnRelatedGraph | newGraphSetup04() | Vertex 0 | A minimum spanning tree is created using Prim's algorithm on the related graph starting from Vertex 0. The result is an ArrayList of pairs representing the edges in the minimum spanning tree. The result is validated to have a size of 7. |
| TestListIGraph | createMinimumSpanTreeOnNonRelatedGraph | newGraphSetup02() | Vertex 0 | A minimum spanning tree is created using Prim's algorithm on the non-related graph starting from Vertex 0. The result is an ArrayList of pairs representing the edges in the minimum spanning tree. The result is validated to have a size of 7. |
| TestListIGraph | MinimumSpanTreeOnGraphWithLoopsException | newGraphSetup03() | Vertex 0 | Creating a minimum spanning tree on a graph with loops (newGraphSetup03()) throws a LoopsNotAllowedException. |

## 3.2 Matrix Graph

### Scenarios

| Name | Class | Scenario |
|---|---|---|
| setUpGraph | TestMatrixGraph | Initializes a new MatrixGraph instance with size 10. |
| addFiveVertexes | TestMatrixGraph | Adds five vertices ("a", "b", "c", "d", "e") to the graph created in setUpGraph. |
| intGraphSetup1 | TestMatrixGraph | Initializes a new MatrixGraph instance with size 8 and adds vertices (0-7). Then, it adds several edges between the vertices. |

### Tests

| **Objective:** To test the addition of a new edge to the graph | | | | |
|---|---|---|---|---|
| Class | Method | Scenario | Inputs | Expected Result |
| MatrixGraph Test | addMultipleEdges() | addFiveVertexes(), addEdge() | New edges between existing vertices with unique label and weight | The edges are successfully added to the graph. The searchEdge() method returns the expected results. |
| MatrixGraph Test | addSingleEdgeWithLoopException() | addFiveVertexes(), addEdge() | New edge with the same source and destination vertex | LoopsNotAllowedException is thrown |
| MatrixGraph Test | addMultipleConnectionsTwoVertex() | addFiveVertexes(), addEdge() | Multiple edges with the same source and destination vertex | MultipleEdgesNotAllowedException is thrown |

| **Objective:** To test the correct delete of a vertex from the graph | | | | |
|---|---|---|---|---|
| Class | Method | Scenario | Inputs | Expected Result |
| MatrixGraph Test | deleteVertexMultipleConnections() | addFiveVertexes(), addEdge(), deleteVertex() | Existing vertex with multiple connections | The vertex and its associated edges are successfully deleted. The searchEdge() method returns the expected results. The VertexNotFoundException is thrown for each deleted edge. |
| MatrixGraph Test | deleteVertexSingleConnection() | addFiveVertexes(), addEdge(), deleteVertex() | Existing vertex with a single connection | The vertex and its associated edge are successfully deleted. The searchEdge() method returns the expected results. The VertexNotFoundException is thrown for the deleted edge. |
| MatrixGraph Test | deleteVertexNoConnections() | addFiveVertexes(), addEdge(), deleteVertex() | Non-existent vertex with attempted connection | The VertexNotFoundException is thrown when attempting to add an edge to the deleted vertex. |

| **Objective:** To test the deletion of a edge from the graph | | | | |
|---|---|---|---|---|
| Class | Method | Scenario | Inputs | Expected Result |
| MatrixGraph Test | deleteMultipleEdges() | addFiveVertexes(), addEdge(), deleteEdge() | Existing edges to be deleted | The specified edges are successfully deleted. The searchEdge() method returns the expected results. |
| MatrixGraph Test | deleteNonExistingEdge() | addFiveVertexes(), addEdge(), deleteEdge() | Non-existing edge to be deleted | The EdgeNotFoundException is thrown when attempting to delete the non-existing edge. |
| MatrixGraph Test | deleteMultipleNonConnectedEdges() | addFiveVertexes(), addEdge(), deleteEdge() | Existing non-connected edges to be deleted | The specified edges are successfully deleted. The searchEdge() method returns the expected results. |

| **Objective:** To test the if two vertex are related, and the path between them | | | | |
|---|---|---|---|---|
| Class | Method | Scenario | Inputs | Expected Result |
| MatrixGraph Test | calculateDistanceOnGuidedGraphUsingBfs() | intGraphSetup1(), calculateDistance() | Source and destination vertices | The calculated distance using BFS matches the expected distance. |
| MatrixGraph Test | calculateDistanceOfNonAchievableVerticesException() | intGraphSetup1(), calculateDistance() | Source and destination vertices | The VertexNotAchievableException is thrown when attempting to calculate the distance between non-achievable vertices. |
| MatrixGraph Test | dijkstraOnDirectedGraph() | intGraphSetup1(), dijkstra() | Source and destination vertices | The calculated shortest path using Dijkstra's algorithm matches the expected path. |

| **Objective:** To test the correct creation of a minimum spread tree along different graphs | | | | |
|---|---|---|---|---|
| Class | Method | Scenario | Inputs | Expected Result |
| MatrixGraph Test | createMinimumSpreadTree() | addFiveVertexes(), addEdge(), prim() | Starting vertex, graph structure | The generated minimum spread tree contains the expected number of edges. |
| MatrixGraph Test | twoBranchesConnected() | addFiveVertexes(), | Starting vertex, graph structure | The generated minimum spread tree |

| **Objective:** To test the correct creation of a minimum spread tree along different graphs | | | | |
|---|---|---|---|---|
| Class | Method | Scenario | Inputs | Expected Result |
| | | addVertex(), addEdge(), prim() | | connects the two branches and contains the expected number of edges. |
| MatrixGraph Test | mstZigZag() | setUpGraph( ), addVertex(), addEdge(), prim() | Starting vertex, graph structure | The generated minimum spread tree follows the zigzag pattern and contains the expected number of edges. |

### 3.3 Manager

#### Scenarios

| Method Name | Class | Scenario |
|---|---|---|
| initEmptyGraphs | ManagerTest | Initial setup for an empty graph |
| initHouses | ManagerTest | Initialize houses with IDs 1 to 14 and add them to the graph |
| sampleGraphSetup | ManagerTest | Set up a sample graph with connected houses and edges |

#### Tests

| Test Objective: Test the addHouse function by adding houses to an empty graph and an existing graph. | | | | |
|---|---|---|---|---|
| Class | Method | Scenario | Inputs | Expected Result |
| ManagerTest | addHousesEmptyGraph | initHouses(), addHouse() | Houses with IDs 1 to 10 added to an empty graph | All houses are successfully added to the graph |
| ManagerTest | addNewHouseExistingGraph | sampleGraphSetup(), addHouse() | New house with ID 15 added to an existing graph | The new house is successfully added to the graph |
| ManagerTest | checkHouseLimit | - | 128 houses added to the graph | All 128 houses are successfully added to the graph |

| Test Objective: Test the addEdge function where there are and when they are not connections. | | | | |
|---|---|---|---|---|
| Class | Method | Scenario | Inputs | Expected Result |
| ManagerTest | addHouseConnection | sampleGraphSetup(), addEdge() | Edge added between houses 3 and 7 | Connection between houses 3 and 7 is established |
| ManagerTest | addExistingConnection | sampleGraphSetup(), addEdge() | Existing edge between houses 3 and 6 is added | MultipleEdgesNotAllowedException is thrown |
| ManagerTest | addEdgesNonExistentHouses | initHouses(), addEdge() | Edge added between non-existent houses 3 and 6 | VertexNotFoundException is thrown |

| Test Objective: Test the checkPathBetweenHouses function by verifying paths between houses in the graph. | | | | |
|---|---|---|---|---|
| Class | Method | Scenario | Inputs | Expected Result |
| ManagerTest | checkNonExistentPath | sampleGraphSetup(), checkPathBetweenHouses() | Path checked between houses 10 and 14, which are not connected | The path between houses 10 and 14 does not exist |
| ManagerTest | checkExistentLongPath | sampleGraphSetup(), checkPathBetweenHouses() | Path checked between houses 10 and 9, which are connected | The path between houses 10 and 9 exists |
| ManagerTest | checkEdgeAsPath | sampleGraphSetup(), checkPathBetweenHouses() | Path checked between houses 1 and 3, which are connected | The path between houses 1 and 3 exists |

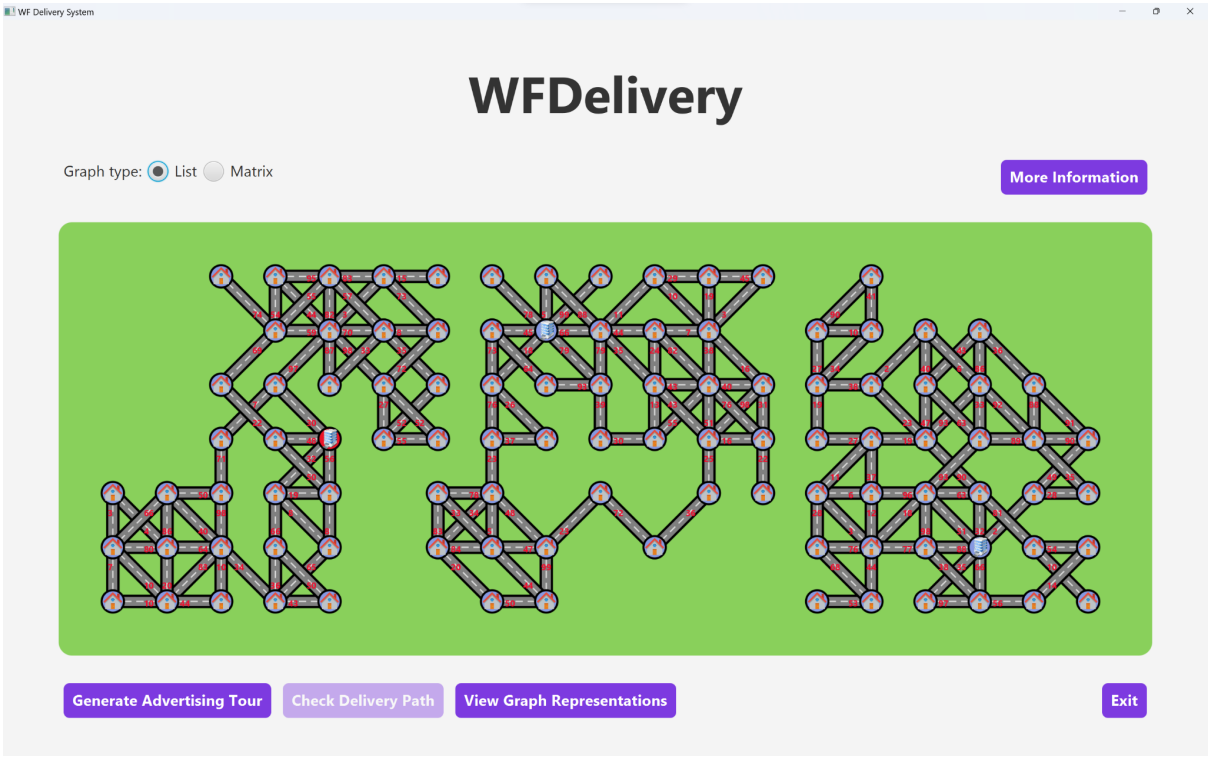
| Test Objective: Test the calculateMinimumPath function by calculating the minimum path between houses in the graph. | | | | |
|---|---|---|---|---|
| Class | Method | Scenario | Inputs | Expected Result |
| ManagerTest | minimumPathIsLonger | sampleGraphSetup(), calculateMinimumPath() | Minimum path calculated between houses 1 and 3 | The minimum path contains 3 edges |
| ManagerTest | minimumPathIsEdge | sampleGraphSetup(), calculateMinimumPath() | Minimum path calculated between houses 1 and 10 | The minimum path contains 1 edge |
| ManagerTest | minimumPathDoesNotExist | sampleGraphSetup(), calculateMinimumPath() | Minimum path calculated between houses 1 and 13 | VertexNotAchievableException is thrown |

| Test Objective: Test the generateDeliveryTour function by generating a delivery tour for houses in the graph. | | | | |
|---|---|---|---|---|
| Class | Method | Scenario | Inputs | Expected Result |
| ManagerTest | unconnectedHouseTour | sampleGraphSetup(), generateDeliveryTour() | Delivery tour generated for a house not connected to the graph | The delivery tour does not contain edges to the unconnected house |
| ManagerTest | calculateTourLinear | sampleGraphSetup(), generateDeliveryTour() | Delivery tour generated for a linear graph | The delivery tour visits all houses in the correct order |
| ManagerTest | calculateTourCheckVisited | sampleGraphSetup(), generateDeliveryTour() | Delivery tour generated for a graph with a starting house | The delivery tour visits all other houses except the starting house |

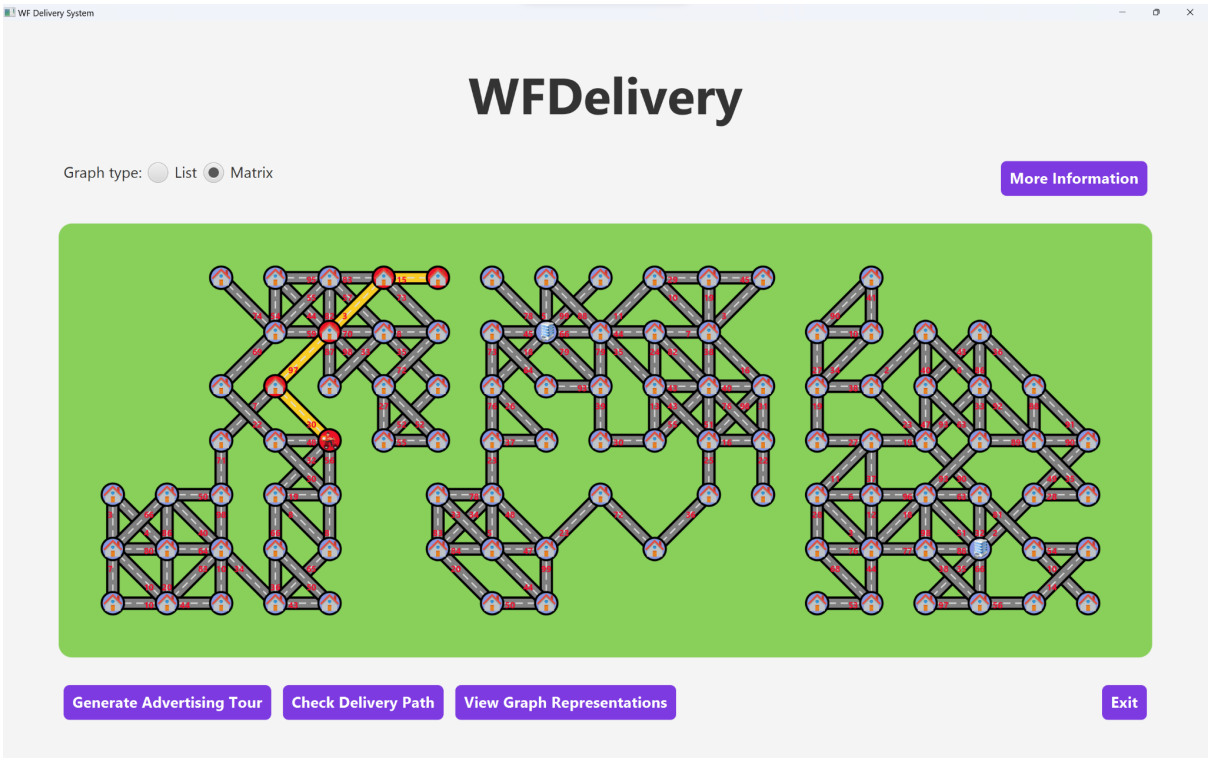
# Implementation

Link to the hub: [JMMA86/WF-Delivery-System (github.com)]

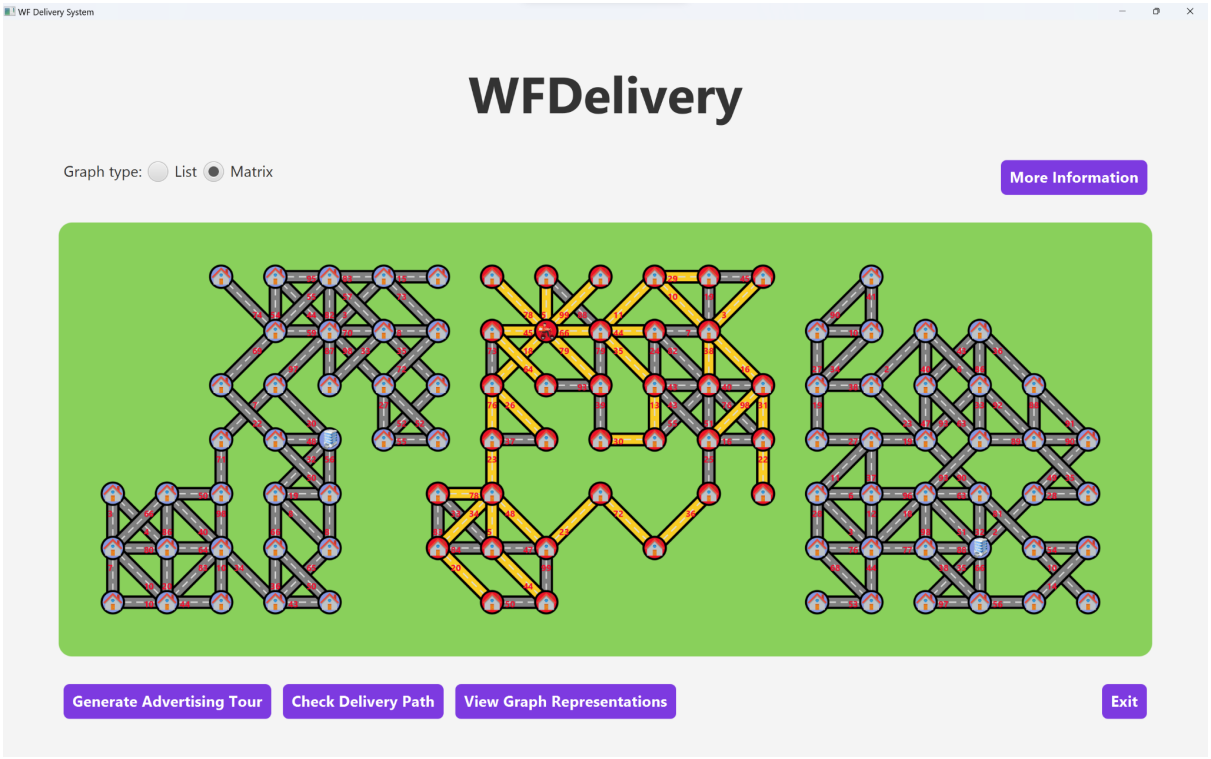
## Initial Interface




## Calculate Path



Generate Tour:




Zoom in: