INTEGRATIVE TASK II
ENGINEERING DESIGN METHOD

Developed by:

Vanessa Sánchez Morales (A00397949)

Luis Manuel Rojas Correa (A00399289)

Gabriel Escobar (A00399291)

Icesi University

Professor: Dr. Marlon Gomez Victoria

Santiago de Cali,

Republic of Colombia

November 3rd, 2023

ENGINEERING DESIGN METHOD

Marlon Mania

For our second integrative task we have been asked to develop a game. This game must be implemented using graphs. To accomplish this task, we have decided to develop Marlon Mania, a single player a game.

In this game players will have to go deep into the sewage system to solve a massive problem caused by a huge earthquake. This earthquake has destroyed some crucial parts of the sewage system leaving most of the population without access to water. To solve this problem the player must be able to connect two sets of pipes that were disconnected due to the earthquake. The starting point will be known as the source and the arrival point will be known as the drainage.

The idea is that the player achieves this connection using the least amount of pipes and for the most serious damage, the player must place the pipes so that the water takes the shortest time to arrive. In order to do this, the player is able to place 3 different types of pipes: the first one is the horizontal pipe that allows water to flow form right to left or vice versa, the second one is the vertical pipe that allows water to flow up and down or vice versa, and finally the third one is the circular pipe, which allows to change the flow of water from up or down to left or right. Lastly, players must know that the game calculates the score based on how effective the player's solution is.

For the requirements of the project, look at the requirements analysis document.

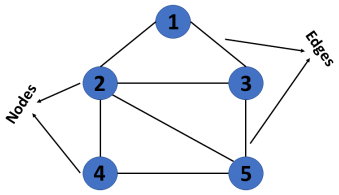
PHASE 1: IDENTIFICATION OF THE PROBLEM (Software Requirement Specification-SRS)

FASE 2: COMPILATION OF NECESSARY INFORMATION

Important Terminology

Graph:

Graphs in data structures are non-linear data structures made up of a finite set of nodes or vertices and the edges that connect them. Graphs in data structures are used to address real-world problems in which it represents the problem area as a network. The graph is denoted by G(E, V).



- Vertex:
  The vertices are the fundamental units of the graph, sometimes also denominated as node. Every node/vertex can be labeled or unlabeled.

- Edge:
  Edges connect two nodes of the graph in any possible way. Depending on the type of graph, the set of vertices it connects must be ordered or not. Sometimes known as arcs, they can be labeled/unlabeled (also determined as weight of the edge).

- Path:
  A path (of length n) in an (undirected) graph G is a sequence of vertices {v0, v1, ..., vn-1, vn} such that there is an edge between vi and vi+1 ∀i ∈ [0..n-1] along the path.

- Adjacency:
  A vertex is said to be adjacent to another vertex if there is an edge in common connecting them.

Adjacency List:

An adjacency List (AL) is an array of V lists, one for each vertex (usually in increasing vertex number) where for each vertex i, AL[i] stores the list of i's neighbors or adjacent nodes. For weighted graphs, the list can store pairs of (neighbor vertex number, weight of this edge) instead.

Adjacency Matrix:

The adjacency matrix of a simple labeled graph G is the matrix *A* with A[[*i*,*j*]]=1 or 0 according to whether the vertex *vj*, is adjacent to the vertex *v*j or not. For simple graphs without self-loops, the adjacency matrix has 0 s on the diagonal. For undirected graphs, the adjacency matrix is symmetric.

$$A_{ij} = \begin{cases} 1 \; if \; (v_i, v_j) \; is \; an \; edge \; in \; G \\ 0 \; otherwise \end{cases}$$

User Interface:

User Interface (UI) is the point at which the human interacts with a computer, website and/or application. The UI must be intuitive and ease the user's experience with the usage of the software product, requiring minimum effort on the user's part to receive the maximum desired outcome.

Dijkstra's Algorithm:

Algorithm for finding the shortest path from a starting node to a target node in a weighted graph. The algorithm creates a tree of shortest paths from the starting vertex, the source, to all other points in the graph.

Minimum Spanning Tree:
A spanning tree of a graph is a collection of connected edges that include every vertex in the graph, but that do not form a cycle. The Minimum Spanning Tree is the one whose cumulative edge weights have the smallest value possible.

Pipe:

Connecting lines or tubes between (for the context of the program) other pipes. They act as edges between two of the vertices of the graph that represent the matrix of 8x8 that will be the playground for each of the user's game.

They are classified in 4 different types (for more characterization of each one of them, look at phase 1: SRS).

- Default (x)
- Vertical (| |)
- Horizontal (=)
- Circular (°)

PHASE 3: RESEARCH OF CREATIVE SOLUTIONS

Alternative 1

1. **User Interface:**
   JavaFX is a modern library to create user interaction interfaces and can be used in various platforms such as Windows, macOS and Linux. Interactive Graphics can be created with this library.

2. **Game:**
   Marlon Mania is a game that consists of a sewer system simulation. In this game, the player can locate three different types of "pipes" within an 8x8 board/matrix, with the objective of connecting the "water source" to the "draining pipe". The users can also view a players' ranking according to the scores gained, with each of the players' names. There are two levels of difficulty: easy, where the player must complete the connect the water source to the draining pipe; and hard, where the player, to get points, must connect the last-mentioned pipes by using the shortest path.

3. **Versions of graph:**
   For the first version of the graph, an adjacency list is considered.
   An adjacency List (AL) is a representation of a graph through an array of V lists, one for each vertex (usually in increasing vertex number) where for each vertex i, AL[i] stores the list of adjacent nodes. For weighted graphs, the list can store pairs of (neighbor vertex number, weight of this edge) instead.

   For the second version of the graph, an adjacency matrix is considered.
   The adjacency matrix of a simple labeled graph G is the matrix A with A[[i,j]]=1 or 0 according to whether the vertex vj, is adjacent to the vertex vj or not.

4. **Graph Algorithms:**
   For the first graph algorithm to be used, a graph-traversal algorithm is considered, specifically DFS.
   Depth-first search (DFS) is an algorithm for searching a graph or tree data structure. The algorithm starts at the root (top) node of a tree and goes as far as it can down a given branch (path), then backtracks until it finds an unexplored path, and then explores it. The algorithm does this until the entire graph has been explored.

   For the second graph algorithm to be used, a Single-Source Shortest Path (SSSP) algorithm (given a source vertex it finds the shortest path from the source to all other vertices) is considered, specifically Dijkstra's algorithm.
   Dijkstra's Algorithm works on the basis that, in a Graph G={A,B,C,D}, any subpath B -> D of the shortest path A -> D between vertices A and D is also the shortest path between vertices B and D. Dijkstra used this property in the opposite direction overestimating the distance of each vertex from the starting vertex. Then each node and its neighbors are visited to find the shortest subpath to those neighbors.

Alternative 2

1. **User Interface:**
   It will be created using the GUI package, composed of the two classes AWT and Swing, which are rich in components and containers (Component, container, Jcomponent, Jframe, Jdialog, JApplet, Jpanel, Graphics), whose tools and classes are interactable and simple, making the user interface creating process more practical.

2. **Game:**
   Pipe-Mania is a game that consists of a sewer system simulation. In this game, the player can locate three different types of "pipes" within an 8x8 board/matrix, with the objective of connecting the "water source" to the "draining pipe". The users can also view a players' ranking according to the scores gained, with each of the players' names.
   The player, to get points, must connect the water source to the draining pipe.

3. **Versions of Graph:**
   For the first version of a graph, an incidence matrix is considered.
   The incidence matrix A of an undirected graph has a row for each vertex and a column for each edge of the graph. The element A[[i,j]]=1 if the ith vertex is a vertex of the jth edge, otherwise A[[i,j]]=0.

   For the second version of a graph, an edge list is considered.
   An edge list is a list or array of all the edges in a graph. The underlying data structure for keeping track of all the nodes and edges is a single list of pairs. Each pair represents a single edge and is comprised of the two unique IDs of the nodes involved. Each line/edge in the graph gets an entry in the edge list, and that single data structure then encodes all nodes and relationships.

4. **Graph Algorithms:**
   For the first graph algorithm to be used, a graph-traversal algorithm is considered, specifically BFS.
   Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node.

   For the second graph algorithm to be used, a Shortest Path (SP) algorithm (finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized) is considered, specifically Floyd-Warshall's algorithm.
   It computes the computes the shortest distances between every pair (or a given pair) of vertices in the input graph.

Alternative 3

1. **User Interface:**
   It will be created by console, making the effort of using different java functionalities and colors to make the user experience more user friendly, whilst making it easier for the programmer to implement their data structures, logic and functionality of the program with more flexibility.

2. **Game:**
   Pipe-Mania is a game that consists of a sewer system simulation. In this game, the player can locate three different types of "pipes" within an 6x6 board/matrix, with the objective of connecting the "water source" to the "draining pipe".
   The player, to get points, must connect the water source to the draining pipe.

3. **Versions of Graph:**
   For the first version of a graph, an edge list is considered.
   An edge list is a list or array of all the edges in a graph. The underlying data structure for keeping track of all the nodes and edges is a single list of pairs. Each pair represents a single edge and is comprised of the two unique IDs of the nodes involved. Each line/edge in the graph gets an entry in the edge list, and that single data structure then encodes all nodes and relationships.

   For the second version of a graph, an adjacency list is considered.
   An adjacency List (AL) is a representation of a graph through an array of V lists, one for each vertex (usually in increasing vertex number) where for each vertex i, AL[i] stores the list of adjacent nodes. For weighted graphs, the list can store pairs of (neighbor vertex number, weight of this edge) instead.

4. **Graph Algorithms:**
   For the graph algorithms, an alternative is to use MST (Minimum Spanning Tree) algorithms. In the first place is Kruskal's algorithm. In Kruskal's algorithm, all edges of the given graph are sorted in increasing order. Then new edges and nodes keep being added in the MST if the newly added edge does not form a cycle. It picks the minimum weighted edge at first and the maximum weighted edge at last.

   For the second algorithm, Prim's is an alternative.
   The algorithm starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, and the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

PHASE 4: TRANSITION FROM IDEA FORMULATION TO PRELIMINARY DESIGNS

Alternative 1 (Chosen):

1. **User Interface:**
   - JavaFx allows the use of more animation and visual functionalities.
   - It requires more time and effort to learn how to use said library.
   - JavaFx usage applies the model, control, view design pattern that guarantees the application of correct programming practices.

2. **Game:**
   - Marlon Mania´s game needs the usage of graphs to verify the route of pipes made by the player
   - It is an entertaining game that requires design principles and algorithmic logic to be developed, following, that way, with the principles of said integrative task.
   - The game, by having an 8x8 board, accomplishes the non-functional requirement of using a graph with more than 50 vertices (as it has 64 nodes).
   - It has different levels and game complexities that not only enrich the gaming experience, but also apply different graph algorithms that are mandatory (non-functional requirement).

3. **Versions of Graph:**
   - An adjacency list clearly states the relations between the vertices and the edges that those relationships form. It also facilitates the management of the vertices, where each node is managed as an object, which will have control and information of the edges formed with itself.
   - An adjacency matrix clearly states the relationships between each pair of vertices. Although its methods and the structure itself entail a bigger spatial complexity, the management of the vertices and edges, as they are described by 1 and 0´s in the case of the non-weighted graphs, and in contrast with other structures, it also eases the visualization and storage of weighted graphs.

4. **Graph Algorithm:**
   - DFS is an algorithm that can be easily implemented and adapted to the developer's and the project's necessities. It also solves one of the game's functionalities, which is verifying whether the water flow is correctly built. In other words, it efficiently checks whether two nodes or the nodes of a graph are connected.
   - Although Dijkstra's Algorithm algorithmic complexity is significant, it solves a NP-Complete problem and solves one of the functionalities of the game, which is finding the shortest/less weighted path in a weighted graph.

Alternative 2 (Chosen):

1. **User Interface:**
   - The GUI package can be easily implemented in the development of a Java application.
   - The package and its containers (Component, container, Jcomponent, Jframe, Jdialog, JApplet, Jpanel, Graphics) are practical when used and accomplish the development of a user-friendly application.

2. **Game:**
   - Pipe Mania´s game needs the usage of graphs to verify the route of pipes made by the player
   - It is an entertaining game that requires design principles and algorithmic logic to be developed, following, that way, with the principles of said integrative task.
   - The game, by having an 8x8 board, accomplishes the non-functional requirement of using a graph with more than 50 vertices (as it has 64 nodes).

3. **Versions of Graph:**
   - An incidence matrix states the relationships between a vertex and a specific edge. Although its methods and the structure itself entail a bigger spatial complexity, the management of the vertices and edges, as they are described by 1 and 0´s.
   - An edge list contains the information of all the relationships between each of the vertices of the graph (the edges). Although its spatial complexity is smaller than other versions of the graph, the methods that it entails require bigger algorithmic time and the vertices are not managed by themselves but only as pairs.

4. **Graph Algorithm:**
   - BFS is an algorithm that can be easily implemented. It also solves one of the game's functionalities, which is verifying whether the water flow is correctly built. In other words, it efficiently checks whether two specific and/or given vertices or the nodes of a graph are connected.
   - Floyd-Warshall's algorithm, like Dijkstra's, solves a NP-Complete problem and solves one of the functionalities of the game, which is finding the shortest/less weighted path in a weighted graph. However, their differences rely on the fact that Floyd-Warshall's returns the shortest path (not checks) and has bigger algorithmic complexity.

Alternative 3 (Discarded):

1. **User Interface:**
   - Although an application that runs by console is easier to implement and is more flexible to changes, it lacks design, doesn´t require the application of design patterns and is not user-friendly. Meaning, it doesn't satisfy the non-functional requirement of the project of having an user interface and it affects the gaming experience.
   - Therefore, this option is completely discarded.

2. **Game:**
   - The version of Pipe Mania described in this alternative doesn't follow many of the requirements of the projects.
   - Firstly, by having a 6x6 game board, it does not satisfy the non-functional requirement of the project of implementing a graph with more than 50 nodes/vertices.
   - Secondly, by not counting with any levels of difficulty, this solution does not require the implementation of different graph algorithms, which is also mandatory in the development of the game.
   - Finally, this version of the game does not count with any levels of difficulty or points ranking that motivates the player or enrich their gaming experience.

- Therefore, this option is discarded.

3. **Versions of Graph:**
   - An edge list contains the information of all the relationships between each of the vertices of the graph (the edges). Although its spatial complexity is smaller than other versions of the graph, the methods that it entails require bigger algorithmic time and the vertices are not managed by themselves but only as pairs.
   - An adjacency list clearly states the relations between the vertices and the edges that those relationships form. It also facilitates the management of the vertices, where each node is managed as an object, which will have control and information of the edges formed with itself.
   - Although these versions of graph are possible candidates for their implementation, it has already been proposed in other alternatives and does not bring anything new to the brainstorming process.

4. **Graph Algorithm:**
   - MST (Minimum Spanning Tree), although they can be helpful in various applications, they do not achieve the purpouse of this project, which is the one of checking the connection between vertices of a graph and finding the less weighted path.
   - In conclusion, this option is discarded.

PHASE 5: EVALUATION AND SELECTION OF THE SOLUTION

5.1 Criteria Evaluation Definition in terms of Quality:

CRITERIA QUALITY EVALUATION DEFINITION		
Criteria	**Definition**	**Evaluation Scale**
Functionality	Assess how well the software meets the required functionality, including precision, adequacy, interoperability, conformance, and security.	-1: Does not meet functionality requirements -2: Partially meets functionality requirements -3: Moderately meets functionality requirements -4: Adequately meets functionality requirements -5: Fully and precisely meets functionality requirements
Reliability	Evaluate the software's reliability in terms of maturity, error tolerance, and recoverability.	-1: Highly unreliable and lacks maturity -2: Moderately reliable with some maturity -3: Reasonably reliable with good maturity -4: Highly reliable and mature -5: Exceptionally reliable and mature
Usability	Assess the usability of the software, including comprehensibility, learnability, operability, and attractiveness.	-1: Highly unusable, poor user experience -2: Moderately usable but needs improvement -3: Reasonably usable with a good user experience -4: Highly usable with an excellent user experience -5: Exceptionally usable and provides an outstanding user experience
Efficiency	Evaluate the software's efficiency in terms of response time (algorithmic complexity), memory usage, and resource utilization.	-1: Highly inefficient and consumes excessive resources -2: Moderately efficient but could be more resource-friendly -3: Reasonably efficient with acceptable resource usage -4: Highly efficient with minimal resource consumption -5: Exceptionally efficient, making optimal use of resources

5.1.2 Evaluation of the Chosen Alternatives in terms of Quality:

ALTERNATIVE 1 EVALUATION				
Requisit	**Functionality Punctuation**	**Reliability Punctuation**	**Usability Punctuation**	**Efficiency Punctuation**
1. User Interface	5	4	4	5
2. Game	5	5	5	4
3. Versions of Graph	5	5	5	5 (adjacency list) + 3 (adjacency matrix)
4. Graph Algorithms	5	5	5	4 (DFS) + 4 (Dijkstra's)
Total Punctuation	**20**	**19**	**19**	**25**

ALTERNATIVE 2 EVALUATION				
Requisit	**Functionality Punctuation**	**Reliability Punctuation**	**Usability Punctuation**	**Efficiency Punctuation**
1. User Interface	3	3	3	3
2. Game	3	4	4	4
3. Versions of Graph	3	4	2	3 (incidence matrix) + 2 (edge list)
4. Graph Algorithms	2	4	4	4 (BFS) + 1 (Floyd-Warshall's)
Total Punctuation	**11**	**15**	**13**	**17**

Criteria	Functionality	Reliability	Usability	Efficiency	**Total**
Alternative 1	**20**	**19**	**19**	**25**	**83**
Alternative 3	**11**	**15**	**13**	**17**	**56**

5.2 Criteria Evaluation Definition in terms of Algorithm and Spatial Complexity (Worst Case):

Font: Medium
https://medium.com/@Hollyzhou/data-structure-the-big-o-notation-e3e2405bb8eb

BIG (O) EVALUATION SCALE		
Complexity	**Definition**	**Punctuation**
Cubic	$O(n^3)$	0
Quadratic	$O(n^2)$	1
Lineal	O(n)	2
Log-linear Complexity	O(n Log(n))	3
Logarithmic	O(Log(n))	4
Constant	O(1)	5

ALGORITHMICAL EVALUATION CRITERIA		
Criteria	**Definition**	**Evaluation Scale**

Verifying sewer system		
Checking the water flow for Easy Level	This criterion evaluates how efficiently the system checks that the pipes are correctly connected, meaning, that the vertices are connected. For this functionality, the DFS or BFS algorithm would be used	BIG O CRITERIA
Checking the water flow for the Hard Level	This criterion evaluates how efficiently the system checks that the pipes are correctly connected, meaning, that the vertices are connected. Besides, it also guarantees that the path built is the most efficient one. For this functionality, Dijkstra's or Floyd-Warshall's algorithm would be used.	BIG O CRITERIA

5.2.1 Evaluation of Structures in Chosen Alternatives in terms of Algorithm Complexity (Worst Case):

Alternative 1 Evaluation

ALTERNATIVE 1 (DFS & Dijkstra's Algorithm) Verifying sewer system	
Criteria	**Evaluation Scale (Big-O)**
Checking the water flow for Easy Level	2 [O(V+E)]
Checking the water flow for the Hard Level	1
Total Punctuation	**3**

Alternative 3 Evaluation

ALTERNATIVE 1 (BFS & Floyd-Warshall's Algorithm) Verifying sewer system	
Criteria	**Evaluation Scale (Big-O)**
Checking the water flow for Easy Level	2 [O(V+E)]
Checking the water flow for the Hard Level	0
Total Punctuation	**2**

Comparison of Alternative 1 and Alternative 3 Algorithmical Time Complexity:

Criteria	**Verifying Sewer System**	
	Checking the water flow for Easy Level	*Checking the water flow for the Hard Level*
Alternative 1	**2**	**2**
Alternative 3	**1**	**0**
Total	**3**	**2**

5.3 Selection of the best Alternative:

Alternative 1:

This solution has been selected due to its comprehensive approach to the task at hand. Firstly, it is possible to observe that its quality fit properly not only the objectives of the game proposed, but also the requisites of the integrative task. On the other hand, the algorithms proposed by alternative one, and that will deal with the functional requirement of verifying the sewer system, are more effective than the second alternative and embrace more integrally the objectives of the levels of the game proposed.

PHASE 6: PREPARATION OF REPORTS AND SPECIFICATIONS

6.1 General Problem Specification:

Marlon Mania is a single player a game in which players will have to go deep into the sewage system to solve a massive problem caused by a huge earthquake. This earthquake has destroyed some crucial parts of the sewage system leaving most of the population without access to water. To solve this problem the player must be able to connect two sets of pipes that were disconnected due to the earthquake. The starting point will be known as the source and the arrival point will be known as the drainage.

The idea is that the player achieves this connection using the least amount of pipes and for the most serious damage, the player must place the pipes so that the water takes the shortest time to arrive. In order to do this, the player is able to place 3 different types of pipes: the first one is the horizontal pipe that allows water to flow form right to left or vice versa, the second one is the vertical pipe that allows water to flow up and down or vice versa, and finally the third one is the circular pipe, which allows to change the flow of water from up or down to left or right. Lastly, players must know that the game calculates the score based on how effective the player's solution is.

To model the situation at hand, a graph will be used as the main data structure. The user must have the possibility of switching between two different versions of graph (Adjacency List and Adjacency Matrix) and to check whether the user's solution is correctly built, two different types of graph's algorithms will be implemented (DFS and Dijkstra's Algorithm).

6.2 SubProblems Specification:

<table>
<tr><th colspan="2">Start a new Game</th></tr>
<tr><td>Sub-Problem Specification</td><td>The program has a to start a game, creating a new graph of 64 nodes with all connected edged but empty vertices.</td></tr>
<tr><td>Inputs</td><td>- Inp_1: Player's name
- Inp_2: Game level
- Inp_3: Type of graph</td></tr>
<tr><td>Outpust</td><td>- Out_1: 8x8 board with vertices of a graph, where two of them are of a specific type (source and drainage)</td></tr>
<tr><td colspan="2">Considerations:
- After the system receives the data entered by the user, it creates a new User with the name as the one entered, and created for them a new game board, which will be visualized as a matrix but is managed in a graph structure.
- The graph will be implemented according to the version chosen by the user. It can be:
- Adjacency matrix
- Adjacency list
- The levels chosen by the user can be:
- Hard
- Easy</td></tr>
</table>

Place pipe	
Sub-Problem Specification	The system must allow the user to place a pipe in one of the vertices of the boardgame (matrix)
Inputs	- **Inp_1:**Type of pipe - **Inp_2:** X coordinate of the pipe - **Inp_3:** Y coordinate of the pipe.
Outpust	- **Out_1:** 8x8 board with vertices of a graph, where two of them are of a specific type (source and drainage) and extra(s) vertex/vertices is "filled" with another type of pipe (the new vertex is located in coordinate (x,y)).

Considerations:
- The types of pipes that the user can connect are:
  - Horizontal (=)
  - Vertical (| |)
  - Circular (o)
- The pipes can be located within x values between 0 and 7, and y values between 0,7.
- The new pipe inserted in a vertex is registered in the matrix structure that is being used.	

Verify sewer system	
Sub-Problem Specification	The program must verify that, according to the level, that: - The pipes are correctly connected and create a path from the "source" to the "drainage" (Easy level). - The pipes are correctly connected, they create a path from the "source" to the "drainage" and they go through the "shortest" path possible.
Inputs	- **Inp_1:**N/A

Outpust	- **Out_1:** Confirmation Message of Success/Failure. Example:
	  - *"You won!"*
	  - *"You lost ☹. Keep trying"*

Considerations: - The system verifies, for both levels, that the pipes are correctly connected. The horizontal pipes can only relate to another horizontal pipe (on their right/left side); vertical pipes can	

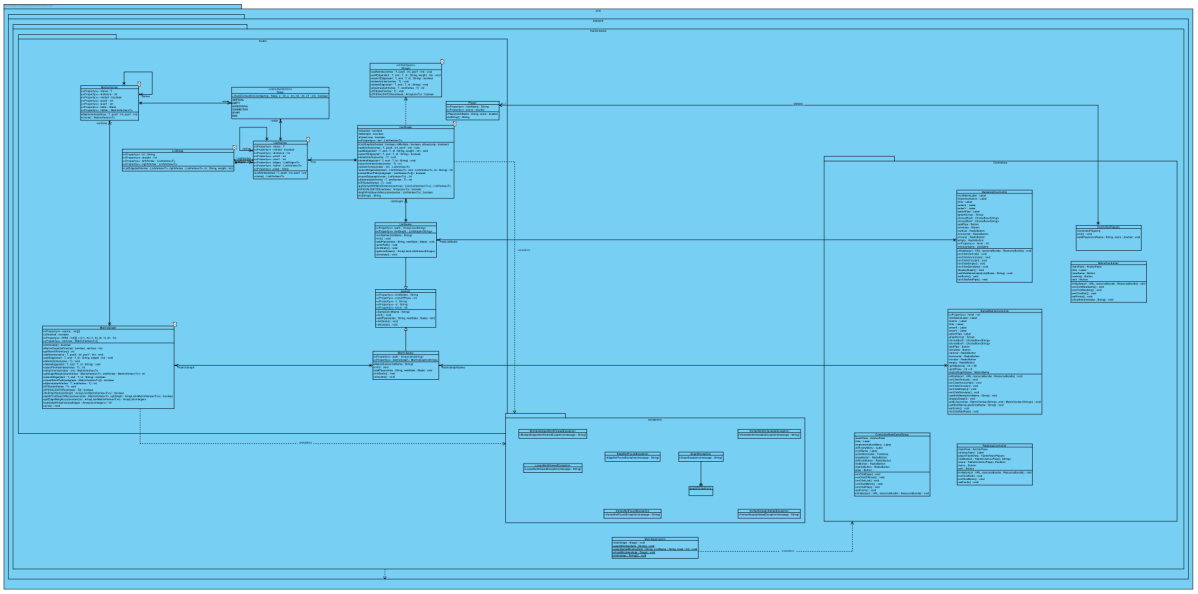
only be connected to another vertical pipe (up and down); and circular pipes connect a horizontal and vertical pipe forming a 90° angle with each other.

- For both levels, the system uses DFS algorithm on the matrix structure to verify whether the source and the drainage vertices are connected.
- For the hard level, the system applies Dijkstra´s algorithm on the matrix to verify that the path created by the user (if connected) is the one with the lowest weight inside the matrix.
- If the solution of the user accomplishes what the level requires, the system calculates its score, saves it in a scores list and show a success message for then to returns to the main page. Otherwise, the system show a failure message and returns to the game.

View Scores	
Sub-Problem Specification	The program must store in a list the scores of all the users that have successfully connected the pipes following their level requirements.
Inputs	- **Inp_1:**N/A
Outpust	- **Out_1:** Table including the players' names and their correspondent score.
Considerations: - The list is empty if no user has played or if no user has been able to successfully accomplish their respective level challenges.	

PHASE 7: Design Implementation

Class Diagram (*for better visualization, go to documentation*):



Graph Implementation (Matrix Graph)

(*See section below*)

Subroutine Specification

Name:	MatrixGraph()
Description	It instantiates a new matrix graph
Input	isDirected: boolean vertices: int
Return	none (Object MatrixGraph instantiated)

Construction

```
public MatrixGraph(boolean isDirected, int
vertices) {
    this.isDirected = isDirected;
    this.vertices = new
MatrixVertex[vertices];
    this.matrix = new
int[vertices][vertices];
}
```

Name:	getNumOfVertices()
Description	It returns the number of vertices in the graph
Input	none
Return	vertices.length (number of vertices in the graph)

```
public int getNumOfVertices(){
    return vertices.length;
}
```

Name:	addVertex()
Description	Creates a new vertex in a graph
Input	value: T
Return	None (new vertex added to the graph)

```
public void addVertex(T value) throws
VertexAlreadyAddedException {
    boolean stop = false;
    if (searchVertexIndex(value) != -1)
        throw new
VertexAlreadyAddedException("There is a vertex
with the same value");
    for (int i = 0; i < vertices.length &&
!stop; i++) {
        if (vertices[i] == null) {
            vertices[i] = new
MatrixVertex<>(value);
            vertices[i].setState(State.EMPTY);
            stop = true;
        }
    }
}
```

Name:	addEdge()
Description	Connects two vertices of a graph, creating an edge with weigh and identifier
Input	start: T end: T id: String weight: int
Return	None (new edge created in the graph)

```
public void addEdge(
        T start, T end, String id, int
weight
) throws VertexNotFoundException,
LoopsNotAllowedException,
MultipleEdgesNotAllowedException {
    int vertex1 = searchVertexIndex(start);
    int vertex2 = searchVertexIndex(end);

    if (vertex1 == vertex2) {
        System.out.println(start + " -> " +
end);
        throw new
LoopsNotAllowedException("Loops are not
allowed");

    }

    if (vertex1 != -1 && vertex2 != -1) {
        if (matrix[vertex1][vertex2] != 0)
            throw new
MultipleEdgesNotAllowedException("Multiples
edges are not allowed");
        if (isDirected) {
            matrix[vertex1][vertex2] =
weight;
        } else {
            matrix[vertex1][vertex2] =
weight;
            matrix[vertex2][vertex1] =
weight;
        }
    } else {
        String vError = vertex1 == -1 ?
"vertex1" : "vertex2";
        throw new
VertexNotFoundException("Cannot find the
vertex " + vError);
    }
}
```

Name:	deleteVertex()
Description	Deletes one of the vertices of the graph
Input	value: T
Return	None (vertex deleted from the graph)

```
public void deleteVertex(T value) throws
VertexNotFoundException {
    int oldVerPos =
searchVertexIndex(value);
    if (oldVerPos == -1) throw new
VertexNotFoundException("There's no such
vertex in the graph");
    for (int i = 0; i < vertices.length;
i++) {
        if
(vertices[i].getValue().equals(value)) {
            vertices[i] = null;
            break;
        }
    }

    for (int i = 0; i < matrix.length; i++)
{
        matrix[oldVerPos][i] = 0;
        matrix[i][oldVerPos] = 0;
    }
}
```

Name:	deleteEdge()
Description	Deletes one of the edges of the graph
Input	start: T end: T id: String
Return	None (edge identified deleted from the graph)

```
public void deleteEdge(T start, T end,
String id) throws EdgeNotFoundException,
VertexNotFoundException {
    int posVal1 = searchVertexIndex(start);
    int posVal2 = searchVertexIndex(end);

    boolean hasEdge = searchEdge(start,
end, id);

    if (hasEdge) {
        matrix[posVal1][posVal2] = 0;
        matrix[posVal2][posVal1] = 0;
    } else {
        throw new
EdgeNotFoundException("The edge was not
found");
    }
}
```

Name:	searchVertexIndex()
Description	Returns the index of and specific vertex in the graph
Input	value: T
Return	-1 (vertex not found) Index (index of the vertex in the matrix)

```
public int searchVertexIndex(T value) {
    for (int i = 0; i < matrix.length; i++)
{
        if (vertices[i] != null &&
vertices[i].getValue().equals(value)) {
            return i;
        }
    }

    return -1;
}
```

Name:	obtainVertex()
Description	Returns a specific vertex given an index
Input	index: int
Return	vertices[index] (vertex located in the index given)

```
public MatrixVertex<T> obtainVertex(int
index){
    return vertices[index];

}
```

Name:	getEdgeWeight()
Description	Returns the weight of an edge formed by given vertex
Input	startVertex: MatrixVertex<T> endVertex: MatrixVertex<T>
Return	0 (edgeNotFound) matrix[startIndex] [endIndex] (weight of edge connecting start and end vertex)

```
public int getEdgeWeight(MatrixVertex<T>
startVertex, MatrixVertex<T> endVertex) {
    int startIndex =
searchVertexIndex(startVertex.getValue());
    int endIndex =
searchVertexIndex(endVertex.getValue());

    if (startIndex != -1 && endIndex != -1)
{
        return
matrix[startIndex][endIndex];
    } else {
        // Manejar el caso donde uno de los
vértices no se encuentra
        return 0; // O podrías lanzar una
excepción, dependiendo de tus necesidades
    }
}
```

Name:	searchEdge()
Description	Returns whether there is an specific edge in the graph
Input	start: T end: T id: String
Return	result (boolean indicating wheter the edge with those vertices exists or not)

```
public boolean searchEdge(T start, T end,
String id) throws VertexNotFoundException {
    int vertex1 = searchVertexIndex(start);
    int vertex2 = searchVertexIndex(end);
    boolean result = false;

    String error = vertex1 == -1 ? "start"
: "end";
    if (vertex1 == -1 || vertex2 == -1)
throw new VertexNotFoundException("The " +
error + " vertex was not found");

    if (isDirected) {
        if (matrix[vertex1][vertex2] != 0)
{
            result = true;
        }
    } else {
        if (matrix[vertex1][vertex2] != 0
&& matrix[vertex2][vertex1] != 0) {
            result = true;
        }
    }

    return result;
}
```

Name:	dijkstra()
Description	Returns the distance of the path with the less weight in the graph
Input	startVertex: T endVertex: T
Return	Vertices[endVertexIndex]. getDistance() (length of the shortest path in a graph from one specific vertex to another)

```
public int dijkstra(T startVertex, T
endVertex) throws VertexNotFoundException,
VertexNotAchievableException {
    int startVertexIndex =
searchVertexIndex(startVertex);
    int endVertexIndex =
searchVertexIndex(endVertex);
    if (startVertexIndex == -1 ||
endVertexIndex == -1) {
        throw new
VertexNotFoundException("Vertex was not
found");
    }

    PriorityQueue<MatrixVertex<T>> q = new
PriorityQueue<>(Comparator.comparingInt(Mat
rixVertex::getDistance));

    for (MatrixVertex<T> vertex : vertices)
{
        if (vertex != null) {

vertex.setDistance(Integer.MAX_VALUE);
            vertex.setFather(null);
            vertex.setVisited(false);
        }
    }


vertices[startVertexIndex].setDistance(0);
    q.add(vertices[startVertexIndex]);

    while (!q.isEmpty()) {
        MatrixVertex<T> u = q.poll();
        if (u.isVisited()) {
            continue;
        }
        int uIndex =
searchVertexIndex(u.getValue());

        for (int i = 0; i <
vertices.length; i++) {
            if (matrix[uIndex][i] != 0 &&
!vertices[i].isVisited()) {
                int alt = u.getDistance() +
matrix[uIndex][i];
                if (alt <
vertices[i].getDistance()) {

vertices[i].setDistance(alt);

vertices[i].setFather(u);
                    q.add(vertices[i]);
                }
            }
        }
        u.setVisited(true);
    }
    if
(vertices[endVertexIndex].getDistance() ==
Integer.MAX_VALUE) {
        throw new
VertexNotAchievableException("The end
vertex is not reachable from the start
vertex.");
    }
    return
vertices[endVertexIndex].getDistance();
}
```

Name:	DFS()
Description	Checks wheter the nodes of the graph from an specific start point are connected
Input	startVertex: T
Return	none (traversal of the graph from the starting vertex given)

```
public void DFS(T startVertex) throws
VertexNotFoundException,
VertexNotAchievableException {
    Stack<MatrixVertex<T>> stack = new
Stack<>();
    int startIndex =
searchVertexIndex(startVertex);

    if (startIndex == -1) {
        throw new
VertexNotFoundException("Vertex was not
found");
    }

    stack.push(vertices[startIndex]);
    vertices[startIndex].setVisited(true);

    while (!stack.isEmpty()) {
        MatrixVertex<T> vertex =
stack.pop();
        for (int i = 0; i <
vertices.length; i++) {
            int currentIndex =
searchVertexIndex(vertex.getValue());

            if (!vertices[i].isVisited() &&
matrix[currentIndex][i] != 0) {
                stack.push(vertices[i]);

vertices[i].setVisited(true);
            }
        }
    }
}
```

Name:	DFSVALIDATOR()
Description	Checks wheter the nodes of the graph from an specific start point are connected and returns its findings
Input	vertexes: T[]
Return	dfsSimplified(sub_graph) (boolean indicating wheter the nodes in a graph are connected)

```
public boolean DFSVALIDATOR(T[] vertexes)
throws VertexNotFoundException,
VertexNotAchievableException {
    ArrayList<Integer> indexes = new
ArrayList<>();
    ArrayList<MatrixVertex<T>> temps = new
ArrayList<>();
    ArrayList<MatrixVertex<T>> sub_graph =
new ArrayList<>();
    for (T vertex : vertexes) {
    indexes.add(searchVertexIndex(vertex));
    }

    for (int i = 0; i < vertices.length-1;
i++) {
        if (indexes.get(i) == -1) {
            throw new
VertexNotFoundException("Start vertex not
found.");
        }

temps.add(vertices[indexes.get(i)]);
    }

    sub_graph.add(temps.get(0).clone());
    sub_graph.get(0).setVisited(false);

   for (int i = 1; i < temps.size(); i++) {
       sub_graph.add(temps.get(i).clone());
        sub_graph.get(i).setVisited(false);
    }
    return dfsSimplified(sub_graph);
}
```

Name:	dfsSimplified()
Description	Checks wheter the nodes of the subgraph with certain vertices are connected
Input	subGraph: ArrayList<MatrixVertex<T>>
Return	false (vertices in the subgraph are not connected) true (vertices in the subgraph are connected)

```
private boolean
dfsSimplified(ArrayList<MatrixVertex<T>>
subGraph) {
    for (MatrixVertex<T> vertex : subGraph)
{
        if (!vertex.isVisited()) {
            if
(!depthFirstSearchRecursive(vertex,
subGraph)) {
                return false;
            }
        }
    }
    return true;
}
```

Name:	depthFirstSearchRecursive()
Description	Checks whether a graph is connected given its list of vertices and a start
Input	vertex: MatrixVertex<T> subgraph: ArrayList<MatrixVertex<T>>
Return	false (graph not connected) true (graph connected)

```
private boolean
depthFirstSearchRecursive(MatrixVertex<T>
vertex, ArrayList<MatrixVertex<T>>
subGraph) {
    vertex.setVisited(true);

    for (MatrixVertex<T> neighbor :
subGraph) {
        if (!neighbor.isVisited() &&
vertex.getState().checkConnection(neighbor.
getState(),vertex.getPosX(),vertex.getPosY(
),neighbor.getPosX(),neighbor.getPosY())) {
            if
(!depthFirstSearchRecursive(neighbor,
subGraph)) {
                return false;
            }
        }
    }

    return true;
}
```

Name:	getEdgeWeighsList()
Description	List of the weights of all edges
Input	vertexList: ArrayList<MatrixVertex<T>>
Return	edgeWeights (list of all the weights of the edges of the list of vertices)

```
public ArrayList<Integer>
getEdgeWeightsList(ArrayList<MatrixVertex<T
>> vertexList) {
    ArrayList<Integer> edgeWeights = new
ArrayList<>();

    for (int i = 0; i < vertexList.size() -
1; i++) {
        MatrixVertex<T> currentVertex =
vertexList.get(i);
        MatrixVertex<T> nextVertex =
vertexList.get(i + 1);

        int currentVertexIndex =
searchVertexIndex(currentVertex.getValue())
;
        int nextVertexIndex =
searchVertexIndex(nextVertex.getValue());

        if (currentVertexIndex != -1 &&
nextVertexIndex != -1) {
            int weight =
matrix[currentVertexIndex][nextVertexIndex]
;
            edgeWeights.add(weight);
        } else {
            System.out.println("One of the
vertices is not found in the graph.");
        }
    }

    return edgeWeights;
}
```

Name:	subGraphDistance()
Description	Total weight of the edges of the path
Input	subEdges: ArrayList<MatrixVertex<T>>
Return	edgeWeights (list of all the weights of the edges of the list of vertices)

```
public int
subGraphDistance(ArrayList<Integer>
subEdges){
    return
subEdges.stream().mapToInt(Integer::valueOf
).sum();}
```

Name:	getVertices()
Description	Returns a matrix of vertices of a graph
Input	none
Return	Vertices (vertices of the graph)

```
public MatrixVertex<T>[] getVertices() {
    return vertices;
}
```

Name:	print()
Description	Show the graph in its matrix representation
Input	none
Return	None (shows the graph in a matrix representation)

```
public void print(){

    StringBuilder msg = new StringBuilder("
");
    for (MatrixVertex<T> vertex : vertices)
{

msg.append(vertex.getValue()).append("   ");
    }
    System.out.println(msg.toString());

    for (int i = 0; i < vertices.length;
i++) {
        System.out.printf("%-4s [ ",
vertices[i].getValue());

        for (int j = 0; j <
vertices.length; j++) {
            System.out.printf("%-4d ",
matrix[i][j]);
        }

        System.out.print("]");
        System.out.println();
    }

}
```

Name:	checkShortPath()
Description	Checks wheter an array of vertices corresponds to the shortest path of the graph
Input	Subgraph: MatrixVertex<T>[]
Return	res==acu (whether the subgraph is the shortest path in the graph)

```
public boolean checkShortPath(
        MatrixVertex<T>[] subgraph
) throws VertexNotAchievableException,
VertexNotFoundException {
    var res =
dijkstra(subgraph[0].getValue(),
subgraph[subgraph.length - 1].getValue());

    int first =
searchVertexIndex(subgraph[0].getValue());
    int second =
searchVertexIndex(subgraph[1].getValue());
    int acu = matrix[first][second] +
subgraph[0].getDistance();
    for (int i = 1; i < subgraph.length;
i++) {
        int last =
searchVertexIndex(subgraph[i -
1].getValue());
        int act =
searchVertexIndex(subgraph[i].getValue());
        acu += matrix[act][last] +
subgraph[i - 1].getDistance();
    }

    return res == acu;
}
```

Graph (Adjacency List) Subroutines

Subroutine Specification

Name:	ListGraph()
Description	It instantiates a new list graph
Input	isGuided: boolean isMultiple: boolean allowsLoop: boolean
Return	none (Object ListGraph instantiated)

Construction

```
public ListGraph(boolean isGuided, boolean
isMultiple, boolean allowsLoop) {
    list = new ArrayList<>();
    this.isGuided = isGuided;
    this.isMultiple = isMultiple;
    this.allowsLoop = allowsLoop;
}
```

Name:	addVertex()
Description	Creates a new vertex in a graph
Input	vertex: T
Return	None (new vertex added to the graph)

```
public void addVertex(T vertex) throws
VertexAlreadyAddedException {
    if (searchVertexIndex(vertex) == -1) {
        ListVertex<T> newVertex = new
ListVertex<>(vertex);
        newVertex.setState(State.EMPTY);
        list.add(newVertex);
    } else {
        System.err.println("ALGO RARO");
        /*throw new
VertexAlreadyAddedException("Vertex found:
" + vertex);*/
    }
}
```

Name:	addEdge()
Description	Connects two vertices of a graph, creating an edge with weigh and identifier
Input	start: T end: T id: String weight: int
Return	None (new edge created in the graph)

```
public void addEdge(T start, T end, String
id, int weight)
        throws VertexNotFoundException,
LoopsNotAllowedException,
MultipleEdgesNotAllowedException {
    int startVertex =
searchVertexIndex(start);
    int endVertex = searchVertexIndex(end);

    // Verifica si los vértices existen
    if (startVertex == -1) {
        throw new
VertexNotFoundException("Error. Start
vertex not found: " + start);
    }
    if (endVertex == -1) {
        throw new
VertexNotFoundException("Error. End vertex
not found: " + end);
    }
    // Verifica si hay bucles
    if (startVertex == endVertex &&
!allowsLoop) {
        throw new
LoopsNotAllowedException("Error. Loops not
allowed.");
    }
    // Verifica si ya existe la arista
    if
(searchEdgeIndex(list.get(startVertex),
list.get(endVertex), id) != -1 &&
!isMultiple) {
        throw new
MultipleEdgesNotAllowedException("Error.
Multiple edges between vertex not
allowed.");
    }
    // Añade la arista
    if (!isGuided) {
        // Si no es dirigido, añade la
arista en ambas direcciones

list.get(endVertex).getEdges().add(new
ListEdge<>(list.get(endVertex),
list.get(startVertex), id, weight));
    }

list.get(startVertex).getEdges().add(new
ListEdge<>(list.get(startVertex),
list.get(endVertex), id, weight));
}
```

Name:	searchEdge()
Description	Returns whether there is a specific edge in the graph
Input	start: T end: T id: String
Return	true/false (boolean indicating wheter the edge with those vertices exists or not)

```
public boolean searchEdge(T start, T end,
String id) throws VertexNotFoundException {
    if (searchVertexIndex(start) == -1 ||
searchVertexIndex(end) == -1) {
        throw new
VertexNotFoundException("Error. One vertex
not found.");
    }
    int startIndex =
searchVertexIndex(start);
    for (int i = 0; i <
list.get(startIndex).getEdges().size();
i++) {
        ListEdge<T> edge =
list.get(startIndex).getEdges().get(i);
        if
(edge.getRightVertex().getValue() == end &&
edge.getId().equals(id)) {
            return true;
        }
    }
    return false;
}
```

Name:	deleteVertex()
Description	Deletes one of the vertices of the graph
Input	value: T
Return	None (vertex deleted from the graph)

```
public void deleteVertex(T vertex) throws
VertexNotFoundException {
    int vertexIndex =
searchVertexIndex(vertex);
    if (vertexIndex == -1) {
        throw new
VertexNotFoundException("Error. Vertex not
found: " + vertex);
    }
    for (ListVertex<T> tVertex : list) {
        for (int j = 0; j <
tVertex.getEdges().size(); j++) {
            if
(tVertex.getEdges().get(j).getLeftVertex()
== list.get(vertexIndex) ||
tVertex.getEdges().get(j).getRightVertex()
== list.get(vertexIndex)) {

tVertex.getEdges().remove(j);
            }
        }
    }
    list.remove(vertexIndex);
}
```

Name:	deleteEdge()
Description	Deletes one of the edges of the graph
Input	start: T end: T id: String
Return	None (edge identified deleted from the graph)

```
public void deleteEdge(T start, T end,
String id) throws EdgeNotFoundException,
VertexNotFoundException {
    int startIndex =
searchVertexIndex(start);
    int endIndex = searchVertexIndex(end);
    if (startIndex == -1 || endIndex == -1)
{
        throw new
VertexNotFoundException("Error. One vertex
not found.");
    }
    if
(searchEdgeIndex(list.get(startIndex),
list.get(endIndex), id) == -1) {
        throw new
EdgeNotFoundException("Error. Edge not
found: " + start + " -> " + end + " (" + id
+ ")");
    }
    if (!isGuided) {

list.get(endIndex).getEdges().remove(search
EdgeIndex(list.get(endIndex),
list.get(startIndex), id));
    }

list.get(startIndex).getEdges().remove(sear
chEdgeIndex(list.get(startIndex),
list.get(endIndex), id));
}
```

Name:	searchVertexIndex()
Description	Returns the index of and specific vertex in the graph
Input	value: T
Return	-1 (vertex not found) Index (index of the vertex in the matrix)

```
public int searchVertexIndex(T vertex) {
    for (int i = 0; i < list.size(); i++) {
        if
(list.get(i).getValue().equals(vertex)) {
            return i;
        }
    }
    return -1;
}
```

Name:	obtainVertex ()
Description	Returns the vertex located in a certain index
Input	index: int
Return	list.get(index) (Vertex located in the ´index´ position)

```
public ListVertex<T> obtainVertex(int
index) {
    return list.get(index);
}
```

Name:	seachEdgeIndex()
Description	Returns the index of a certain edge in the graph
Input	start: ListVertex<T> end: ListVertex<T> id: String
Return	-1 (edge not found) Index (index of the edge being looked for)

```
private int searchEdgeIndex(ListVertex<T>
start, ListVertex<T> end, String id) {
    for (int i = 0; i <
start.getEdges().size(); i++) {
        if
(start.getEdges().get(i).getLeftVertex() ==
start &&
start.getEdges().get(i).getRightVertex() ==
end &&
start.getEdges().get(i).getId().equals(id))
{
            return i;
        }
    }
    return -1;
}
```

Name:	checkShortPath()
Description	Checks whether a list of vertices corresponds to the shortest path of the graph
Input	subgraph: ListVertex<T>[]
Return	res==subgraphValue (whether the subgraph is the shortest path in the graph)

```
public boolean checkShortPath(
        ListVertex<T>[] subgraph
) throws VertexNotAchievableException,
VertexNotFoundException {
    var res =
dijkstra(subgraph[0].getValue(),
subgraph[subgraph.length - 1].getValue());
    int subgraphValue =
travelSubgraph(subgraph[0]);

    return res == subgraphValue;
}
```

Name:	travelSubgraph()
Description	Distance of the path beginning in a certain vertex
Input	node: ListVertex<T>[]
Return	node.getDistance() (distance of the path of the graph from said vertex)

```
public int travelSubgraph(ListVertex<T>
node) {
    if (node == null) return 0;

    for (ListEdge<T> edge :
node.getEdges()) {
        return node.getDistance() +
edge.getWeight() +
travelSubgraph(edge.getRightVertex());
    }

    return node.getDistance();
}
```

Name:	dijkstra()
Description	Returns the distance of the path with the less weight in the graph
Input	startVertex: T endVertex: T
Return	Vertices[endVertexIndex].getDistance() (length of the shortest path in a graph from one specific vertex to another)

```
public int dijkstra(T startVertex, T
endVertex) throws VertexNotFoundException,
VertexNotAchievableException {
    int startVertexIndex =
searchVertexIndex(startVertex);
    int endVertexIndex =
searchVertexIndex(endVertex);

    if (startVertexIndex == -1 ||
endVertexIndex == -1) {
        throw new
VertexNotFoundException("Start or end
vertex not found.");
    }

    PriorityQueue<ListVertex<T>> q = new
PriorityQueue<>(Comparator.comparingInt(Lis
tVertex::getDistance));

    for (ListVertex<T> vertex : list) {

vertex.setDistance(Integer.MAX_VALUE);
        vertex.setFather(null);
        q.add(vertex);
    }

list.get(startVertexIndex).setDistance(0);

    while (!q.isEmpty()) {
        ListVertex<T> u = q.poll();
        if (u == null) {
            throw new
VertexNotAchievableException("No path found
between the specified vertices.");
        }
        for (ListEdge<T> edge :
u.getEdges()) {
            ListVertex<T> v =
edge.getRightVertex();
            int alt = u.getDistance() +
edge.getWeight();
            if (alt < v.getDistance()) {
                v.setDistance(alt);
                v.setFather(u);
                q.remove(v);
                q.add(v);
            }
        }
    }
    ListVertex<T> currentVertex =
list.get(endVertexIndex);
    if (currentVertex.getDistance() ==
Integer.MAX_VALUE) {
        throw new
VertexNotAchievableException("End Vertex
Not Achievable");
    }


    return currentVertex.getDistance();
}
```

Name:	DFS()
Description	Checks wheter the nodes of the graph from a specific start point are connected
Input	startVertex: T
Return	none (traversal of the graph from the starting vertex given)

```
public void DFS(T startVertex) throws
VertexNotFoundException,
VertexNotAchievableException {
    int startIndex =
searchVertexIndex(startVertex);

    if (startIndex == -1) {
        throw new
VertexNotFoundException("Start vertex not
found.");
    }

    Stack<ListVertex<T>> stack = new
Stack<>();
    stack.push(list.get(startIndex));

    while (!stack.isEmpty()) {
        ListVertex<T> currentVertex =
stack.pop();

        if (!currentVertex.isVisited()) {
            currentVertex.setVisited(true);

            for (ListEdge<T> edge :
currentVertex.getEdges()) {
                ListVertex<T> neighbor =
edge.getRightVertex();
                if (!neighbor.isVisited())
{
                    stack.push(neighbor);
                }
            }
        }
    }
}
```

Name:	getVertexWithMinDistance()
Description	Returns the vertex with the minimun distance in a list of vertices
Input	vertices: List<ListVertex<T>>
Return	minVertex (vertex with the minimum distance)

```
private ListVertex<T>
getVertexWithMinDistance(List<ListVertex<T>
> vertices) {
    ListVertex<T> minVertex = null;
    int minDistance = Integer.MAX_VALUE;
    for (ListVertex<T> vertex : vertices) {
        if (vertex.getDistance() <
minDistance) {
            minVertex = vertex;
            minDistance =
vertex.getDistance();
        }
    }
    return minVertex;
}
```

Name:	DFSVALIDATOR()
Description	Checks wheter the nodes of the graph from an specific start point are connected and returns its findings
Input	vertexes: T[]
Return	dfsFirstSearchRecursive(sub_graph) (boolean indicating whether the nodes in a graph are connected)

```java
public boolean DFSVALIDATOR(T[] vertexes)
throws VertexNotFoundException,
VertexNotAchievableException {
    ArrayList<Integer> indexes = new
ArrayList<>();
    ArrayList<ListVertex<T>> temps = new
ArrayList<>();
    ArrayList<ListVertex<T>> sub_graph =
new ArrayList<>();
    for (T vertex : vertexes) {

indexes.add(searchVertexIndex(vertex));
    }

    for (Integer index : indexes) {
        if (index == -1) {
            throw new
VertexNotFoundException("Start vertex not
found.");
        }

        temps.add(list.get(index));
    }

    sub_graph.add(temps.get(0).clone());
    sub_graph.get(0).setVisited(false);

    for (int i = 1; i < temps.size(); i++)
{

sub_graph.add(temps.get(i).clone());
        sub_graph.get(i).setVisited(false);
        sub_graph.get(i -
1).getEdges().add(new
ListEdge<>(sub_graph.get(i - 1),
sub_graph.get(i), i + "", 0));
    }

    return
depthFirstSearchRecursive(sub_graph.get(0))
;
}
```

Name:	depthFirstSearchRecursive()
Description	Checks whether a graph is connected given its list of vertices
Input	vertex: MatrixVertex<T> subgraph: ArrayList<MatrixVertex<T>>
Return	false (graph not connected) true (graph connected)

```java
private boolean
depthFirstSearchRecursive(ListVertex<T>
vertex) {
    vertex.setVisited(true); // Marca el
vértice actual como visitado
    // Recorre los vértices adyacentes no
visitados
    for (ListEdge<T> edge :
vertex.getEdges()) {
        ListVertex<T> neighbor =
edge.getRightVertex();
        if (!neighbor.isVisited()) {
            if
(vertex.getState().checkConnection(neighbor
.getState(), vertex.getPosX(),
vertex.getPosY(), neighbor.getPosX(),
neighbor.getPosY())) {

depthFirstSearchRecursive(neighbor);
            } else {
                return false;
            }
        }
    }
    return true;
}
```

Name:	toString()
Description	Show the graph in its list representation
Input	none
Return	ans.toString() (the graph in a list representation)

```
public String toString() {
    StringBuilder ans = new
StringBuilder();
    for (ListVertex<T> u : list) {
        int limit = 6;
        String valueStr = String.format("%"
+ limit + "s", u.getValue());
        ans.append(String.format("%s -> {
", valueStr));
        for (ListVertex<T> v :
u.getEdges().stream().map(ListEdge::getRigh
tVertex).toList()) {
            ans.append(String.format("%s,
", v.getValue()));
        }
        if (!u.getEdges().isEmpty())
ans.replace(ans.length() - 2, ans.length(),
"");
        ans.append(" }\n");
    }
    return ans.toString();

}
```

Name:	getList()
Description	Returns the list of vertices of the graph with their connections
Input	none
Return	list (list of vertices of the graph)

```
public ArrayList<ListVertex<T>> getList() {
    return list;
}
```