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, difficult levels and a score system through the maze traveling makes the process more interactive and fun.

REQUIREMENT SPECIFICATION

Client: **User:**	**Marlon** Whoever wishes to play the game
Functional Requirements	**FR1. Add player** **FR2. Generate Maze** **FR3. Add enemies** **FR4. Calculate Score** **FR5. Change 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 adding different types of enemies and abilities to the player would be ideal.
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 - Add Player]*		
Summary	*When opening the game GUI, the player must enter a nickname for the score to be associated with later on.*		
Inputs	**Input Name**	**Data Type**	**Valid Value Conditions**
	nickname	String	
Result or postcondition	*The player class is initialized with the entered nickname, and the default abilities.*		
Outputs	**Output Name**	**Data Type**	**Format**
	N/A		

Name and identifier	*[FR2 - Generate Maze]*		
Summary	*After the nickname is entered and the player created, a select difficulty message is shown, and the player must select either easy or hard. After the difficulty is selected, the maze is generated accordingly, and the game starts.*		
Inputs	**Input Name**	**Data Type**	**Valid Value Conditions**
	difficulty	String	*"Easy" || "Hard"*
Result or postcondition	*The maze is created according to the selected difficulty, if the difficulty is easy, the maze will be a undirected graph (where the player can go back and forth on the maze), on the other hand, if the difficulty is hard, the maze will be a directed graph (the player will only be able to move forward), so, if he reaches a dead end, he will have to restart from a checkpoint (and restarting from a checkpoint will affect the score).*		
Outputs	**Output Name**	**Data Type**	**Format**

	selectDifficulty	String(Alert)	“Select a mode: ”

Name and identifier	*[FR3 - Add enemies]*		
Summary	*After the maze is generated and the game starts, the enemies will be added to the game. There will be three types of enemies: the first two will be goblins and bats (the enemies that will be present everywhere throughout the maze), and the last one will be the boss, a minotaur (that the player will encounter at the very end of the labyrinth, as the last obstacle to leave).*		
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	*[FR4 - Calculate Score]*		
Summary	*After the player clears the maze, his score is calculated as the following function:* *(length of the shortest possible path/length of the player’s path) *(100/ time in seconds taken to clear the maze) * (100/ time in seconds taken to defeat the boss) * (number of defeated enemies)* *Note that the length of the shortest possible path is calculated using bfs.*		
Inputs	**Input Name**	**Data Type**	**Valid Value Conditions**
Result or postcondition			

Outputs	**Output Name**	**Data Type**	**Format**
	score	String(Alert)	“Your score was: …“

Name and identifier	*[FR5 - Change Mode]*		
Summary	*The player may press the esc key to open a menu, where there will be 3 options: resume, restart, and change difficulty. The one that really matters is change the game difficulty, doing so will “restart” the game from the start with the new selected difficulty (actually a new game is created with the selected difficulty).*		
Inputs	**Input Name**	**Data Type**	**Valid Value Conditions**
	difficulty	String	*“Easy” || “Hard”*
Result or postcondition	The game mode changed, as the given requirements from the game state, we change the graph from an undirected to a directed one or vice versa.		
Outputs	**Output Name**	**Data Type**	**Format**
	selectDifficulty	String(Alert)	“Select a mode: ”

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.

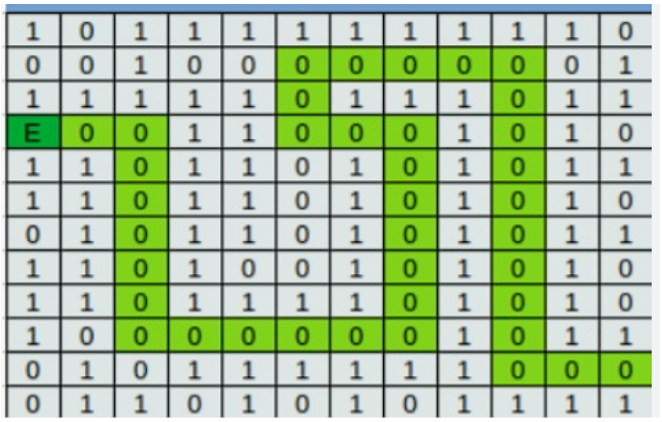
Listing of normal requirements: All functional requirements listed in the first phase of the engineering method (add player, generate maze, add enemies, calculate score, change mode).

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 adding items for the player.

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

Note that our implementation would be the opposite from this image (1's would represent the correct path)

If the grid's entry is equal to 1, that means that there are no obstacles in that position, otherwise the character of the video game cannot be present there.

Brainstorming

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) 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

On the other hand, computing the player's score requires knowing the length of the shortest path in the maze. There we have some ideas.

1)BFS (Breadth-First Search**)**
Bfs travels along a graph in order of levels. This means that nodes with distance 1 are visited first, then the nodes that have a distance 2 are visited, and so on. For calculating the length of the shortest path, we could keep track of that distance and then return that number when the final point is visited.

2) Dijkstra:
We can assign edge weights based on the perceived difficulty or “cost” of moving from one cell to another. In the context of the game the number of enemies in a particular area of the maze could be a relevant indicator of difficulty. For example, a cell with few enemies can have a lower weight, indicating that they are relatively easy areas to navigate. At the same

time, a cell with a lot of enemies can have a higher weight. In this case the easiest way is the best way, less enemies then more points.
3) Floyd-Warshall:
Floyd-Warshall ensures finding the lengths of the shortest paths between all pairs of nodes in the weighted graph of the maze. This provides an exact and complete solution for calculating the length of the shortest path between any pair of points. Again we can assign edge weights based on the perceived difficulty or "cost" of moving from one cell to another.

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.

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.

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:

*Just in this case we are going to have a couple of Criteria

E) Dead Ends
   -3 No dead ends
   -2 Few dead ends
   -1 Lot of dead ends

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

	Criteria A	Criteria B	Criteria C	Criteria D	Criteria E	Criteria F	Total
DFS	2	2	3	3	1	3	14
PRIM'S ALGORITHM	2	2	3	2	2	2	13
Kruskal	2	2	3	1	2	3	13

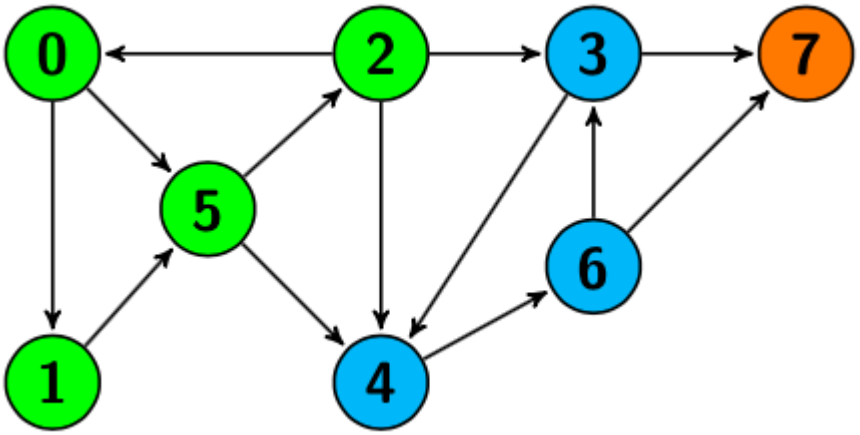
To Find the Shortest Path:

	Criteria A	Criteria B	Criteria C	Criteria D	Total

BFS	2	3	3	3	11
Dijkstra	2	4	3	2	11
Floyd-Warshall	2	1	3	1	7

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: $\{ \Sigma_{v \in V} degree(v) = 2 \lvert E \rvert \}$ 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 $\notin$ 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 $\in$ V $\wedge$ finalVertex $\in$ V $\wedge$ 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 $\neq$ null}
post{returns int (with the number of edges)}

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

removeVertex(graph x vertex):
“Remove a vertex from the graph”
pre{graph $\neq$ null $\wedge$ vertexToRemove $\in$ graph}
post{($\forall$ e(edge) $\in$ vertexToRemove $\rightarrow$ e $\notin$ vertexToRemove) $\wedge$ vertexToRemove $\notin$ graph}

Test

Name	**Scenery**
setUpEmptyGraph()	This method initializes an empty doubly linked list of Integers with the name of “list”.
setUpGraphWithVertex()	This method calls setUpEmptyList() and

	then adds elements to the list. In this case, numbers from 0 to 39 are added to the list.

Tests

Test Name	Used Setups	Method Under Test	Description
testGetFirstInstanceNull()	setUpEmptyList()	getFirstInstance()	Tests that the getFirstInstance() method returns null for each element in an empty list.
testGetFirstInstance()	setUpListWithElements()	getFirstInstance()	Tests that the getFirstInstance method correctly returns elements in a list with elements.
testDeleteFirstInstance()	setUpListWithElements()	removeFirstInstance()	Tests that the removeFirstInstance() method correctly removes the first instance of an element and returns **true**. Additionally, verifies that the method returns **false** when attempting to remove the same instance again.