DOCUMENTATION DSIT2

Grupo: Martín Gómez, Julio Prado, Alejandro Mejía

ENGINEERING METHOD DEVELOPMENT

Phase 1: Problem identification

In the context of engineering, a maze can be seen as a network of interconnected paths, often with various obstacles and potential routes. The objective is to navigate from a starting point to the endpoint, while considering factors such as the efficiency of the path, energy consumption, and obstacle avoidance. Mazes are a funny hobby and serve as a mental exercise.

In this way, developing a software that automates the creation of paths within the maze would be beneficial. Also, adding enemies, and a mechanic for the player to kill them, makes the process more interactive and fun.

REQUIREMENT SPECIFICATION

Client: **User:**	**Marlon** Whoever wishes to play the game
Functional Requirements	**FR1. Generate Maze** **FR2. Add enemies** **FR3. Toggle Mode**
Problem Context	We want to create a maze game, where the player has to fight enemies on his way to the maze exit. The maze itself is to be modeled as a graph, where the player must go from an initial node to an end node. We want the game to be as interactive as possible, so the player will be something like a bomberman, where he can throw bombs to kill the enemies, and the enemies will follow him using the shortest possible path.
Non-functional requirements	**NFR1. Easy to use interface on any device** **NFR2. Easy maintenance and updating of the system.** **NFR3. Use of generics**

	NFR4. Use of graphs with at least 50 nodes and 50 connections.

Name and identifier	*[FR1 - Generate Maze]*		
Summary	*When the player presses the start game button, a maze will be generated.*		
Inputs	**Input Name**	**Data Type**	**Valid Value Conditions**
	N/A	N/A	N/A
Result or postcondition	*The maze is created.*		
Outputs	**Output Name**	**Data Type**	**Format**
	N/A	N/A	N/A

Name and identifier	*[FR2 - Add enemies]*		
Summary	*After the maze is generated and the game starts, the enemies will be added to the game, said enemies have the capability of following the player using the shortest path.*		
Inputs	**Input Name**	**Data Type**	**Valid Value Conditions**
	N/A		
Result or postcondition	Enemies generated.		
Outputs	**Output Name**	**Data Type**	**Format**
	N/A		

Name and identifier	*[FR3 - Toggle Mode]*

Summary	*The method that generates the map has two booleans, one representing whether the graph is directed, and one that controls the graph representation (whether it is represented as a matrix or a list). This makes it possible to toggle the representation, as per the given requirements, however it won't change anything within the game.*		
Inputs	**Input Name**	**Data Type**	**Valid Value Conditions**
	N/A	N/A	N/A
Result or postcondition	The graph representation changed, as the given requirements from the game state, we change the graph from an undirected to a directed one or vice versa, or from matrix to list or list to matrix.		
Outputs	**Output Name**	**Data Type**	**Format**
	N/A	N/A	N/A

Phase 2: Compilation of necessary information

DEFINITIONS

Graph: A graph is a mathematical object that can be seen as a set of vertices and edges. Vertices are points and edges represent a connection between two of these vertices.
Graphs have a wide range of applications, such as maps, social networks, machine learning and circuit design.

Types of graphs:

- Directed: The existence of an edge between the vertices v_1 and v_2 means that it is possible to go from v_1 to v_2, but not from v_2 to v_1 (unless there is another edge).
- Non-directed: The edges are bidirectional. The existence of an edge between the vertices v_1 and v_2 means that it is possible to go from v_1 to v_2 and from v_2 to v_1.
- Weighted: Each edge, in addition to the initial vertex and the final vertex, has a weight that represents the cost of using the connection.
- Non-weighted: The associated cost with using any kind of vertex is the same, 1.

Different graph algorithms:

Graph Traversals:

BFS (Breadth-First Search): Explores the graph level by level, revealing its structure in layers. Useful for finding the shortest distance between nodes and understanding graph connectivity.

DFS (Depth-First Search): Explores the graph in depth, unveiling complete branches of each node. Useful for cycle detection and discovering connected components.

Dijkstra's Algorithm: Finds the shortest path from starting node to all other nodes in a weighted graph. Highlights the most efficient connection in terms of weight.

Floyd-Warshall Algorithm: Computes the shortest paths between all pairs of nodes in a weighted graph. Provides a global view of accessibility between nodes, even considering negative weights.

Prim's Algorithm: Builds a minimum spanning tree, emphasizing edges that connect all nodes with the lowest total cost. Highlights essential connections to maintain graph connectivity.

Kruskal's Algorithm: Similar to Prim's algorithm, finds a minimum spanning tree by adding edges in ascending order of weight. Highlights the most efficient connections to preserve graph connectivity.

Bibliography:

Moreno, E., & Ramírez, H. (Año). Grafos: Fundamentos y Algoritmos. Universidad Adolfo Ibañez y Universidad de Chile. https://cmmedu.uchile.cl/repositorio/Instructional%20design%20%28of%20materiales%20or%20pedagogical%20models%29./Herramientas%20para%20la%20formaci%C3%B3n%20de%20profesores%20de%20matem%C3%A1tica/01%20-%20GRAFOS%20Fundamentos%20y%20algoritmos.pdf
https://www.geeksforgeeks.org

REQUIREMENT ELICITATION

Quality Function Deployment

In this technique customer satisfaction is of prime concern, hence it emphasizes on the requirements which are valuable to the customer. As the identification of stakeholders (in this case just user and client, which both are pretty much ourselves and the professor), and the listing of the requirements is already done from the first phase of the engineering method, we just need to classify 3 types of requirements:

- **Normal requirements :** In this the objective and goals of the proposed software are discussed with the customer.
- **Expected requirements :** These requirements are so obvious that the customer need not explicitly state them.
- **Exciting requirements :** It includes features that are beyond customer's expectations and prove to be very satisfying when present.

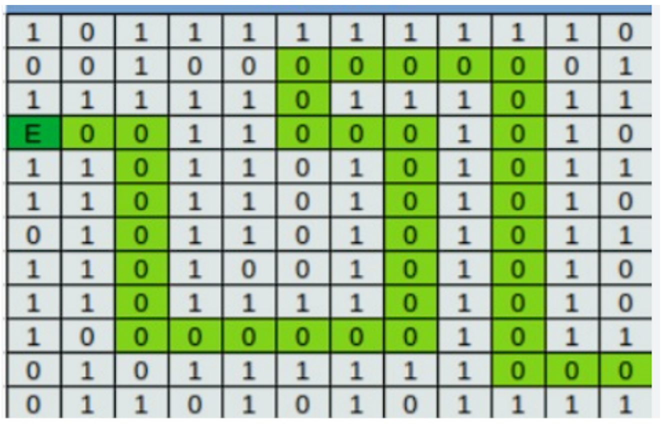
Listing of normal requirements: All functional requirements listed in the first phase of the engineering method (generate maze, add enemies, toggle mode).

Listing of expected requirements: Non functional requirements listed in the first phase of the engineering method(easy to use interface on any device, easy maintenance and updating of the system, use of generics, use of graphs with at least 50 nodes and 50 connections)

Listing of exciting requirements: The implementation of even more interactive features, like making the player able to drop bombs.

Phase 3: Creative solution searching

In order to find a solution more easily, by now, the problem was reduced to generate a binary grid NxN that serves as a maze blueprint.



1	0	1	1	1	1	1	1	1	1	1	0
0	0	1	0	0	0	0	0	0	0	0	1
1	1	1	1	1	0	1	1	1	0	1	1
E	0	0	1	1	0	0	0	1	0	1	0
1	1	0	1	1	0	1	0	1	0	1	1
1	1	0	1	1	0	1	0	1	0	1	0
0	1	0	1	1	0	1	0	1	0	1	1
1	1	0	1	0	0	1	0	1	0	1	0
1	1	0	1	1	1	1	0	1	0	1	0
1	0	0	0	0	0	0	0	1	0	1	1
0	1	0	1	1	1	1	1	1	0	0	0
0	1	1	0	1	0	1	0	1	1	1	1

In this matrix 1 represents a wall.

The idea consists on creating a graph in which each

Brainstorming

Maze Path Generation:

1) Brute force for Maze Path Generation

   One option could involve assigning 1 or 0 to each grid's entry randomly, except for the starting point and the final point which are set to 1 by default. Then the algorithm has to validate whether there is a valid path between the key points. The process is repeated until there is a valid path,

2) Randomized Depth-First Search (DFS) Maze Path Generation:

   - Initialize the maze grid with all cells set to 0, representing no connection between nodes.
   - Start at the entrance point (the starting point) and mark it as visited (set the value to 1).
   - Use a stack data structure to keep track of the current cell and its neighbors.
   - Repeatedly perform the following steps until the stack is empty:
     1. Pop a cell from the stack.
     2. Examine its unvisited neighbors (cells with a value of 0).
     3. If there are unvisited neighbors, pick one randomly.
     4. Mark it as visited (set the value to 1), indicating a connection, and remove the "wall" (the 0) between the current cell and the chosen neighbor.
     5. Push the chosen neighbor onto the stack.

3) Prim's Algorithm for Maze Path Generation:

   - Initialize the maze grid with all cells set to 0, signifying no connection between nodes.
   - Start with an empty set of cells (the "MST," Minimum Spanning Tree).
   - Pick a random cell and add it to the MST, marking it as visited (set the value to 1).
   - While the MST is not the entire maze:
     1. Randomly select a cell from the MST.
     2. Randomly select a neighboring cell not in the MST.
     3. Mark the neighboring cell as visited, set the value to 1, and remove the "wall" (the 0) between the selected cell and the neighboring cell.
     4. Add the neighboring cell to the MST.

4) Kruskal's Algorithm for Maze Path Generation:
    - Initialize the maze grid with all cells set to 0, indicating no connection between nodes.
    - Create a list of all walls (edges) in the maze.
    - Randomly shuffle the list of walls.
    - Start with an empty set of cells.
    - While the MST is not the entire maze:
        1. Take the next wall from the shuffled list.
        2. If the wall separates two cells, one marked as visited (value 1) and the other not visited (value 0), remove the "wall" (set the value to 1), indicating a connection between these cells

For the enemies short path finding

On the other hand, making the enemy follow the player requires knowing the length and the sequence of the shortest path in the maze.

1)BFS (Breadth-First Search)

- Place the start position in a queue, the enemy position.
- While the queue is not empty, dequeue the current position.
- Examine adjacent nodes. Enqueue unvisited nodes and mark the current node as visited.
- This is repeated until we find the player's position.

Important: BFS finds the shortest path but doesn't consider the actual cost of reaching each node. It's great for simplicity and speed and only works for unweighted graphs.

2) Dijkstra:

- Assign infinite cost to all nodes except the initial one, which gets a cost of zero.
- While there are nodes yet to be evaluated, select the node with the lowest current cost.
- Update the costs of adjacent nodes if a more efficient path is found through the current cell.
- This is repeated until we find the player's position.

Important: Dijkstra finds the shortest path considering the actual cost but can be computationally more expensive than BFS.

3) A*:

- Assign initial costs to nodes based on both the distance travelede and an estimated distance to the player, this is called the heuristic. Set the starting position.

- Maintain two lists, one called “open” which contains the nodes to be evaluated, and the other called “closed” which contains the nodes already evaluated.
- While the “open” list is not empty, select the nodes with the lowest total cost (sum of distance traveled and heuristic).
- Examine adjacent nodes. Update their costs and add them to the “open” list if they're not already there.
- This needs to be repeated until we find the player's position.

Important: A* is usually faster than Dijkstra because it employs a heuristic to guide the search and does not explore all directions equally.

Phase 4: Transition from idea formulation to preliminary design

For the maze generation the **brute force** alternative is **discarded**. The main issue with this approximation is that finding the matrix that satisfies the condition of having a valid path depends on probabilities. It makes it impossible to estimate the time performance of the algorithm, the other alternatives are much better.

Randomized DFS:
DFS tends to create mazes with long and winding paths, resulting in visually interesting and aesthetically pleasing mazes. This is because it needs to explore a path deeply before backtracking, and the use of a stack for backtracking makes the algorithm relatively easy to implement. However, DFS can generate mazes with dead ends, making paths less efficient in terms of length, and algorithmic complexity becomes an issue.

Prim’s:
Prim's algorithm creates mazes with a more even distribution of paths, resulting in a visually balanced maze. Additionally, the process of selecting neighboring cells from the Minimum Spanning Tree (MST) tends to result in mazes with fewer dead ends compared to DFS. However, this algorithm tends to have a more complex implementation than DFS due to the management of the MST and the selection of cells.

Kruskal:
Very similar to Prim's, but Kruskal considers walls and connections. Kruskal can result in mazes with fewer dead ends. The Kruskal algorithm requires additional data structures, such as disjoint sets, making it highly complex to implement compared to simpler algorithms, unlike Prim's, which has a lower complexity.

For the enemies short path finding

1)BFS (Breadth-First Search**)**

BFS involves initializing the starting position in a queue, then repeatedly dequeuing the current position, exploring adjacent nodes, enqueuing unvisited ones, and marking the current node as visited. This process continues until the player's position is found. BFS is great for finding the shortest path quickly but **doesn't consider the actual cost of reaching each node**, this works for unweighted graphs.

2) Dijkstra:
Dijkstra's starts by assigning infinite cost to all nodes except the initial one, which gets a cost of zero, can be the enemy. The main loop selects the nodes with the lowest current cost, updates the costs of adjacent nodes if a more efficient path is found through the current node, and repeats until the player's position is reached. Dijkstra's is effective in finding the shortest path considering the actual cost but can be computationally more expensive than BFS.

3) A*:
A* begins his journey by assigning initial costs to nodes based on both the distance traveled and an estimated distance to the player, known as the heuristic. Establish two lists, "open" for nodes to be evaluated, and "closed" for nodes already evaluated. While the "open" list persists, select nodes with the lowest total cost (sum of distance traveled and heuristic). Investigate adjacent nodes, update their costs, and add them to the "open" list if absent. This sequence repeats until the player's position is unveiled. Is important to consider that A* is generally more efficient than Dijkstra, due to its heuristic guided search.

Phase 5: Evaluation for the best option

Criteria

A) Precision of solution
   -2 Exact
   -1 Approximate
B) Efficiency
   -4 logarithm
   -3 lineal
   -2 cuasilineal
   -1 Quadratic
C) Completeness
   -3 All of them
   -2 More than one
   -1 None or only one.
D) Ease of algorithm implementation
   -3 pretty easy
   -2 medium
   -1 hard

To Maze Generation:

*In this case we are going to have one more Criteria

E) Memory Usage
   -3 Low Memory Usage
   -2 Medium Memory Usage
   -1 High Memory Usage

	Criteria A	Criteria B	Criteria C	Criteria D	Criteria E	Total
DFS	2	4	3	3	3	15
PRIM'S ALGORITHM	2	3	3	2	2	12
Kruskal	2	2	3	1	3	11

Based on these results, the Randomized DFS was chosen as the best option for random maze generation.

To Find the Shortest Path:

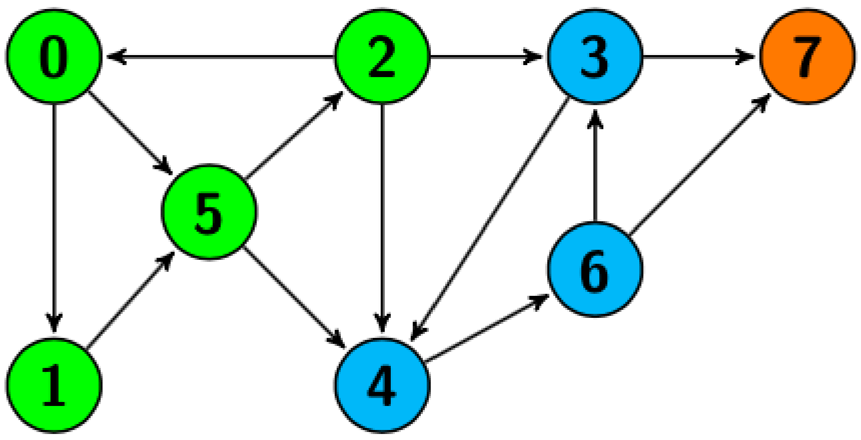
	Criteria A	Criteria B	Criteria C	Criteria D	Total
BFS	2	2	3	3	10
Dijkstra	2	3	3	1	9
A*	2	2	3	2	9

Based on these results, BFS was chosen as the best option for the enemies to find the shortest path to the player.

DESIGN

TAD OF USED DATA STRUCTURES

Graph



Attributes

V (vertexes)= {v_0,v_1,v_2….}

E (edges)= { (v_0,v_1,w),(v_1,v_2, w_2)....   }

inv: { $\Sigma_{v \in V} degree(v) \;=\; 2\,|E|$ }

inv: {$\forall v_1,\ v_2 \in V,\ (v_1 \neq v_2) \rightarrow (ID\ of\ v_1 \neq ID\ of\ v_2)$}

Primitive operations			type
create		Graph	constructor
addVertex	graph x vertex	Graph	modifier
addEdge	graph x initialVertex x finalVertex x weight	Graph	modifier
numEdges	graph	int	Analyzer
numVertex	graph	int	Analyzer
removeVertex	graph x Vertex	Graph	modifier

Graph():
“Creates an empty graph”
true
post{graph}

addVertex(graph x vertex):
“Adds a new vertex, with the given id, to the graph”
pre{vertex ∉ V}
post{V={v_0,v_1,v_2,,.... vertec}}

addEdge(graph x intialVertex x finalVertex x weight):
“Adds a connection,with the given weight between the initial vertex and the final vertex”
pre{intialVertex ∈V ∧ finalVertex ∈ V ∧ weight>=0}
post{E={(v_0,v_1,w),(v_1,v_2, w_2)....(initialVertex,finalVertex,weight)}}

numEdges(graph):
“Returns the number of edges in the graph”
pre{graph ≠ null}
post{returns int (with the number of edges)}

numVertex(graph):
“Returns the number of vertex in the graph”
pre{graph ≠ null}
post{returns int (with the number of vertex)}

removeVertex(graph x vertex):
“Remove a vertex from the graph”
pre{graph ≠ null ∧ vertexToRemove ∈ graph}
post{(∀e(edge) ∈ vertexToRemove → e ∉ vertexToRemove) ∧ vertexToRemove ∉ graph}

TESTS

Graph Structure Tests

addVertex(vertex)

Class	**Method**	**Scenario**	**Input**	**Expected Output**
Graph	addVertex(vertex)	Add a new vertex to an empty graph	Empty graph, and a vertex	A string representing all the vertices in the graph, only showing the vertex added
Graph	addVertex(vertex)	Add a new vertex to a graph that already has some vertices	A graph with vertex in it. and the new vertex	A string representing all vertices in the graph, including the new vertex added
Graph	addVertex(vertex)	Attempt to add a vertex with the same ID as an existing vertex.	Graph with vertex including the vertex V_2. And the new vertex duplicated V_2	A string representing all vertices in the graph, with only one vertex V_2. Then the vertex wasn´t added.

addEdge(graph x intialVertex x finalVertex x weight)

Class	**Method**	**Scenario**	**Input**	**Expected Output**
Graph	addEdge(graph x intialVertex x finalVertex x weight)	Add a connection between two existing vertices.	Graph with vertices V_1, V_2. Initial vertex: V_1 Final vertex: V_2 weight: n (positive)	A string that represents the connection between the two vertices and the edge weight. (V_1, V_2, n)

Graph	addEdge(graph x intialVertex x finalVertex x weight)	Add a connection between two existing vertices with a weight of 0.	Graph with vertices V_1, V_2. Initial vertex: V_1 Final vertex: V_2 weight: 0	A string that represents the connection between the two vertices and the edge weight 0. (V_1, V_2, 0)
Graph	addEdge(graph x intialVertex x finalVertex x weight)	Attempt to add a connection between two vertices, one of them don't exist in the graph.	Graph with vertex V_1. Initial vertex: V_1 Final Vertex: V_2 (Do not exist in the graph) weight: n (positive)	A string showing that one of the vertices does not exist inside the graph

numEdges(graph)

Class	**Method**	**Scenario**	**Input**	**Expected Output**
Graph	numEdges(graph)	Calculate the number of edges in the graph with three edges	Graph with edges (v_1, v_2, 5), (v_1, v_3, 1), (v_3, v_1, 2)	The method should return 3
Graph	numEdges(graph)	Calculate the number of edges in an empty graph	An empty graph	The method should return 0.
Graph	numEdges(graph)	Calculate the number of edges in a graph with a edge with a self-connection, like a loop.	Graph with edges (v_1, v_2, 5), (v_2, v_3, 1), (v_3, v_3, 2)	The method should return 3.

numVertex(graph)

Class	**Method**	**Scenario**	**Input**	**Expected Output**

Graph	numVertex(graph)	Calculate the number of vertices in an empty graph.	An empty graph	The method should return 0
Graph	numVertex(graph)	Calculate the number of vertices in a graph with multiple vertices.	Graph with vertices V_1, V_2, V_3	The method should return 3

removeVertex(graph x vertex)

Class	Method	Scenario	Input	Expected Output
Graph	removeVertex(graph x vertex)	Remove a vertex from a graph with 3 vertices	Graph with vertices V_1, V_2, V_3. Vertex to remove: V_2	The graph with the vertices: V_1, V_3
Graph	removeVertex(graph x vertex)	Remove a vertex from an empty graph	An empty graph. Vertex to remove: V_3	A string showing that the graph is empty.
Graph	removeVertex(graph x vertex)	Remove a vertex that is not included in the graph	Graph with vertices V_1, V_2, V_3. Vertex to remove: V_4	A string showing that the vertex does not exist in the graph.

Methods used for code implementation

Adjacency List Directed And Weighted Tests

Setup before each test: Initialize an empty adjacency list graph, directed and weighted

addNode(int n)

Class	Method	Scenario	Input	Expected Output
AdjacencyList Graph	addNode(int n)	Add a node with a number of 1. The try to add the same node with the same num 1.	int num = 1 and then again int num = 1	The first try is ok, then returns true. The second try is

				not possible because in a graph we can have a node with the same num, return false.

addEdge(int from, int to, int weight)

Class	**Method**	**Scenario**	**Input**	**Expected Output**

AdjacencyList Graph	addEdge(int from, int to, int weight)	Add a node with a number of 1 and other with the number of 2. Then try to make an edge between node 1 and node 2. And then try to make an edge between node 1 and node 3 (dont exist).	First try:
			int from = 1;
			int to = 2;
			
			Second try:
			int from = 1;
			int to = 3;
				The first try is valid, then returns true.
				
				The second try is not possible because the node 3 does not exist in the graph, return false.

nodeInfluenceDirectly(int from, int to)

Class	**Method**	**Scenario**	**Input**	**Expected Output**
AdjacencyList Graph	nodeInfluenceDirectly(int from, int to)	Add a node with a number of 1 and other with the number of 2. Then make an edge from node 1 to node 2. Try to see if node 1 have a direct influence on node 2. Try to see if node 2 have a direct influence on node 1.		The first try is valid, then returns true. The second try is not possible, return false.

toString()

Class	**Method**	**Scenario**	**Input**	**Expected Output**
AdjacencyList Graph	toString	Add a node with a number of 1 and other with the number of 2. Then make an edge from node 1 to node 2. Try to see if the expected string: “Adjacency List: 1-> { (2, 1) } 2-> {]” is equals to the toString()		In this case the expected String is gonna be equal to the toString.

Adjacency List Not Directed And Not Weighted Tests

Setup before each test: Initialize an empty adjacency list graph, not directed and not weighted

addNode(int n)

Class	**Method**	**Scenario**	**Input**	**Expected Output**
AdjacencyList Graph	addNode(int n)	Add a node with a number of 1 and then try to add the same node again.	int num = 1 and again: int num = 1	The first try is valid, returns true. The second try is not possible, return false.

addEdge(int from, int to, int weight)

Class	**Method**	**Scenario**	**Input**	**Expected Output**
AdjacencyList Graph	addEdge(int from, int to, int weight)	Add nodes with number 1 and 2. Add an edge from node 1 to node 2. And then try to	int num = 1 int num = 2	The first try is valid, returns true. The second try is

		make an edge between node 1 and node 3 (dont exist).		not possible, return false.

nodeInfluenceDirectly(int from, int to)

Class	Method	Scenario	Input	Expected Output
AdjacencyList Graph	nodeInfluenceDirectly(int from, int to)	Add a node with a number of 1 and other with the number of 2. Then make an edge from node 1 to node 2. Try to see if node 1 have a direct influence on node 2. Try to see if node 2 have a direct influence on node 1.	int num = 1 int num = 2	The first try is valid, returns true. The second try is, in this case, also valid because is a not directed graph, returns true.

toString()

Class	Method	Scenario	Input	Expected Output
AdjacencyList Graph	toString()	Add a node with a number of 1 and other with the number of 2. Then make an edge from node 1 to node 2. Try to see if the expected string: “Adjacency List: 1-> { (2) } 2-> { (1) }” is equals to the toString()		In this case the expected String is gonna be equal to the toString.

Adjacency Matrix Directed And Weighted Tests

Setup before each test: Initialize an empty adjacency matrix graph, directed and weighted

addNode(int n)

Class	Method	Scenario	Input	Expected Output
AdjacencyMatrixGraph	addNode(int n)	Add a node with a number of 1 and then try to add the same node again.	int num = 1 and again: int num = 1	The first try is valid, returns true. The second try is not possible, return false.

addEdge(int from, int to, int weight)

Class	Method	Scenario	Input	Expected Output
AdjacencyMatrixGraph	addEdge(int from, int to, int weight)	Add nodes with number 1 and 2. Add an edge from node 1 to node 2. And then try to make an edge between node 1 and node 3 (dont exist).	int num = 1 int num = 2	The first try is valid, returns true. The second try is not possible, return false.

nodeInfluenceDirectly(int from, int to)

Class	Method	Scenario	Input	Expected Output
AdjacencyMatrixGraph	nodeInfluenceDirectly(int from, int to)	Add a node with a number of 1 and other with the number of 2. Then make an edge from node 1 to node 2. Try to see if node 1 have a direct influence on node 2.	int num = 1 int num = 2	The first try is valid, returns true. The second try is, in this case, also valid because is a not directed graph, returns true.

		Try to see if node 2 have a direct influence on node 1.		

toString()

Class	Method	Scenario	Input	Expected Output
AdjacencyMatrixGraph	toString()	Add a node with a number of 1 and other with the number of 2. Then make an edge from node 1 to node 2. Try to see if the expected string: “Adjacency Matrix: _ 1 2 1 0 1 2 0 0” is equals to the toString()		In this case the expected String needs to be equal to the toString.

Adjacency Matrix Not Directed And Not Weighted Tests

Setup before each test: Initialize an empty adjacency matrix graph, not directed and not weighted

addNode(int n)

Class	Method	Scenario	Input	Expected Output
AdjacencyMatrixGraph	addNode(int n)	Add a node with a number of 1 and then try to add the same node again.	int num = 1 and again: int num = 1	The first try is valid, returns true. The second try is not possible, return false.

addEdge(int from, int to, int weight)

Class	Method	Scenario	Input	Expected

				Output
AdjacencyMatrixGraph	addEdge(int from, int to, int weight)	Add nodes with number 1 and 2. Add an edge from node 1 to node 2. And then try to make an edge between node 1 and node 3 (dont exist).	int num = 1 int num = 2	The first try is valid, returns true. The second try is not possible, return false.

nodeInfluenceDirectly(int from, int to)

Class	Method	Scenario	Input	Expected Output
AdjacencyMatrixGraph	nodeInfluenceDirectly(int from, int to)	Add a node with a number of 1 and other with the number of 2. Then make an edge from node 1 to node 2. Try to see if node 1 have a direct influence on node 2. Try to see if node 2 have a direct influence on node 1.	int num = 1 int num = 2	The first try is valid, returns true. The second try is, in this case, also valid because is a not directed graph, returns true.

toString()

Class	Method	Scenario	Input	Expected Output
AdjacencyMatrixGraph	toString()	Add a node with a number of 1 and other with the number of 2. Then make an edge from node 1 to node 2. Try to see if the expected string:		In this case the expected String needs to be equal to the toString.

		"Adjacency Matrix: _ 1 2 1 0 1 2 1 0" is equals to the toString()		

Map Generation Tests

generateMap(int row, int col, boolean adjacencyList, boolean directed, Graph graph)

Class	**Method**	**Scenario**	**Input**	**Expected Output**

MapGenerator()	generateMap()
	
	toString()
		Create a graph with a set of specifics nodes
		
		Create the expected toString of the matrix, which represents the map generated.
		
		Try to see if the expected string is equals to the toString() of the map
			int row = 21;
			int col = 21;
			boolean adjacencyList = false;
			Graph graph = { V_1, V_2, V_3, ...}
				In this case the expected String needs to be equal to the toString().

generateMap(int row, int col, boolean adjacencyList, boolean directed, Graph graph)

Adjacency List

Class	**Method**	**Scenario**	**Input**	**Expected Output**

MapGenerator()	generateMap()	Generate a new map with AdjacencyList, then try to see if the numbers of rows are the expected ones.
		And see if the numbers of
			int row = 5;
			int col = 5;
			boolean adjacencyList = true;
			directed = false;
			Graph graph = null
				In this case the expected rows and columns need to be equal to the map generated.

		columns are also the expected ones.		

generateMap(int row, int col, boolean adjacencyList, boolean directed, Graph graph)

Directed Graph

Class	**Method**	**Scenario**	**Input**	**Expected Output**

MapGenerator()	generateMap()	Generate a new map with a directed graph, then try to see if the numbers of rows are the expected ones. And see if the numbers of columns are also the expected ones.	int row = 5;
			int col = 5;
			boolean adjacencyList = false;
			directed = true;
			Graph graph = null
				In this case the expected rows and columns need to be equal to the map generated.

generateMap(int row, int col, boolean adjacencyList, boolean directed, Graph graph)

Adjacency List and Directed Graph

Class	**Method**	**Scenario**	**Input**	**Expected Output**

MapGenerator()	generateMap()	Generate a new map with adjacency list and a directed graph, then try to see if the numbers of rows are the expected ones. And see if the numbers of columns are also the expected ones.	int row = 5;
			int col = 5;
			boolean adjacencyList = true;
			directed = true;
			Graph graph = null
				In this case the expected rows and columns need to be equal to the map generated.

generateMap(int row, int col, boolean adjacencyList, boolean directed, Graph graph)

Adjacency List and not Directed Graph

Class	**Method**	**Scenario**	**Input**	**Expected Output**

MapGenerator()	generateMap()	Generate a new map with adjacency list and a not directed graph, then try to see if the numbers of rows are the expected ones.
		And see if the numbers of columns are also the expected ones.
			int row = 5;
			int col = 5;
			boolean adjacencyList = true;
			directed = false;
			Graph graph = null
				In this case the expected rows and columns need to be equal to the map generated.

generateMap(int row, int col, boolean adjacencyList, boolean directed, Graph graph)

Adjacency Matrix

Class	**Method**	**Scenario**	**Input**	**Expected Output**

MapGenerator()	generateMap()	Generate a new map with adjacency Matrix, then try to see if the numbers of rows are the expected ones.
		And see if the numbers of columns are also the expected ones.
			int row = 5;
			int col = 5;
			boolean adjacencyList = false;
			directed = false;
			Graph graph = null
				In this case the expected rows and columns need to be equal to the map generated.

generateMap(int row, int col, boolean adjacencyList, boolean directed, Graph graph)

Adjacency Matrix and Directed Graph

Class	**Method**	**Scenario**	**Input**	**Expected Output**

MapGenerator()	generateMap()	Generate a new map with adjacency Matrix and a Directed Graph, then try to see if the numbers of rows are the expected ones.
		And see if the numbers of
			int row = 5;
			int col = 5;
			boolean adjacencyList = false;
			directed = true;
			Graph graph = null
				In this case the expected rows and columns need to be equal to the map generated.

		columns are also the expected ones.		

generateMap(int row, int col, boolean adjacencyList, boolean directed, Graph graph)

Adjacency Matrix and Not Directed Graph

Class	**Method**	**Scenario**	**Input**	**Expected Output**

MapGenerator()	generateMap()	Generate a new map with adjacency Matrix and a not Directed Graph, then try to see if the numbers of rows are the expected ones.
		And see if the numbers of columns are also the expected ones.
			int row = 5;
			int col = 5;
			boolean adjacencyList = false;
			directed = false;
			Graph graph = null
				In this case the expected rows and columns need to be equal to the map generated.

Shortest Path Tests

getShortestPath(MatrixCor origin, MatrixCor destination, int[][] maze, boolean directed, boolean adjacencyList)

Class	**Method**	**Scenario**	**Input**	**Expected Output**

PathFinder()	getShortestPath(MatrixCor origin, MatrixCor destination, int[][] maze, boolean directed, boolean adjacencyList)	Create a 2D maze. Define an origin and a destination.
		
		Calculate the shortest path in the maze using the PathFinder algorithm
			Maze = 2D maze defined in the method
			
			Directed = false
			
			AdjacencyList = true;
				Check if the shortest path matches the expected result.

getShortestPath(MatrixCor origin, MatrixCor destination, int[][] maze, boolean directed, boolean adjacencyList)

Class	Method	Scenario	Input	Expected Output

PathFinder()	getShortestPath(MatrixCor origin, MatrixCor destination, int[][] maze, boolean directed, boolean adjacencyList)	Create a 2D maze with a starting point and without a path to the destination.
		
		Define an origin and a destination.
		
		Calculate the shortest path in the maze using the PathFinder algorithm
			Maze = 2D maze defined in the method;
			
			Directed = true;
			
			AdjacencyList = true;
				Check if the shortest path is an empty list, because the maze dont have a possible path to the destination.

getShortestPath(MatrixCor origin, MatrixCor destination, int[][] maze, boolean directed, boolean adjacencyList)

Class	Method	Scenario	Input	Expected Output

PathFinder()	getShortestPath(MatrixCor origin, MatrixCor destination, int[][] maze, boolean directed, boolean adjacencyList)	Create a 2D maze.
		
		Define an origin and a destination.
		
		Calculate the shortest path in the maze using the PathFinder algorithm.
		
		In this case we are going to try to find the shortest path in a Directed adjacency list.
			Maze = 2D maze defined in the method;
			
			Directed = true;
			
			AdjacencyList = true;
				Check if the shortest path matches the expected result.

getShortestPath(MatrixCor origin, MatrixCor destination, int[][] maze, boolean directed, boolean adjacencyList)

Class	Method	Scenario	Input	Expected Output

PathFinder()	getShortestPath(MatrixCor origin, MatrixCor destination, int[][] maze, boolean directed, boolean adjacencyList)	Create a 2D maze.
		
		Define an origin and a destination.
		
		Calculate the shortest path in the maze using the PathFinder algorithm.
		
		In this case we are going to try to find the shortest path in a Directed adjacency Matrix.
			Maze = 2D maze defined in the method;
			
			Directed = true;
			
			AdjacencyList = false;
				Check if the shortest path matches the expected result.

getShortestPath(MatrixCor origin, MatrixCor destination, int[][] maze, boolean directed, boolean adjacencyList)

Class	Method	Scenario	Input	Expected Output

PathFinder()	getShortestPath(MatrixCor origin, MatrixCor destination, int[][] maze, boolean directed, boolean adjacencyList)	Create a 2D maze.
		
		Define an origin and a destination.
		
		Calculate the shortest path in the maze using the PathFinder algorithm.
		
		In this case we are going to try to find the shortest path in a Undirected adjacency List.
			Maze = 2D maze defined in the method;
			
			Directed = false;
			
			AdjacencyList = true;
				Check if the shortest path matches the expected result.

getShortestPath(MatrixCor origin, MatrixCor destination, int[][] maze, boolean directed, boolean adjacencyList)

Class	**Method**	**Scenario**	**Input**	**Expected Output**

PathFinder()	getShortestPath(MatrixCor origin, MatrixCor destination, int[][] maze, boolean directed, boolean adjacencyList)	Create a 2D maze.
		
		Define an origin and a destination.
		
		Calculate the shortest path in the maze using the PathFinder algorithm.
		
		In this case we are going to try to find the shortest path in a Undirected adjacency Matrix.
			Maze = 2D maze defined in the method;
			
			Directed = false;
			
			AdjacencyList = false;
				Check if the shortest path matches the expected result.