**Development of engineering method:**

The goal of this project is to create a captivating and immersive maze game, leveraging graph-based structures, that provides an entertaining gaming experience while exploring the eerie and mysterious theme of the "**backrooms**."

**Project Objective:**

Our main objective is to design and implement a maze game with a "backrooms" theme that engages players in a thrilling adventure. This maze game will serve as a practical application of graph theory and algorithms. It will encompass various elements, from maze generation to player interaction, to create a seamless gaming experience.

1. ***Context and problem identification.***

***Identification of the needs:***

* We will employ graph structures to procedurally generate the maze. Each room in the maze will be represented as a vertex, and the connections between rooms as edges. This approach ensures that the maze is intricate and labyrinthine, capturing the essence of the "backrooms."
* A user-friendly interface is a critical component of our game. It will allow players to navigate, explore, and interact with the maze seamlessly. The interface will be designed to evoke the eerie and mysterious atmosphere of the "backrooms."
* Players will embark on a journey within the maze, facing challenges such as puzzles, riddles, and unexpected encounters. The game will offer an immersive experience, encouraging players to solve challenges to progress further through the maze.
* The main objective for the players is to find an escape route from the maze. This goal provides players with direction and purpose and drives their exploration and decision-making within the game.
* We will implement the maze structure using graph representations. Two versions, based on adjacency matrices and adjacency lists, will be created to ensure flexibility. These representations will be the backbone of the maze and will help in efficient pathfinding and exploration.
* To enhance the gaming experience, we will integrate graph algorithms such as depth-first search (DFS) and breadth-first search (BFS) for pathfinding and exploration within the maze. These algorithms will play a key role in guiding players through the complex network of rooms.

1. ***Information Gathering.***

*Hash Table:*

A Hash table is a data structure that stores some information, and the information has basically two main components, i.e., key and value. The hash table can be implemented with the help of an associative array. The efficiency of mapping depends upon the efficiency of the hash function used for mapping.

Font:

[**https://www.javatpoint.com/hash-table**](https://www.javatpoint.com/hash-table)

*Queues*

A Queue is defined as a linear data structure that is open at both ends and the operations are performed in First in First Out (FIFO) order. Typically, the end where items are added is referred to as the "back," "tail," or "rear" of the queue, while the end where items are removed is known as the "head" or "front" of the queue.

The operation of adding an element to the rear of the queue is known as enqueue, and the operation of removing an element from the front is known as dequeue.

Font:

<https://en.wikipedia.org/wiki/Queue_(abstract_data_type)>

[https://www.geeksforgeeks.org/](https://www.geeksforgeeks.org/queue-data-structure/)

*Stacks*

A stack is a sequential data structure that adheres to a specific execution order for its operations. This order can be either Last in First Out (LIFO), where the most recently added element is the first to be removed, or First in Last Out (FILO), where the initial element added is the last to be removed.

A stack features 2 primary operations:

* Push: This operation involves adding an element to the collection.
* Pop: It entails the removal of the most recently added element that has not been previously removed.

Font:

<https://www.geeksforgeeks.org/stack-data-structure/>

[https://en.wikipedia.org/wiki/Stack](https://en.wikipedia.org/wiki/Stack_(abstract_data_type))

*Generics*

Generics, in essence, refer to parameterized types. The concept revolves around enabling types (like Integer, String, user-defined types, etc.) to serve as parameters for methods, classes, and interfaces. Using Generics, it becomes possible to construct classes that can function with various data types. Any entity, whether it's a class, interface, or method, operating on a parameterized type, is considered a generic entity.

In contrast, the Object class stands as the superclass for all other classes, permitting an Object reference to point to any object.

Font:

<https://www.geeksforgeeks.org/generics-in-java/>

*Adjacency Matrix*

An adjacency matrix is ​​a square matrix of size N x N, where N is the number of vertexes in the graph, that represents the connections between edges of the graph. Each number of the matrix should be initialized as 0 or a value that represents the absence of a connection between the vertex and other, and if the graph is weighted, then the values represent the weight of the edge.

If the graph has only a few edges, the matrix is ​​sparse. Many graph algorithms such as Dijkstra's algorithm, Floyd-Warshall algorithm, and Kruskal's algorithm use adjacency matrices to represent graphs. Its advantages are that the adjacency matrix is ​​simple and easy to understand, adding or removing edges from your diagram is quick and easy and it allows time-constant access to every edge in the graph.

Font:

[https://www.geeksforgeeks.org/adjacency-matrix](https://www.geeksforgeeks.org/adjacency-matrix-meaning-and-definition-in-dsa/)

*Adjacency List*

An adjacency list represents a graph as an array of linked lists. The index of the array represents a vertex, and each element in its linked list represents other vertices that form an edge with that vertex. Adjacency lists are efficient in terms of storage because we only need to store the edge values. This can result in significant space savings for sparse graphs with millions of vertices and edges and it also helps to easily find all vertices adjacent to a vertex.

Looking up a neighbor list is no faster than looking up an adjacency matrix because all connected nodes must first be checked to find them.

Font:

<https://www.programiz.com/dsa/graph-adjacency-list>

*Graph*

Graphs are nonlinear data structures composed of vertices and edges. Vertices are sometimes called nodes, and edges are lines or arcs that connect any two nodes in the graph. Formally, a graph consists of a set of vertices (V) and a set of edges (E). The graphs have two ways of being represented; Adjacency Matrix and Adjacency List. The main components of the graph, as we said, are:

* Vertex: A vertex is the basic unit of a graph. Vertices are sometimes also called vertices or nodes. Each node/vertex can be labeled or unlabeled, this vertex may also contain any object we may want to use do operations in the future, like in a social media, every vertex would be a person.
* Edges: Draw or use edges to connect two nodes of a graph. It can be an ordered pair of nodes in a directed graph. An edge can connect any two nodes in any way. There are no rules. Sometimes edges are also called arcs. Each edge can be marked/unmarked. These edges may be directed or undirected, if they are directed then the connection follows a specific direction but if is undirected then the connection is bidirectional. Edges may also have a weight property that represents the length or any value depending on the context we are working in, to determine connection between two vertexes.

Graphs are effective tools for representing complex data, especially when the relationships between the data points are not straightforward. They can help to uncover patterns, trends, and insights that may be difficult to see using other methods.

Font:

[https://www.geeksforgeeks.org/graph-data-structure-](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/)

*BFS*

Breadth-first search is a graph traversal algorithm that traverses the graph starting from the root node and checking all adjacent nodes. Then it selects the nearest node and explores all unexplored nodes. When using BFS traversal, each node in the graph can be considered a root node.

It is a recursive algorithm for searching all vertices of a tree or graph data structure. BFS assigns each vertex of the graph to two classes: visited and unvisited. It selects a single node in the graph and then visits all nodes adjacent to the selected node. This algorithm follows the next steps widely:

1. Step 1: SET STATUS = 1 (ready state) for each node in G
2. Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)
3. Step 3: Repeat Steps 4 and 5 until QUEUE is empty.
4. Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).
5. Step 5: Enqueue all the neighbors of N that are in the ready state (whose STATUS = 1) and set
6. their STATUS = 2 (waiting state)
7. [END OF LOOP]
8. Step 6: EXIT

Time complexity of BFS depends upon the data structure used to represent the graph. The time complexity of BFS algorithm is O(V+E), since in the worst case, BFS algorithm explores every node and edge.

Font:

[https://www.javatpoint.com/BFS](https://www.javatpoint.com/breadth-first-search-algorithm)

*DFS*

An approach for navigating or searching across tree or graph data structures is called depth-first search. Starting from the root node (assuming, in the case of a graph, an arbitrary node), the method proceeds as far as it can along each branch before turning around. This algorithm is implemented by using a stack, as we are going to see in the next steps:

1. Step 1: Create a stack with the total number of vertices in the graph as the size.
2. Step 2: Choose any vertex as the traversal's beginning point. Push a visit to that vertex and add it to the stack.
3. Step 3 - Push any non-visited adjacent vertices of a vertex at the top of the stack to the top of the stack.
4. Step 4 - Repeat steps 3 and 4 until there are no more vertices to visit from the vertex at the top of the stack.
5. Step 5 - If there are no new vertices to visit, go back and pop one from the stack using backtracking.
6. Step 6 - Continue using steps 3, 4, and 5 until the stack is empty.
7. Step 7 - When the stack is entirely unoccupied, create the final spanning tree by deleting the graph's unused edges.

Fonts:

[https://www.geeksforgeeks.org/dfs](https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/)

[https://www.simplilearn.com/tutorials/dfs](https://www.simplilearn.com/tutorials/data-structure-tutorial/dfs-algorithm#:~:text=The%20depth%2Dfirst%20search%20or,far%20as%20possible%20before%20backtracking.)

*Dijkstra*

The algorithm determines the shortest path between each source node, a structure, and every other node it is most used in directed graphs with unbounded non-negative weights, but it also works for undirected graphs. By pausing the process once the shortest path to the target node has been found, it may also be used to find the shortest pathways from a single node to a single destination node. Dijkstra's algorithm can be used to find the shortest route between any two cities, for example, if the nodes of the graph represent cities and the costs of edge paths represent driving distances between pairs of cities connected by a direct road (ignore stop signs, red lights, toll roads, and other obstructions for simplicity). Network routing protocols are one common application of shortest route algorithms, most. It commonly follows the next steps:

1. Assign a current distance of 0 to the source node and infinite to the remaining nodes.
2. Assign the current node to the non-visited node that is currently the closest.
3. The current node N adds the weight of the edge linking 0 to 1 to the current distance of each neighboring node. Set it as the new current distance of N if it is less than the current distance of Node.
4. Designate node 1 as visited at this time.
5. If nodes remain unvisited, go to step 2.

To implement the algorithm, there are also several options, but most common are:

There are several ways to Implement Dijkstra’s algorithm, but the most common ones are:

1. Using priority queues to keep track of all vertices.
2. Using an array to keep track of Distances.
3. Using a set to keep track of the visited vertices.

In this project we will do it using a priority queue.

Fonts:

[https://www.geeksforgeeks.org/dijkstras-shortest-path](https://www.geeksforgeeks.org/introduction-to-dijkstras-shortest-path-algorithm/)

[https://en.wikipedia.org/wiki/Dijkstra\_algorithm](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm)

1. ***Alternatives:***

*Alternative 1/ Quadtree-Based Maze Generation:*

* 1. The maze is represented using a quadtree data structure, providing a hierarchical and spatially partitioned layout. The variable-sized nodes in the quadtree allow for the creation of diverse room sizes, adding visual interest to the maze. The hierarchical structure facilitates efficient spatial partitioning, aiding in collision detection and performance optimization. The depth of the quadtree can be adjusted dynamically to control the overall complexity of the maze, offering a flexible approach to maze generation.
  2. This alternative emphasizes efficiency and visual variety, creating a maze environment that is visually engaging and adaptable to different levels of player preference for complexity. The quadtree structure is utilized to achieve a balance between performance optimization and dynamic maze layouts, providing a unique experience for players.

*Alternative 2/ Linked Data Structure for Room Interactions:*

* 1. The maze is modeled as a linked list, creating a sequential progression through rooms. Each room node contains a small graph structure to represent interactions, challenges, or events within that room. The linked data structure allows for dynamic room connections, enabling the addition or removal of rooms during gameplay.
  2. Events associated with specific room nodes trigger puzzles or narrative elements, providing a story-driven experience. This alternative focuses on creating a linear and event-driven progression, guiding players through a curated sequence of challenges while allowing for flexibility in the maze structure. The combination of a linked list and small graph structures within rooms aims to balance player-guided exploration with a structured narrative experience.

*Alternative 3/ Graph-Based Maze with Dynamic Graph Traversal:*

* 1. The maze is conceptualized as a graph, where each tile on the map represents a vertex connected by edges, and the vertexes will be stored in a Hash Table in order to improve the efficiency of the operations that imply searching, deleting, and consulting. Dynamic graph traversal algorithms, such as Breadth-First Search (BFS) and Depth-First Search (DFS), are implemented to allow players to explore the maze in different ways. The graph structure enables dynamic changes during gameplay, introducing new connections or altering the maze's topology based on player actions. Player-triggered events associated with specific vertices enhance the interactive aspect, ensuring that each room in the maze can hold surprises, challenges, or narrative elements. prioritizing adaptability and strategic exploration, offering players the freedom to choose their paths.

1. ***Preliminary Designs:***

*Alternative 1 / Quadtree-Based Maze Generation:*

**Maze Generation with Quadtree:**

* + 1. Instead of representing the maze as a traditional graph, use a quadtree data structure to procedurally generate the maze. Divide the game area into quadrants recursively, creating a hierarchical structure. Each leaf node of the quadtree represents a room or a section of the maze.
  1. **Variable Room Sizes:**
     1. Quadtree-based generation allows for variable-sized rooms, introducing diversity in the maze layout. Larger rooms can contain complex puzzles or challenges, while smaller rooms may serve as connecting corridors.
  2. **Efficient Spatial Partitioning:**
     1. Quadtree provides efficient spatial partitioning, which can aid in collision detection and optimization, especially in a 2D game environment. This can improve the overall performance and responsiveness of the game.
  3. **Dynamic Maze Complexity:**
     1. Adjust the depth of the quadtree dynamically to control the overall complexity of the maze. Deeper levels may result in more intricate mazes with smaller rooms, while shallower levels can simplify the maze for a different gaming experience.
  4. **Pathfinding with Quadtree:**
     1. Implement pathfinding algorithms tailored for quadtree structures. This might involve traversing the quadtree to find the optimal path between rooms or using hierarchical pathfinding approaches.
     2. Quadtree structures may not represent all types of maze layouts effectively, potentially limiting the variety of possible maze structures.

*Alternative 2 / Linked Data Structure for Room Interactions*

* 1. **Linked List for Room Sequencing:**
     1. Represent the maze as a linked list where each node corresponds to a room. This data structure can encapsulate the sequential order of rooms, creating a linear progression through the maze.
  2. **Dynamic Room Connections:**
     1. Use dynamic memory allocation to allow for flexible room connections. Rooms can be added or removed during gameplay, altering the overall structure of the linked list and providing a dynamic maze experience.
  3. **Event-Driven Interactions:**
     1. Associate events with each room node in the linked list. Events trigger when players enter a room, introducing challenges, puzzles, or narrative elements. This approach allows for a more event-driven and story-driven gameplay experience.
  4. **Graph Representation for Room Interactions:**
     1. Within each room, represent interactions (such as puzzles or challenges) using a small graph structure. This graph can be used to model the dependencies and relationships between different elements within a room.
  5. **Cyclic Paths and Non-linear Progression:**
     1. Introduce the possibility of cyclic paths or non-linear progression through the linked list, allowing players to backtrack or take different routes. This adds complexity and variety to the maze exploration.
  6. Player Understanding:
     1. Communicating the sequential nature of the linked list and the interconnectedness of rooms to players may pose a challenge, potentially leading to confusion.

*Alternative 3/ Graph-Based Maze with Dynamic Graph Traversal:*

* 1. Graph Representation:
     1. Represent the maze as a graph where each tile is a vertex. Adjacent tiles are connected by edges. The graph can be implemented using an adjacency list or adjacency matrix, providing an efficient representation for navigation.
  2. BFS and DFS for Exploration:
     + 1. Implement Breadth-First Search (BFS) and Depth-First Search (DFS) algorithms to facilitate maze exploration. Players can choose between these traversal methods, each offering a unique approach to navigating the maze.
     1. BFS for Efficiency:
        1. Use BFS for finding the shortest path between two points in the maze. This can be particularly useful when players are seeking the optimal route to escape the maze. BFS ensures that the first solution found is the shortest.
     2. DFS for Non-linear Exploration:
        1. Allow players to use DFS for non-linear exploration, encouraging them to take less direct routes or discover hidden areas. This adds an element of strategy and encourages players to thoroughly explore the maze.
  3. Dynamic Graph Evolution:
     1. Introduce dynamic changes to the graph during gameplay. For example, doors may open, creating new connections between previously isolated parts of the maze, or some paths may become blocked, altering the graph structure dynamically.
  4. Dijkstra's Algorithm for Dynamic Challenges:
     1. Implement Dijkstra's algorithm for finding the shortest paths with weighted edges. Assign weights to edges based on dynamic factors such as changing obstacles or challenges. This adds an extra layer of complexity to the maze, requiring players to adapt to evolving conditions.
  5. Weighted Challenges:
     1. Assign weights to edges based on the difficulty of traversal, creating paths with different levels of challenge. Players may need to strategize and choose paths that align with their abilities and resources.
  6. Player-Triggered Events:
     1. Associate events with specific vertices in the graph. When a player reaches a particular vertex, trigger events such as puzzles, encounters, or changes in the maze layout. This ensures that the gameplay is dynamic and responsive to player actions.
  7. Graph Visualization:
     1. Provide a visual representation of the graph, allowing players to see the connections between tiles and understand the structure of the maze. This can be particularly helpful for planning routes and navigating efficiently.
     2. Communicating the graph structure visually to players may be challenging, potentially leading to confusion during navigation.

1. ***Selection of the best alternative***

* Criterium A: This criterion assesses the level of complexity associated with implementing each alternative. The scale ranges from highly complex and challenging (1) to relatively simple and straightforward (4), considering factors such as coding problems, potential technical difficulties:
  + - 1: Highly complex and challenging to implement.
    - 2: Moderately complex with some challenges.
    - 3: Manageable complexity with clear implementation paths.
    - 4: Relatively simple and straightforward to implement.
* Criterium B: This criterion evaluates the degree to which each alternative engages players throughout the gaming experience. The scale ranges from low engagement with limited interactive elements (1) to exceptional engagement with dynamic and captivating gameplay (4). Player engagement is a crucial aspect of game design, encompassing elements that captivate, challenge, and immerse players:
  + - 1: Low player engagement limited interactive elements.
    - 2: Moderate engagement with some interactive features.
    - 3: High engagement, offering diverse and immersive experiences.
    - 4: Exceptional engagement, providing dynamic and captivating gameplay.
* Criterium C: This criterion assesses how adaptable each alternative is during the development process, considering its flexibility for iterative changes and refinements. The scale ranges from limited flexibility, making it challenging to iterate on design (1), to exceptional flexibility, where the alternative is easy to test and refine during development (4). The is crucial, allowing to adjust, improvements, and optimizations as needed throughout the development lifecycle:
  + - 1: Limited flexibility, challenging to iterate on design.
    - 2: Moderate flexibility with some room for adjustments.
    - 3: High flexibility, allowing for iterative development.
    - 4: Exceptional flexibility, easy to test and refine during development.
* Criterium D: This criterion evaluates the efficiency of each alternative in terms of resource utilization, encompassing both time and space requirements. The scale ranges from highly resource-intensive, demanding extensive time and space (1), to resource-efficient, indicating optimal use of time and space for the given task (4). Resource efficiency is crucial to ensure that the chosen alternative operates smoothly, without placing excessive demands:
  + - 1: Highly resource-intensive, demanding extensive time and space.
    - 2: Moderately resource-intensive, with notable demands on time and space.
    - 3: Balanced resource utilization, manageable time, and space requirements.
    - 4: Resource-efficient, optimal use of time and space for the given task.

Alternative 1: Quadtree-Based Maze Generation

* + - Implementation Complexity - 2: Quadtree-based maze generation introduces a moderate level of complexity, with challenges associated with spatial partitioning and hierarchical structures.
    - Player Engagement - 2: The variable room sizes and efficient spatial partitioning contribute to player engagement, but it may not provide the same level of dynamic interaction as Alternative 3.
    - Flexibility for Iteration: - 3: The ability to adjust the depth of the quadtree dynamically allows for flexibility, enabling iterative development and changes to maze complexity.
    - Resource Efficiency - 3: Quadtree structures, when implemented well, offer efficient spatial partitioning and resource utilization, contributing to resource efficiency.

Alternative 2: Linked Data Structure for Room Interactions

* + - Implementation Complexity – 2 : The linked data structure introduces moderate complexity, with challenges related to dynamic room connections and player-triggered events.
    - Player Engagement - 3 : The linked data structure, with event-driven interactions and sequential progression, contributes to a high level of player engagement.
    - Flexibility for Iteration - 2:While the linked data structure allows for dynamic room connections, making significant changes may be moderately challenging.
    - Resource Efficiency - 2:The use of linked data structures requires careful management of memory, and efficiency may be moderate in terms of both time and space.

Alternative 3: Graph-Based Maze with Dynamic Graph Traversal

* Implementation Complexity - 3:This alternative involves a moderate level of complexity. While dynamic graph traversal can be challenging, the overall implementation path is relatively clear.
* Player Engagement - 3:The use of dynamic graph traversal and player-triggered events contributes to high player engagement, providing diverse and interactive experiences.
* Flexibility for Iteration - 3 :The graph-based structure allows for iterative development with flexibility, accommodating changes and refinements throughout the process.
* Resource Efficiency - 3:Dynamic graph traversal, when implemented efficiently, can strike a good balance between computational resources, contributing to resource efficiency.

In summary:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | ***Criterium 1*** | ***Criterium 2*** | ***Criterium 3*** | ***Criterium 4*** | ***Total*** |
| ***Alternative 1*** | ***2*** | ***2*** | ***3*** | ***3*** | ***10*** |
| ***Alternative 2*** | ***2*** | ***3*** | ***2*** | ***2*** | ***9*** |
| ***Alternative 3*** | ***3*** | ***3*** | ***3*** | ***3*** | ***12*** |

* In conclusion, we are going to use alternative #3 where the maze is modeled as a graph and implements transversal algorithms and Dijkstra.

1. ***Preparation for reports and specifications:***

* Situation: Modelate the graph structure using adjacency matrix and adjacency list. (All operations related to this).
* For Adjacency matrix

Function addVertex(key, data)

if vertices contains key then

throw InvalidEntriesException("Vertex with the same key already exists.")

end if

if data is null then

throw InvalidEntriesException("Vertex data cannot be null.")

end if

for each vertex in vertices do

if vertex.getData() equals data then

throw InvalidEntriesException("Vertex with the same data already exists.")

end if

end for

newVertex ← createVertex(key, data, numVertices)

vertices.put(key, newVertex)

newAdjMatrix ← createNewAdjMatrix(numVertices + 1)

for i ← 0 to numVertices do

for j ← 0 to numVertices do

if i < numVertices and j < numVertices then

newAdjMatrix[i][j] ← adjMatrix[i][j]

else

newAdjMatrix[i][j] ← -1

end if

end for

end for

numVertices ← numVertices + 1

adjMatrix ← newAdjMatrix

end procedure

* For adjacency List:

function addVertex(key, data)

if key is not null and data is not null and vertices does not contain key then

vertex ← createVertexList(key, data)

vertices.put(key, vertex)

end if

end procedure

* Creating the connections for adjacency matrix:

function addEdge(sourceKey, destinationKey, weight)

if weight < 0 then

throw InvalidEntriesException("Edge weight cannot be negative.")

end if

sourceVertex ← vertices.get(sourceKey)

destinationVertex ← vertices.get(destinationKey)

if sourceVertex is null or destinationVertex is null then

throw IllegalArgumentException("One or both vertices not found in the graph.")

end if

if sourceVertex equals destinationVertex then

throw InvalidEntriesException("Cannot add an edge that points to itself.")

end if

sourceVertex.addEdge(destinationVertex, weight)

sourceIndex ← sourceVertex.getIndex()

destinationIndex ← destinationVertex.getIndex()

adjMatrix[sourceIndex][destinationIndex] ← weight

end procedure

* Creating the connections for adjacency list:

function addEdge(sourceKey, destinationKey, weight)

origin ← vertices.get(sourceKey)

destiny ← vertices.get(destinationKey)

if origin is not null and destiny is not null then

origin.addEdge(destiny, weight)

origin.addVertex(destiny)

else

throw IllegalArgumentException("Vertex not found.")

end if

end procedure

1. Design implementation. (The subroutine is constructed in the java programming language, below the definition of each one ).

R1. Generate Labyrinth.

R2. Verify Possible paths to end.

R3. Verify shortest path.

R4. Move character.

R5. Calculate player score.

R6. Generate treasures.

R7. Player inventory.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name and ID** | **R1: Generate Labyrinth** | | |
| **Review** | *The system must generate the labyrinth in which the game will take place* | | |
| **Inputs** | **Input name** | **Data type** | **Valid values** |
| mazeConfiguration | Configuration object | Define maze parameters |
| **Result** | The labyrinth is generated | | |
| **Outputs** | **Output name** | **Data type** | **Format** |
| labyrinth | Graph | Representation of maze |

|  |
| --- |
| *Creation of the graph* |
| public class GraphList<K,T> implements IGraph<K,T> {  private Hashtable<K, VertexList<T,K>> vertices;  public GraphList() {  this.vertices = new Hashtable<>();  } |

|  |
| --- |
| *Creation of the graph* |
| public class GraphMatriz<K,T> implements IGraph<K,T> {  private Hashtable<K, Vertex<T,K>> vertices;  private int[][] adjMatrix;  private int numVertices;  public GraphMatriz() {  this.vertices = new Hashtable<>();  this.numVertices = 0;  this.adjMatrix = new int[numVertices][numVertices];  for (int i = 0; i < numVertices; i++) {  for (int j = 0; j < numVertices; j++) {  adjMatrix[i][j] = -1;  }  }  } |

|  |
| --- |
| *In game Maze* |
| public class Maze extends GraphList<String, Cell> {  public Maze() {  super();  }  public void addCell(int row, int col, CellType type) {  Cell cell = new Cell(row, col, type);  String vertexKey = generateKey(row, col);  addVertex(vertexKey, cell);  }  public void connectCells(Cell cell1, Cell cell2) {  if (canConnect(cell1, cell2)) {  String key1 = generateKey(cell1.getRow(), cell1.getCol());  String key2 = generateKey(cell2.getRow(), cell2.getCol());  addEdge(key1, key2, 1);  }  }  public VertexList<Cell, String> getCell(int row, int col) {  String key = generateKey(row, col);  return getVertex(key);  }  public int getCellSizeX() {  return Cell.SIZE;  }  public int getCellSizeY() {  return Cell.SIZE;  }  public List<VertexList<Cell, String>> getNeighbors(VertexList<Cell, String> vertex) {  List<VertexList<Cell, String>> neighbors = new ArrayList<>();  // Obtén las celdas adyacentes a la celda representada por el vértice  int row = vertex.getData().getRow();  int col = vertex.getData().getCol();  if (row > 0) {  neighbors.add(getCell(row - 1, col));  }  if (row < 49) {  neighbors.add(getCell(row + 1, col));  }  if (col > 0) {  neighbors.add(getCell(row, col - 1));  }  if (col < 49) {  neighbors.add(getCell(row, col + 1));  }  return neighbors;  }  public List<VertexList<Cell, String>> findShortestPath(Cell startCell, Cell endCell) {  String startKey = generateKey(startCell.getRow(), startCell.getCol());  String endKey = generateKey(endCell.getRow(), endCell.getCol());  return dijkstra(startKey, endKey);  }  private boolean canConnect(Cell cell1, Cell cell2) {  return cell1.getType() == CellType.EMPTY && cell2.getType() == CellType.EMPTY;  }  public String generateKey(int a, int b){  return (a+"-"+b);  }  public VertexList<Cell, String> getVertex(String key) {  return getVertices().get(key);  }  public int getNumRows() {  return 50;  }  public int getNumCols() {  return 50;  }  } |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name and ID** | **R2**: Verify possible paths to the exit. | | |
| **Review** | *The system must verify and store the possible paths leading from the start to the exit using some search algorithm, either DFS or BFS.* | | |
| **Inputs** | **Input name** | **Data type** | **Valid values** |
| mazeGraph | Graph | *Representation of the labyrinth* |
| **Result** | All the possible paths are stored in a linked list composed of others linked list of vertexes. | | |
| **Outputs** | **Output name** | **Data type** | **Format** |
| paths | LinkedList | Linked list of linked lists |

|  |
| --- |
| Bfs method for Adjacency List Graph. |
| public String bfs(K startVertexKey) {  VertexList<T,K> startVertex = vertices.get(startVertexKey);  StringBuilder result = new StringBuilder();  if (startVertex != null) {  Queue<VertexList<T,K>> queue = new LinkedList<>();  queue.add(startVertex);  startVertex.setVisited(true);  while (!queue.isEmpty()) {  VertexList<T,K> currentVertex = queue.poll();  result.append("Visited: ").append(currentVertex.getId()).append("\n");  for (VertexList<T,K> adjacentVertex : currentVertex.getAdjacencyList()) {  if (!adjacentVertex.isVisited()) {  queue.add(adjacentVertex);  adjacentVertex.setVisited(true);  }  }  }  resetVisitedState();  } else {  result.append("Vertex not found.");  }  return result.toString();  } |

|  |
| --- |
| Dfs method for Adjacency List Graph. |
| public String dfs(K startVertexKey) {  VertexList<T, K> startVertex = vertices.get(startVertexKey);  StringBuilder result = new StringBuilder();  if (startVertex != null) {  Stack<VertexList<T, K>> stack = new Stack<>();  stack.push(startVertex);  while (!stack.isEmpty()) {  VertexList<T, K> currentVertex = stack.pop();  if (!currentVertex.isVisited()) {  result.append("Visited: ").append(currentVertex.getId()).append("\n");  currentVertex.setVisited(true);  List<VertexList<T, K>> reversedAdjacents = new ArrayList<>(currentVertex.getAdjacencyList());  Collections.reverse(reversedAdjacents);  for (VertexList<T, K> adjacentVertex : reversedAdjacents) {  if (!adjacentVertex.isVisited()) {  stack.push(adjacentVertex);  }  }  }  }  resetVisitedState();  } else {  result.append("Vertex not found.");  }  return result.toString();  } |

|  |
| --- |
| Bfs method for Adjacency Matrix Graph. |
| public String bfs(K startVertexKey) {  StringBuilder result = new StringBuilder();  try {  Vertex<T,K> startVertex = vertices.get(startVertexKey);    if (startVertex == null) {  throw new IllegalArgumentException("Start vertex not found.");  }  Queue<Vertex<T,K>> queue = new LinkedList<>();  queue.add(startVertex);  startVertex.setVisited(true);  while (!queue.isEmpty()) {  Vertex<T,K> currentVertex = queue.poll();  result.append("Visited: ").append(currentVertex.getId()).append("\n");  for (Edge<T,K> edge : currentVertex.getAristas()) {  Vertex<T,K> adjacentVertex = edge.getDestination();  if (!adjacentVertex.isVisited()) {  queue.add(adjacentVertex);  adjacentVertex.setVisited(true);  }  }  }  resetVisitedState();  } catch (IllegalArgumentException e) {  result.append("Error: ").append(e.getMessage()).append("\n");  }  return result.toString();  } |

|  |
| --- |
| Dfs method for Adjacency Matrix Graph. |
| public String dfs(K startVertexKey) {  StringBuilder result = new StringBuilder();  try {  Vertex<T,K> startVertex = vertices.get(startVertexKey);  if (startVertex == null) {  throw new IllegalArgumentException("Start vertex not found.");  }  Stack<Vertex<T,K>> stack = new Stack<>();  stack.push(startVertex);  startVertex.setVisited(true);  while (!stack.isEmpty()) {  Vertex<T,K> currentVertex = stack.pop();  result.append("Visited: ").append(currentVertex.getId()).append("\n");  int currentVertexIndex = currentVertex.getIndex();  for (int i = 0; i < adjMatrix[currentVertexIndex].length; i++) {  if (adjMatrix[currentVertexIndex][i] != -1) {  Vertex<T,K> adjacentVertex = getVertexByIndex(i);  if (!adjacentVertex.isVisited()) {  stack.push(adjacentVertex);  adjacentVertex.setVisited(true);  }  }  }  }  resetVisitedState();  } catch (IllegalArgumentException e) {  result.append("Error: ").append(e.getMessage()).append("\n");  }  return result.toString();  } |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name and ID** | **R3:** Verify shortest path. | | |
| **Review** | *The system should check and save the shortest path leading from the start to the exit, using a minimum weight paths algorithm such as Djikstra or Floyd-Warshall.* | | |
| **Inputs** | **Input name** | **Data type** | **Valid values** |
| mazeGraph | Graph | *Representation of the labyrinth* |
| **Result** | The shortest path is stored in a linked list of vertexes | | |
| **Outputs** | **Output name** | **Data type** | **Format** |
| shortestPath | String | Linked list of vertices |

|  |
| --- |
| *Dijkstra for Adjacency List graph* |
| public String dijkstra(K startVertexId, K endVertexId) {  VertexList<T,K> startVertex = vertices.get(startVertexId);  VertexList<T,K> endVertex = vertices.get(endVertexId);  if (startVertex == null || endVertex == null) {  return "Error: One or both vertices not found.";  }  Map<K, Integer> distances = new HashMap<>();  Map<K, List<VertexList<T,K>>> shortestPaths = new HashMap<>();  for (VertexList<T,K> vertex : vertices.values()) {  distances.put(vertex.getId(), Integer.MAX\_VALUE);  shortestPaths.put(vertex.getId(), new ArrayList<>());  vertex.setVisited(false);  }  distances.put(startVertexId, 0);  shortestPaths.get(startVertexId).add(startVertex);  PriorityQueue<VertexList<T,K>> priorityQueue = new PriorityQueue<>(Comparator.comparingInt(distances:get));  priorityQueue.add(startVertex);  while (!priorityQueue.isEmpty()) {  VertexList<T,K> currentVertex = priorityQueue.poll();  if (!currentVertex.isVisited()) {  currentVertex.setVisited(true);  for (VertexList<T,K> neighbor : currentVertex.getAdjacencyList()) {  if (!neighbor.isVisited()) {  int newDistance = distances.get(currentVertex.getId()) + getWeightBetweenVertices(currentVertex, neighbor);  if (newDistance < distances.get(neighbor.getId())) {  distances.put(neighbor.getId(), newDistance);  List<VertexList<T,K>> currentPath = new ArrayList<>(shortestPaths.get(currentVertex.getId()));  currentPath.add(neighbor);  shortestPaths.put(neighbor.getId(), currentPath);  priorityQueue.add(neighbor);  }  }  }  }  }  StringBuilder result = new StringBuilder();  result.append("Shortest path from ").append(startVertexId).append(" to ").append(endVertexId).append(": ");  List<VertexList<T,K>> path = shortestPaths.get(endVertexId);  if (path.isEmpty() || !path.get(0).equals(startVertex)) {  result.append("No path found.");  } else {  for (VertexList<T,K> vertex : path) {  result.append(vertex.getId()).append(" -> ");  }  result.delete(result.length() - 4, result.length());  }  return result.toString();  } |

|  |
| --- |
| *Dijkstra for Adjacency Matrix graph* |
| public String dijkstra(K startVertexKey, K destinyKey) {  StringBuilder result = new StringBuilder();  try {  Vertex<T,K> startVertex = vertices.get(startVertexKey);  Vertex<T,K> destinyVertex = vertices.get(destinyKey);  if (startVertex == null || destinyVertex == null) {  throw new IllegalArgumentException("Start or destiny vertex not found.");  }  int numVertices = vertices.size();  int[] distances = new int[numVertices];  int[] previousVertices = new int[numVertices];  PriorityQueue<VertexDistancePair<T,K>> priorityQueue = new PriorityQueue<>();  for (int i = 0; i < numVertices; i++) {  distances[i] = Integer.MAX\_VALUE;  previousVertices[i] = -1;  }  int startVertexIndex = startVertex.getIndex();  distances[startVertexIndex] = 0;  priorityQueue.add(new VertexDistancePair<T,K>(startVertex, 0));  while (!priorityQueue.isEmpty()) {  VertexDistancePair<T,K> currentPair = priorityQueue.poll();  Vertex<T,K> currentVertex = currentPair.getVertex();  for (int i = 0; i < numVertices; i++) {  if (adjMatrix[currentVertex.getIndex()][i] != -1) {  int edgeWeight = adjMatrix[currentVertex.getIndex()][i];  int newDistance = distances[currentVertex.getIndex()] + edgeWeight;  if (newDistance < distances[i]) {  distances[i] = newDistance;  previousVertices[i] = currentVertex.getIndex();  priorityQueue.add(new VertexDistancePair<T,K>(getVertexByIndex(i), newDistance));  }  }  }  }  int destinyVertexIndex = destinyVertex.getIndex();  if (distances[destinyVertexIndex] == Integer.MAX\_VALUE) {  result.append("No path found.");  } else {  while (destinyVertexIndex != -1) {  result.insert(0, " -> " + getVertexByIndex(destinyVertexIndex).getId());  destinyVertexIndex = previousVertices[destinyVertexIndex];  }  result.delete(0, 4);  }  } catch (IllegalArgumentException e) {  result.append("Error: ").append(e.getMessage());  }  return result.toString();  } |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name and ID** | R4: Move character. | | |
| **Review** | *The system must move the character in the maze whenever the user presses a move key on any of the available squares.* | | |
| **Inputs** | **Input name** | **Data type** | **Valid values** |
| userClick | Event | User clic position |
| **Result** | The character is moved to the new position. | | |
| **Outputs** | **Output name** | **Data type** | **Format** |
| characterMovement |  | Update character's position in the maze |

|  |
| --- |
| *Movement Logic* |
| public void onKeyPressed(KeyEvent event){  switch (event.getCode()){  case RIGHT : {  state = State.RUN\_RIGHT;  rightPressed = true;  break;  }  case LEFT: {  state = State.RUN\_LEFT;  leftPressed = true;  break;  }  case UP:{  state = State.RUN\_UP;  upPressed = true;  break;  }  case DOWN:{  state = State.RUN\_DOWN;  downPressed = true;  break;  }  }  }  public void onKeyReleased(KeyEvent event){  switch (event.getCode()){  case RIGHT : {  state = State.IDLE;  rightPressed = false;  break;  }  case LEFT: {  state = State.IDLE;  leftPressed = false;  break;  }  case UP:{  state = State.IDLE;  upPressed = false;  break;  }  case DOWN:{  state = State.IDLE;  downPressed = false;  break;  }  }  } |

|  |
| --- |
| *Movement Painting* |
| public void paint(){  stop();  onMoveRight();  onMoveLeft();  onMoveUp();  onMoveDown();  switch(state){  case IDLE -> {graphicsContext.drawImage(idles.get(frame%2), position.getX(), position.getY());  frame++;  }  case RUN\_RIGHT -> {  graphicsContext.drawImage(runRight.get(frame%4), position.getX(), position.getY());  frame++;  }  case RUN\_LEFT -> {  graphicsContext.drawImage(runLeft.get(frame%4), position.getX(), position.getY());  frame++;  }  case RUN\_UP -> {  graphicsContext.drawImage(runUpper.get(frame%3), position.getX(), position.getY());  frame++;  }  case RUN\_DOWN -> {  graphicsContext.drawImage(runDown.get(frame%3), position.getX(), position.getY());  frame++;  }  }  } |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name and ID** | R5: Calculate and show score | | |
| **Review** | *The system must calculate the score of the player based on the number of steps taken and the treasures that the player picked up.* | | |
| **Inputs** | **Input name** | **Data type** | **Valid values** |
| numberOfSteps | int | Should be greater than 0 |
| numberOf Treasures | int | Should be greater than or equal to 0 |
| **Result** | The score is calculated and shown to the user | | |
| **Outputs** | **Output name** | **Data type** | **Format** |
| score | String | Message with the score |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name and ID** | R6: Treasures generation. | | |
| **Review** | *The system must generate treasures in random locations.* | | |
| **Inputs** | **Input name** | **Data type** | **Valid values** |
| treasureConfig | Configuration object | Define parameters for treasure generation |
| **Result** | Treasures are generated in random locations. | | |
| **Outputs** | **Output name** | **Data type** | **Format** |
| treasure | Treasure | An object of the Treasure class in a random location. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name and ID** | R7: Player inventory. | | |
| **Review** | *The system must allow the user to view his inventory, which stores the treasures he has collected.* | | |
| **Inputs** | **Input name** | **Data type** | **Valid values** |
| userClick | Event | - |
| **Result** | The inventory is shown. | | |
| **Outputs** | **Output name** | **Data type** | **Format** |
| inventory | - | A window showing a table with the name treasures. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name and ID** | R8: User interface (GUI). | | |
| **Review** | *The system must deploy a user interface that allows the development of the game. Once the user chooses to start the game, the system must change the window to the labyrinth one, allowing the user to interact directly with the labyrinth and to see his score once the game is finished.* | | |
| **Inputs** | **Input name** | **Data type** | **Valid values** |
|  |  | - |
| **Result** | A window with the main menu and the button to start the game is shown. | | |
| **Outputs** | **Output name** | **Data type** | **Format** |
| mainWindow |  | A window showing the main menu and the respective button to start the game. |

|  |
| --- |
| *Main Display* |
| public class MainMenuController{  @FXML  private Button playBt;  @FXML  private Button exitBt;  public void onPlay(){  GameApplication.openWindow("game-view.fxml");  Stage stage = (Stage) playBt.getParent().getScene().getWindow();  stage.close();  }  public void onExit(){  Stage stage = (Stage) exitBt.getParent().getScene().getWindow();  stage.close();  }  } |