**THE DESIGN METHOD IN ENGINEERING**

**PHASE 1. IDENTIFYING THE PROBLEM**

**Problem identification:**

A group of programmers want to create a video game that generates a maze, by default. In this game, the player must try to find the exit while is being chased by an enemy who will always know the shortest path to reach the player character. If the player finds the level too challenging, they will have the option to see the path from where they are to the exit but use this power will make the enemy increase their speed.

|  |  |
| --- | --- |
| **Customer** | Group of programmers |
| **User** | Clients of the group |
| **Functional requirements** | * FR 1. Create a maze. * R.F 2. Generate player. * R.F 3. Generate enemy. * R.F 4. Show the way to the exit. * R.F 5. Allow the player to move using the keyboard. * R.F 6. Create the graphical user interface. |
| **Context of the problem** | A group wants to create a video game that allows the users to try to escape from a maze while they are being chased |
| **Non-functional requirements** | NFR1: Scalable Software |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name or identifier** | FR 1. Create a maze | | |
| **Summary** | The system must allow the creation of the object/screen maze. | | |
| **Inputs** | **input name** | **Data type** | **Selection or repetition condition** |
| isDirected | boolean | Determines if the paths of the created maze could be traveled in a single direction or not |
| start | Object | It is the point at which the player appears |
| exit | Object | It is the point that the player must reach without being eliminated by the enemy |
| **General activities needed to obtain the results** | * Receive the attributes of the maze. | | |
| **Result or postcondition** | The maze was successfully created | | |
| **Outputs** | **Output name** | **Data type** | **Selection or repetition condition** |
| errorMssg | String | Show was an error during the creation of the maze. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name or identifier** | FR 2. Generate a player | | | |
| **Summary** | The system must allow the automatic generation of the player once the user clicked the “play” button | | | |
| **Inputs** | **input name** | **Data type** | | **Selection or repetition condition** |
| name | String | | Since there is only one player, this is added with their name |
| **General activities needed to obtain the results** | * Verify that the maze has been previously created. * Create the player and place them at the starting point of the maze. | | | |
| **Result or postcondition** | The player was successfully generating | | | |
| **Outputs** | **Output name** | | **Data type** | **Selection or repetition condition** |
| errorMssg | String | | Show it was an error during the creation of the player. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name or identifier** | FR 3. Generate an enemy | | | |
| **Summary** | The system must allow the automatic generation of the enemy once the user clicked the “play” button | | | |
| **Inputs** | **input name** | **Data type** | | **Selection or repetition condition** |
| position | String | | Since there is only one player, this is added with their name |
| velocity | | double | This attribute determines how fast the enemy will move. |
| **General activities needed to obtain the results** | * Verify that the maze has been previously created. * Create the enemy and place them at some point of the maze. | | | |
| **Result or postcondition** | The enemy was successfully generating | | | |
| **Outputs** | **Output name** | | **Data type** | **Selection or repetition condition** |
| errorMssg | String | | Show it was an error during the creation of the enemy. |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Name or identifier** | FR 4. Show the way to the exit | | | | | |
| **Summary** | The system must allow the view of the path that leads from the player's position to the exit of the maze | | | | | |
| **Inputs** | **input name** | **Data type** | | | **Selection or repetition condition** | |
| positionPlayer | Object | | | It’s needed to know the player's position to show the path to the exit. | |
| exit | Object | | | It’s needed to know the exit point of the maze to be able to show a path from the player to this point | |
| **General activities needed to obtain the results** | * Verify that there is a path from where the player is to the exit of the maze. * Increase the speed of movement of the enemy | | | | | |
| **Result or postcondition** | The path to the exit is show | | | | | |
| **Outputs** | **Output name** | | | **Data type** | | **Selection or repetition condition** |
| msg | | String | | | Show the steps from the player to the exit |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name or identifier** | FR 5. S Allow the player to move using the keyboard | | | |
| **Summary** | This feature enables the player to control their in-game character's movement using the keyboard. | | | |
| **Inputs** | **input name** | **Data type** | | **Selection or repetition condition** |
| keyPress | String | | Whenever the player presses a key |
| **General activities needed to obtain the results** | * Continuously monitor keyboard input from the user. * Validate which key was selected by the user to know in which direction their character will move | | | |
| **Result or postcondition** | The player moves in the corresponding direction | | | |
| **Outputs** | **Output name** | | **Data type** | **Selection or repetition condition** |
|  |  | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name or identifier** | FR 6. Undo changes | | | |
| **Summary** | The system must undo the last changes made | | | |
| **Inputs** | **input name** | **Data type** | | **Selection or repetition condition** |
| action | String | | It’s the last action made for the user |
| **General activities needed to obtain the results** | * Go to the list of activities made for the user. * Remove/deleted the last activity. | | | |
| **Result or postcondition** | The last action made for the user is deleted. | | | |
| **Outputs** | **Output name** | | **Data type** | **Selection or repetition condition** |
| errorMessage | String | | Show it was an error during the undo of the action. |

**PHASE 2. GATHERING OF THE NECESSARY INFORMATION**

**Import clarification to kept in mind:**

Since the client wants to create a random maze, it will have different paths through which the player can go from one position to another, in addition to obstacles such as walls or other types. The best way to adapt this is through the Graph structure because allows to use the Prim or Kruskal algorithms with some modifications for the random generation of the maze, which makes the implementation easier compared to other structures such as lists or trees in which the entire algorithm would have to be created instead of modifying one that already exists and is efficient.

It should also be noted that in this structure the search is very efficient, for example, to find the shortest path between two nodes in a weighted graph, which would be equivalent to the enemy chasing the player from the shortest path. Also, each node of a graph can contain information, in this case the nodes can contains information about if the position is an obstacle or not, or the path from one node to another that the player can travel.

Finally, in a graph, the connections between nodes, cells of the maze in this case, can be explicitly represented by edges. This is especially useful for modeling the topology of a maze, where cells are connected by corridors or paths. This is because graphs are very adaptable and can easily represent the non-linear structure of a maze, where each cell can have multiple connections with other cells.

**Needed information:**

***Graph***

A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices (V) and a set of edges (E). The graph is denoted by G (V, E). In this way, the components of a Graph are:

* Vertices: Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabeled.
* Edges: Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labelled/unlabeled.

Diagrama

Descripción generada automáticamente

Also, the most common types of graphs are:

* **Undirected graph:** is a graph where the edges do not have a specific direction and it is bidirectional in nature it does not have a parent-child relation concept as there is no direction.
* **Directed graph:** Is a graph that is unidirectional in this the edges have a specific direction, and the edges have directions specified with them also a directed graph can contain cycles.

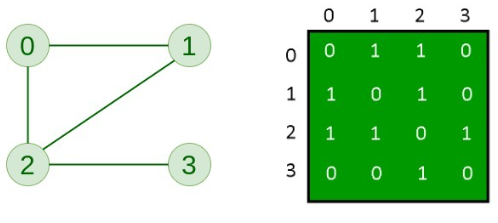
Forma

Descripción generada automáticamente

There are two ways to store a graph:

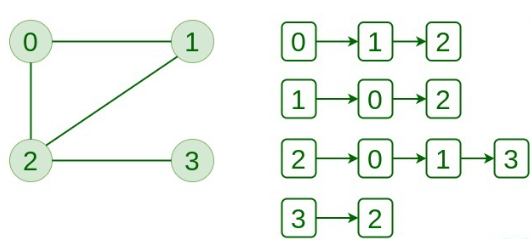
* Adjacency Matrix

In this method, the graph is stored in the form of the 2D matrix where rows and columns denote vertices. Each entry in the matrix represents the weight of the edge between those vertices.



* Adjacency List

This graph is represented as a collection of linked lists. There is an array of pointer which points to the edges connected to that vertex.



Basic Operations on Graphs:

* Insertion of Nodes/Edges in the graph.
* Deletion of Nodes/Edges in the graph.
* Searching on Graphs – Search an entity in the graph.
* Traversal of Graphs – Traversing all the nodes in the graph.

Search algorithms:

* **Breadth First Search (BFS)** algorithm is used to search a graph data structure for a node that meets a set of criteria. It starts at the root of the graph and visits all nodes at the current depth level before moving on to the nodes at the next depth level.
* **Depth-first search** is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

Shortest paths algorithms: The shortest path in a graph is defined as the minimum cost route from one vertex to another. This is most seen in weighted directed graphs but are also applicable to undirected graphs. The two common shortest path algorithms are:

* Dijkstra’s
* Floyd-Warshall

And **Minimum Spanning Tree (MST)** is a subset of edges of a connected weighted undirected graph that connects all the vertices together with the minimum possible total edge weight. To derive an MST, Prim’s algorithm or Kruskal’s algorithm can be used.

(Geeks for Geeks, 2023)

***Tree***

A tree data structure is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search. It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.

The topmost node of the tree is called the root, and the nodes below it is called the child nodes. Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure.

Diagrama

Descripción generada automáticamente

Types of Tree data structures:

* Binary tree: In a binary tree, each node can have a maximum of two children linked to it. Some common types of binary trees include full binary trees, complete binary trees, balanced binary trees, and degenerate or pathological binary trees.
* Ternary Tree: A Ternary Tree is a tree data structure in which each node has at most three child nodes, usually distinguished as “left”, “mid” and “right”.
* N-ary Tree or Generic Tree: Generic trees are a collection of nodes where each node is a data structure that consists of records and a list of references to its children (duplicate references are not allowed). Unlike the linked list, each node stores the address of multiple nodes.

Basic Operation of Tree Data Structure:

* Create – create a tree in the data structure.
* Insert − Inserts data in a tree.
* Search − Searches specific data in a tree to check whether it is present or not.
* Traversal:
  + Preorder Traversal – perform Traveling a tree in a pre-order manner in the data structure.
  + In order Traversal – perform Traveling a tree in an in-order manner.
  + Post-order Traversal –perform Traveling a tree in a post-order manner.

Properties of Tree Data Structure:

* Number of edges: An edge can be defined as the connection between two nodes. If a tree has N nodes, then it will have (N-1) edges. There is only one path from each node to any other node of the tree.
* Depth of a node: The depth of a node is defined as the length of the path from the root to that node. Each edge adds 1 unit of length to the path. So, it can also be defined as the number of edges in the path from the root of the tree to the node.
* Height of a node: The height of a node can be defined as the length of the longest path from the node to a leaf node of the tree.
* Height of the Tree: The height of a tree is the length of the longest path from the root of the tree to a leaf node of the tree.

(Geeks for Geeks, 2023)

**PHASE 3: FINDING CREATIVE SOLUTIONS**

What structures can we use to solve the problem?

The structures that can be used to implement a maze video game would be non-linear structures, so the options are Graphs or Trees, because more than one relationship between nodes will be needed to simulate the paths and the general map of the video game. But from the beginning there was an idea that the data structure to be implemented in the solution was: graph. In phase two, it was defined, making easier to understand its application in relation to the problem. Even so, in the following phases there will be a detailed analysis of why this specific structure were chosen over trees.

**PHASE 4. TRANSITION FROM IDEAS TO PRELIMINARY DESIGNS**

The graph will allow the generation of the maze through the Kruskal/prim algorithms, the search used so that the player can see the exit of the maze through the DFS algorithm and the Dijkstra algorithm so that the enemy chases the player from the shortest path.

**Creation of the maze:**

**Graph to create the maze:**

Efficiency: Graphs provide efficient representations for modeling mazes, especially in scenarios where complex connectivity and diverse pathways are desired. The ability to have multiple connections between nodes allows for the creation of intricate maze designs.

Contextual Justification: In the context of the creation of a maze in a video game, it is crucial to have a flexible and dynamic structure that can offer players a challenging and engaging experience. Graphs, with their ability to represent diverse relationships between nodes, align well with the need for complex maze configurations.

**Pros:**

* More variety in connections, enabling the creation of mazes with multiple paths, loops, and interconnections.
* Search algorithms are natural, such as depth-first search or breadth-first search for navigating through the interconnected nodes of a graph.

**Cons:**

* Designing and working with graphs may require a good understanding of graph theory and algorithms.
* The flexibility of graphs can lead to highly intricate mazes, which might be challenging to manage or understand.

***Inefficiency of alternative approaches:***

**Tree to create the maze:**

Efficiency: While trees are efficient data structures, they may be less suitable for modeling mazes due to their hierarchical and acyclic nature. Trees inherently lack the interconnectedness needed to represent the complex pathways found in mazes.

Contextual Justification: In the context of maze creation for a video game, using a tree structure may limit the potential for intricate and non-linear maze designs. Trees are more appropriate for hierarchical structures rather than the interwoven connections required for a challenging maze.

**Pros:**

* Trees naturally exhibit a hierarchical structure, that is less aligned with the non-linear and interconnected characteristics of a maze.
* A tree might offer a straightforward representation.

**Cons:**

* Trees have limited connectivity compared to graphs, which restricts the complexity and variety of connections that can be represented.
* Maze-solving algorithms may be less naturally applied to trees, as these algorithms often rely on traversing interconnected nodes.

**Enemy pursuit of the player through the shortest path**

**Graph for the enemy pursuing the player using the shortest path:**

Efficiency: Graphs provide an efficient representation for modeling the enemy's pursuit of the player using the shortest path. The interconnected nature of graphs allows for quick and effective calculations of the shortest path between the enemy and the player.

Contextual Justification: Using a graph to model the maze and employing algorithms for finding the shortest path enables the enemy to navigate the maze efficiently, enhancing the overall gaming experience by creating a challenging and dynamic pursuit scenario.

**Pros:**

* Graphs allow for a greater variety of connections between nodes, enabling the enemy to choose from multiple paths in the pursuit of the player.
* Graphs provide a natural representation for search algorithms like Dijkstra's when determining the shortest path.

**Cons:**

* Individuals responsible for creating the enemy behavior must possess a solid knowledge of the maze structure.

***Inefficiency of alternative approaches:***

**Tree for the enemy pursuing the player using the shortest path:**

Efficiency: Decision trees, while not inherently the most efficient for pathfinding, can offer simplicity in implementation. However, their efficiency may be compromised in scenarios with intricate maps or dynamic environments.

Contextual Justification: Decision trees can be justified in scenarios where the game environment is relatively simple, and the pursuit logic doesn't require extensive adaptability. For instance, in a game with a straightforward layout and predictable player movements.

**Pros:**

* Simplicity: Decision trees provide a straightforward and easy-to-understand logic flow, making them suitable for basic pursuit scenarios.
* Ease of Implementation: Implementing a decision tree is generally less complex than graph-based approaches, making it suitable for scenarios with limited environmental complexity.

**Cons:**

* Limited Adaptability: Decision trees lack the dynamic adaptability of graphs. They might struggle to handle unexpected changes in the game environment or player behavior, leading to less realistic pursuit behavior.
* Inefficiency in Complex Environments: In complex game maps with multiple paths and obstacles, decision trees may lead to less efficient pursuit strategies, potentially resulting in less challenging gameplay.

**Enemy pursuit of the player through the shortest path**

**Graph for the know route of scape:**

Efficiency: The efficiency of providing a user with the escape route in a maze using a graph structure is generally high. Graphs are well-suited for representing maze structures, and algorithms like Depth-First Search (DFS) or Breadth-First Search (BFS) can efficiently find the exit path.

Contextual Justification: Using a graph structure for maze representation enables a systematic approach to solving the problem of guiding a user out of the maze. Each intersection or decision point in the maze can be represented as a node, and the paths between these nodes as edges. This allows for the application of graph traversal algorithms to find the optimal or any valid escape route.

**Pros:**

* Graph traversal algorithms like DFS or BFS are well-established and efficient for finding paths in a maze represented as a graph. They have a time complexity of O(V + E), where V is the number of vertices (nodes) and E is the number of edges (paths) in the graph.
* Graph structures provide flexibility in representing complex maze configurations, including loops, multiple paths, and dead ends. This makes them suitable for a variety of maze designs.
* Depending on the specific algorithm used, the system can provide an optimal path to the exit, ensuring that the user takes the shortest route.

**Cons:**

* While graph traversal algorithms are generally efficient, large mazes with a significant number of nodes and edges may consume more computational resources. This can be a consideration for real-time applications or systems with limited computing power.

***Inefficiency of alternative approaches:***

**Graph for the know route of scape:**

Efficiency: The efficiency of providing a user with the escape route in a maze using a tree structure may vary depending on the specific algorithm and the nature of the maze. Unlike graphs, trees may not inherently represent all possible paths in a maze, and finding the exit path might require a different approach.

Contextual Justification: Using a tree structure for maze representation poses challenges compared to a graph, as trees typically have a hierarchical and branching structure that might not directly capture the interconnected nature of maze paths. While tree structures are useful for representing certain types of relationships, they may not be the most intuitive choice for modeling mazes.

**Pros:**

* Tree structures can represent hierarchical relationships, which might be useful in certain maze scenarios where the layout has a clear hierarchical structure. This can be advantageous for mazes with distinct levels or layers.
* In cases where the maze design aligns with a tree structure, representing it as such may simplify the modeling process. This can lead to a more straightforward representation, particularly for mazes with a limited number of branching points.

**Cons:**

* Limited Connectivity: Trees inherently have a single root and branching structure, which may not capture the interconnectedness of maze paths. This limitation could make it challenging to represent certain maze configurations accurately.
* Unlike graph traversal algorithms that are well-suited for finding paths in interconnected structures, finding the exit path in a maze represented as a tree may require a specialized algorithm. This could potentially lead to increased computational complexity.
* Ensuring an optimal path to the exit in a tree structure might be more challenging, as the hierarchical nature may not naturally lend itself to the concept of optimal paths.

**PHASE 5. EVALUATION AND SELECTION OF THE BEST SOLUTION:**

**Creation of the maze:**

Among the possible solutions for generating the maze are the bfs and dfs algorithms

**Criterion A. Guarantees a solution/exit from the maze:**

* [2] Yes
* [1] No

**Criterion B. Produces highly branched connections:**

* [2] No
* [1] Yes

**Criterion C. Type of path created:**

* [2] Long
* [1] Short

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Criterion A | Criterion B | Criterion C | Total |
| bfs | 2 | 2 | 1 | 5 |
| dfs | 1 | 1 | 2 | 4 |

**Enemy pursuit of the player through the shortest path:**

Among the possible solutions for the player’s chase are the Dijkstra or Floyd-Warshall algorithms

**Criterion A. Route precision:**

* [2] Yes
* [1] No

**Criterion B. Highly implementation complexity:**

* [2] No
* [1] Yes

**Criterion C. Computational complexity:**

* [2] O((V+E) \* log(V))
* [1] O(Vˆ3)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Criterion A | Criterion B | Criterion C | Total |
| Dijkstra | 2 | 2 | 2 | 6 |
| Floyd-Warshall | 2 | 1 | 1 | 4 |

**PHASE 6. REPORTS**

|  |  |  |
| --- | --- | --- |
| **Name** | **Class** | **Scenery** |
| setUpStage1 | GraphTest | Initializes one graph structure. |
| setUpStage2 | GraphTest | Initializes one graph structure with 3 vertices. |
| setUpStage3 | GraphTest | Initializes one graph structure with 3 vertices and 2 edges. |
| setUpStage4 | GraphTest | Initializes one graph structure that is a disconnected. |
| setUpStage5 | GraphTest | Initializes one graph structure with 1 vertex. |
| setUpStage6 | GraphTest | Initializes one graph structure that is a complete bipartite. |
| setUpStage7 | GraphTest | Initializes one graph structure that has a cycle. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: addVertexTestStandard()** | | | | |
| **Objective of test: Add a vertex to an empty graph.** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | addVertext() | setUpStage1 | String key | None |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: addVertexTestLimit()** | | | | |
| **Objective of test: Add 1000 vertices to a vertex to an empty graph.** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | addVertext() | setUpStage1 | String key | None |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: addVertexTestInteresting()** | | | | |
| **Objective of test: Add a duplicated vertex to a graph with vertices.** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | addVertext() | setUpStage2 | String key | None |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: removeVertexTestStandard()** | | | | |
| **Objective of test: Remove from a graph with vertices.** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | removeVertext() | setUpStage2 | String key | None |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: removeVertexTestLimit()** | | | | |
| **Objective of test: Remove from an empty graph.** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | removeVertext() | setUpStage1 | String key | None |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: removeVertexTestInteresting()** | | | | |
| **Objective of test: Remove a vertex with edges.** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | removeVertext() | setUpStage3 | String key | None |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: addEdgeTestStandard()** | | | | |
| **Objective of test: Add an edge to a graph with vertices.** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | addEdge() | setUpStage2 | String key,  String key | None |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: addEdgeTestLimit()** | | | | |
| **Objective of test: Add an edge to an empty graph.** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | addEdge() | setUpStage1 | String key,  String key | None |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: addEdgeTestInteresting()** | | | | |
| **Objective of test: Remove edge from a graph with edges.** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | addEdge() | setUpStage3 | String key,  String key | None |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: removeEdgeTestStandard()** | | | | |
| **Objective of test: Remove edge from a graph with edges** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | removeEdge() | setUpStage3 | String key,  String key | None |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: removeEdgeTestLimit()** | | | | |
| **Objective of test: Remove edge from an empty graph** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | removeEdge() | setUpStage1 | String key,  String key | None |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: removeEdgeTestInteresting()** | | | | |
| **Objective of test: Remove an edge from a graph with vertices** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | removeEdge() | setUpStage1 | String key,  String key | None |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: getNeighborsTestStandard()** | | | | |
| **Objective of test: Search neighbors of a vertex with edges** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | getNeighbors() | setUpStage6 | String key | List<String> with elements |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: getNeighborsTestLimit()** | | | | |
| **Objective of test: Search neighbors when the graph just has one vertex** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | getNeighbors() | setUpStage5 | String key | List<String> without elements |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: getNeighborsTestInteresting()** | | | | |
| **Objective of test: Search neighbors of a vertex without edges** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | getNeighbors() | setUpStage3 | String key | List<String> without elements |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: bfsTestStandard()** | | | | |
| **Objective of test: Traverse a graph** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | bfs() | setUpStage4 | String key | List<String> with elements |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: bfsTestLimit()** | | | | |
| **Objective of test: BFS from a vertex that does not exist** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | bfs() | setUpStage1 | String key | List<String> null |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: bfsTestInteresting()** | | | | |
| **Objective of test: BFS from a graph that is disconnected** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | bfs() | setUpStage4 | String key | List<String> with elements |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: dfsTestStandard()** | | | | |
| **Objective of test: Traverse a graph** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | dfs() | setUpStage4 | String key | List<String> with elements |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: dfsTestLimit()** | | | | |
| **Objective of test: DFS from a vertex that does not exist** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | dfs() | setUpStage1 | String key | List<String> without elements |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: dfsTestInteresting()** | | | | |
| **Objective of test: DFS from a graph that is disconnected** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | dfs() | setUpStage4 | String key | List<String> with elements |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: dijkstraTestStandard()** | | | | |
| **Objective of test: Dijkstra from a graph with vertices and is disconnected** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | dijkstra() | setUpStage3 | String key | Map<String, Pair<Integer, String>> with elements |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: dijkstraTestLimit()** | | | | |
| **Objective of test: Dijkstra from graph with one node** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | dijkstra() | setUpStage5 | String key | Map<String, Pair<Integer, String>> with elements |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: dijkstraTestInteresting()** | | | | |
| **Objective of test: Dijkstra from a graph with vertices that is disconnected and the vertex of start don't connected with the rest of the graph in that direction** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | dijkstra() | setUpStage4 | String key | Map<String, Pair<Integer, String>> with elements |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: floydWarshallTestStandard()** | | | | |
| **Objective of test: Floyd-Warshall from a graph with vertices and edges** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | floydWarshall() | setUpStage3 | None | Map<String, Map<String, Integer>> with elements |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: floydWarshallTestLimit()** | | | | |
| **Objective of test: Floyd-Warshall from graph with one node** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | floydWarshall() | setUpStage5 | None | Map<String, Map<String, Integer>> with elements |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: floydWarshallTestInteresting()** | | | | |
| **Objective of test: Floyd-Warshall from a graph with vertices that is disconnected, and the vertex of start don't connected with the rest of the graph in that direction** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | floydWarshall() | setUpStage4 | None | Map<String, Pair<Integer, String>> with elements |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: primTestStandard()** | | | | |
| **Objective of test: Prim from a graph with vertices and edges** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | primMST() | setUpStage3 | None | Map<String, String> with elements |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: primTestLimit()** | | | | |
| **Objective of test: Prim from a graph with one vertex** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | primMST() | setUpStage5 | None | Map<String, String> with elements |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: primTestInteresting()** | | | | |
| **Objective of test: Prim from a graph with a cyclic** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | primMST() | setUpStage7 | None | Map<String, String> with elements |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: kruskalTestStandard()** | | | | |
| **Objective of test: Kruskal from a graph with vertices and edges** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | kruskalMST() | setUpStage3 | None | List<Edge<String>> not null |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: kruskalTestLimit()** | | | | |
| **Objective of test:** **Kruskal from a graph with one vertex** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | kruskalMST() | setUpStage5 | None | List<Edge<String>> not null |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name of test: kruskalTestInteresting()** | | | | |
| **Objective of test: Kruskal from a graph with a cyclic** | | | | |
| **Class** | **Method** | **Scenery** | **Input** | **Output** |
| AdjacencyMatriz or AdjacencyList | kruskalMST() | setUpStage7 | None | Map<String, String> with elements |