**THE ENGINEERING DESIGN METHOD**

**PHASE 1: PROBLEM IDENTIFICATION**

Design a game in which players guide a frog through scenarios composed of water lilies and enemies. The goal is to provide an entertaining experience that balances strategy and action by including elements of risk, such as the loss of energy due to wrong choices and the possibility of losing lives when encountering enemies. The dynamics of the game, is defined by the interaction of the frog with the water lilies and the management of its resources, creating a continuous challenge for the players and contributing to the entertainment of the users.

**Need:**

The need focuses on providing users with a challenging and entertaining game experience through efficient resource management and strategic decision making. The game has a dynamic where players control the frog, managing its energy and lives while navigating through water lilies and avoiding enemies. The complexity of the game is seen by the use of algorithms such as BFS and Dijkstra, which define the connectivity of water lilies and calculate the distance between them. This creates an interactive environment where players must carefully select their destination, considering the consequences of loss of energy or lives.

**Problem definition:**

The goal of the system is to provide users with a form of entertainment through the frog game. The focus is on allowing players to guide the frog through water lilies, dodge enemies, and manage their energy resources and lives.

**TABLA DE ESPECIFICACIÓN DEL PROBLEMA**

| CUSTOMER |  |
| --- | --- |
| USER | Todo público |
| FUNCTIONAL REQUIREMENTS | **R1:** User must enter his name  **R2:** Allow the user to choose which type of network to use in the game.  **R3:** The system must correctly display the connections that a water lily has (vertex)  **R4:** The system must correctly calculate the energy cost from a source water lily to a chosen water lily and to an unchosen water lily.  **R5:** The system must correctly display the user's lives and energy and constantly update them in the game.  **R6:** The system must display a GameOver or Win sale in case of losing or winning the game. |
| CONTEXTO DEL PROBLEMA | Players embark on an exciting game to guide a frog through water lilies to a specific destination. Equipped with 10 energy bars and 3 lives, players explore an interconnected environment of water lilies represented by a graph, using algorithms such as BFS and Dijkstra. In this strategic challenge, route choice is crucial, as inefficient moves, calculated by Dijkstra, drain the frog's energy. Some water lilies have enemies, adding risk to the game and subtracting lives if the frog falls into them. The dynamics of the game are based on a combination of tactical decision making when choosing the destination and efficient energy management, keeping the excitement going until the frog reaches its goal or depletes all its lives. With algorithms such as DFS also influencing the generation of challenges, the frog game offers an immersive experience where strategy and overcoming obstacles are intertwined in a playful and challenging journey. |
| NON-FUNCTIONAL REQUIREMENTS | The game must have smooth and uninterrupted performance, with minimal load times.  The execution of algorithms such as BFS, Dijkstra and DFS must be efficient, ensuring fast responses to player actions.  The user interface should be attractive and easy to understand, with clear indicators of frog energy, remaining lives and movement options. Clear instructions should be provided on the use of algorithms and decision making in the game.  The game should be cross-platform compatible, including PC, mobile devices and consoles, allowing players to enjoy the game on their preferred platform. |

**Functional requirements analysis table**

| Name or identifier | User must enter his name | | |
| --- | --- | --- | --- |
| Summary | The user enters his nickname to start the game | | |
| Input | Nombre entrada | Tipo de dato | Condición de selección o repetición |
| Nickname | String | characters up to 10 digits |
| Result or postcondition |  | | |
| Outputs | Nombre entrada | Tipo de dato | Condición de selección o repetición |
|  |  |  |
|  |  |  |

| Name or identifier | Allow the user to choose which type of network to use in the game. | | |
| --- | --- | --- | --- |
| Summary | The player enters the type of network they want to start the game. | | |
| Input | Nombre entrada | Tipo de dato | Condición de selección o repetición |
| opt | String | Selection between:  -Graph, Dijkstra, BFS, Lista  -Graph, Dijkstra, DFS, Matriz |
| Result or postcondition | The user option is saved and the game is started | | |
| Outputs | Nombre entrada | Tipo de dato | Condición de selección o repetición |
|  |  |  |
|  |  |  |

| Name or identifier | **R2:** Allow the user to choose which type of network to use in the game. | | |
| --- | --- | --- | --- |
| Summary | El sistema utiliza una tabla hash para almacenar la información de las tareas y recordatorios. | | |
| Input | Nombre entrada | Tipo de dato | Condición de selección o repetición |
| Titulo | String |  |
| Descripción de la tarea | String |  |
| Fecha límite | Date |  |
| Prioridad | int | 1. Prioritaria 2. No prioritaria |
| Result or postcondition | La tarea se almacena correctamente en la tabla hash. | | |
| Outputs | Nombre entrada | Tipo de dato | Condición de selección o repetición |
| Mensaje emergente | String | Added successfully o An error occurred while adding |
|  |  |  |

| Name or identifier | **R3:** The system must correctly display the connections that a water lily has (vertex) | | |
| --- | --- | --- | --- |
| Summary | The system with the current vertex gets the vertex connections | | |
| Input | Nombre entrada | Tipo de dato | Condición de selección o repetición |
| Vertice Actual | String |  |
| Result or postcondition | From the current vertex, the connections of the same vertex are obtained | | |
| Outputs | Nombre entrada | Tipo de dato | Condición de selección o repetición |
| Conexiones Vertice | String |  |
|  |  |  |

| Name or identifier | **R4:** The system must correctly calculate the energy cost from a source water lily to a chosen water lily and to an unchosen water lily. | | |
| --- | --- | --- | --- |
| Summary |  | | |
| Input | Nombre entrada | Tipo de dato | Condición de selección o repetición |
| nenufar origen | String |  |
| nenufar destino | String |  |
| Result or postcondition |  | | |
| Outputs | Nombre entrada | Tipo de dato | Condición de selección o repetición |
| Mensaje emergente | String | Added successfully o An error occurred while adding |
|  |  |  |

| Name or identifier | **R5:** The system must correctly display the user's lives and energy and constantly update them in the game. | | |
| --- | --- | --- | --- |
| Summary | The system shows the amount of lives and energy of the frog. | | |
| Input | Nombre entrada | Tipo de dato | Condición de selección o repetición |
|  |  |  |
|  |  |  |
| Result or postcondition |  | | |
| Outputs | Nombre entrada | Tipo de dato | Condición de selección o repetición |
| number of lives | int |  |
| number of energy | int |  |

| Name or identifier | **R6:** The system must display a GameOver or Win sale in case of losing or winning the game. | | |
| --- | --- | --- | --- |
| Summary |  | | |
| Input | Nombre entrada | Tipo de dato | Condición de selección o repetición |
|  |  |  |
|  |  |  |
| Result or postcondition | Shows the game over or win in a window | | |
| Outputs | Nombre entrada | Tipo de dato | Condición de selección o repetición |
| Ventana | void |  |
|  |  |  |

**PHASE 2: GATHERING THE NECESSARY INFORMATION**

For this game we need to define different practices and approaches for the design, it should be taken into account the necessary structures for the realization of the different networks, this to know how each of these networks work, in this case it was used:

**Adjacency List:**

* It is used to represent a graph by storing, for each vertex, the list of vertices adjacent to it. It is space efficient for sparse graphs or graphs with few edges.

**Adjacency Matrix:**

* Uses a matrix to represent the connections between vertices. The entry (i, j) of the matrix indicates whether there is an edge between vertices i and j. It is efficient for dense networks, but may be less space efficient for sparse networks.

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

* A graph search algorithm that explores vertices in layers, visiting all neighbors of a vertex before moving on to neighbors of neighbors. It is used to find shortest paths.

**Dijkstra:**

* Algorithm used to find the shortest path between two vertices in a weighted network. It uses a priority queue to explore connections efficiently.

**DFS (Depth-First Search):**

* A graph search algorithm that explores as far as possible along each branch before backtracking. It can be used to find connected components or perform a deep traversal in the network.

**PHASE 3: FINDING CREATIVE SOLUTIONS**

For the choice and design of the game involving BFS, Dijkstra, DFS, adjacency matrix and adjacency list, several options were explored to effectively apply these algorithms. The key was to find a context where the stations could be represented as vertices and where the distance and connections between them were key elements of the game. The reasons for the choice of each component are detailed here:

**1. Station or Node Based Game:**

* A game involving a series of stations or nodes was selected, as these could efficiently represent the vertices in the network.
* Each station became a node in the network, and connections between stations were translated into connections in the network.

**2. Use of BFS and DFS:**

* The choice of a set with stations allowed BFS and DFS to be applied in a natural way to explore the connections between nodes.
* These algorithms were implemented to provide the player with information about possible routes and connections between stations.

**3. Dijkstra for Best Route:**

* The need to find the best route between two stations led to the choice of Dijkstra.
* Dijkstra was applied to calculate the shortest distance between stations and provide the player with the optimal route.

**4. Matrix and Adjacency List:**

* Both matrices and adjacency lists were used to represent the network.
* The adjacency matrix was useful for games where connections were dense, and the adjacency list for more sparse cases.

**5. Creativity in Game Interpretation:**

* Different games were explored that allowed for creative and relevant interpretation of the network structures.
* The choice of game was based on the possibility of incorporating BFS, Dijkstra and DFS concepts in a coherent way.

**6. Intuitive User Interface:**

* Consideration was given to implementing a graphical user interface that would allow players to easily and visually interact with the network and stations.

After taking all this into account, we thought of a game of a frog jumping from water lily to water lily, where the water lilies were the vertices and the user could choose water lilies until he reached the end, as he advanced he had the possibility of lowering energy or losing a life thanks to the fact that in the water lilies there could be different enemies.

**PHASE 4: TRANSITION FROM IDEA FORMULATION TO PRELIMINARY DESIGNS**

In this phase, the transition from the ideas formulated in the previous stage will be materialized through different analytical, simulation and physical modeling approaches. Each of these methods will bring a unique perspective to refine and validate the frog game design.

**Analytical Models:**

1. **Complexity Analysis 2.**

* Evaluate the algorithmic complexity of key game components, such as BFS, Dijkstra, and DFS.
* Determine the performance of the game to ensure a smooth and lag-free experience.

1. **Memory Efficiency Analysis:**

* Evaluate the memory usage of the data structures used to represent the network.
* Optimize memory management to ensure efficient performance.

**Simulation Models:**

1. **Player Behavior Simulation:**

* Create simulation models that mimic player behavior when selecting different routes and facing challenges.
* Evaluate how the game behaves in different scenarios and adjust gameplay accordingly.

1. **Game Scenario Simulation:**

* Implement simulations that represent various game scenarios, including critical situations like encountering enemies, choosing routes, etc.
* Adjust game difficulty based on the results of the simulations.

**Physical Models:**

1. **Graphic Interface Prototyping:**

* Develop prototypes of the graphical user interface (GUI) to visualize how players will interact with the game.
* Obtain feedback on usability and user experience.

1. **Game Mockups:**

* Build digital mockups of the game that represent the layout of water lilies, stations, and environmental elements.
* Evaluate the readability and comprehensibility of the game design.

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

As the design process evolves, the proposed solutions are subjected to rigorous evaluation. In this phase, the selection of the most appropriate solution to address the problem at hand is sought, considering a variety of factors. Evaluation criteria include accuracy, efficiency, comprehensiveness and adaptability. In addition, the choice of the best solution may be based on economic, social and environmental factors, as well as technical considerations.

Choice Type 1: Effectiveness and Efficiency:

Criterion A. Solution Effectiveness:

[2] Accurate and precise solution.

[1] Approximate solution.

Criterion B. Efficiency:

[4] Constant efficiency.

[3] Efficiency greater than constant.

[2] Logarithmic efficiency.

[1] Linear efficiency.

Choice Type 2: Comprehensiveness and Adaptability:

Criterion C. Comprehensiveness and Adaptability:

[3] Integral and highly adaptive solution.

[2] Several applications, but not integral.

[1] Single solution and not highly adaptable.

Choice Type 3: Implementation and Compatibility:

Criterion D. Ease of Implementation:

[2] Compatible with basic hardware operations.

[1] Not fully compatible with basic hardware operations.

|  | **Parameter A** | **Parameter B** | **Parameter C** | **Parameter D** | **Total** |
| --- | --- | --- | --- | --- | --- |
| Lista de Abyacencia | 2 | 3 | 2 | 2 | 9 |
| Matrix de Abyacencia | 2 | 1 | 2 | 1 | 6 |
| BFS | 1 | 4 | 2 | 2 | 9 |
| Dijkstra | 2 | 2 | 2 | 2 | 8 |
| DFS | 1 | 1 | 2 | 2 | 6 |

|  | **Parameter A** | **Parameter B** | **Parameter C** | **Parameter D** | **Total** |
| --- | --- | --- | --- | --- | --- |
| Modelos Analíticos | 2 | 2 | 2 | 2 | 8 |
| Modelos de simulación | 2 | 3 | 3 | 1 | 9 |
| Modelos físicos | 2 | 3 | 2 | 1 | 8 |

**PHASE 6: PREPARATION OF REPORTS AND SPECIFICATIONS**

For this phase of the system we must create the design documentation, including specifications of the data structures used, class diagrams and descriptions of the system's functionalities. Technical reports explaining the design and implementation of the system are also prepared (they will be found in the documents folder).

**PHASE 7: DESIGN IMPLEMENTATION**

Phase 7 tells us that the implementation of the system must be done according to the proposed design. Extensive testing should be done to ensure that the system works as intended.

\*Summary of chapter 5 of the book Introduction to Engineering. Paul H. Wright. 3rd ed. John Wiley & Sons, Inc. 2022.

**Temporal analysis:**

**First Algorithm**

Function adjacencyExisting(V v, V w):

// Get the vertex indices from the map.

indexV = vertexIndexMap.get(v) → O(1) = 1

indexW = vertexIndexMap.get(w) → O(1) = 1

// Check if the indexes are not null.

If (indexV is not null and indexW is not null) then: → O(1) = 1

// Check if there is an edge in the adjacency matrix.

If (adjacencyMatrix[indexV][indexW] is not null) then: → O(1) = 1

Return True

End If

End If

// If any of the indices is null or there is no edge, return False.

Return False

End Function

The function "adjacencyExisting" is efficient and has constant time complexity (O(1)), which means that the execution time does not depend on the size of the network and is fast regardless of the number of vertices or edges.

Second Algorithm:

Function dfsRecursiveWithNeighbors(int start, boolean[] visited, List<String> result):

result.add(vertices.get(start).toString()) → O(1).

visited[start] = true → O(1) -

For each neighbor i in the set of vertices: → O(n).

Edge<V, T> edge = adjacencyMatrix[start][i] → O(1).

If (edge is not null and !visited[i]) then: → O(1) - Condition check.

result.add(vertices.get(i).toString()) → O(1) - Add neighbor to list.

dfsRecursiveWithNeighbors(i, visited, result)

End Function

The overall time complexity of the function " dfsRecursiveWithNeighbors" is dominated by the loop that iterates over the neighbors, which makes it O(n), where n is the number of vertices in the network. The function performs a depth-first search in the network recursively, visiting each vertex once.

**Spatial Analysis**

**First Algorithm:**

Function adjacencyExisting(V v, V w):

// Get the vertex indices from the map.

indexV = vertexIndexMap.get(v) → O(1) = 1

indexW = vertexIndexMap.get(w) → O(1) = 1

// Check if the indexes are not null.

If (indexV is not null and indexW is not null) then: → O(1) = 1

// Check if there is an edge in the adjacency matrix.

If (adjacencyMatrix[indexV][indexW] is not null) then: → O(1) = 1

Return True

End If

End If

// If any of the indices is null or there is no edge, return False.

Return False

End Function

| Tipo | Variable | Cantidad de Valores Atómicos |
| --- | --- | --- |
| Entrada | v  w | 1  1 |
| Auxiliar | indexV  indexW | 1  1 |
| Salida | - | - |

Spatial Complexity Total = Input + Auxiliary + Output = 4 = O(1)

Spatial Complexity Auxiliary = 2 = O(1)

Spatial Complexity Auxiliary + Output = 2 = O(1)

**Second Algorithm:**

Function dfsRecursiveWithNeighbors(int start, boolean[] visited, List<String> result):

result.add(vertices.get(start).toString()) → O(1).

visited[start] = true → O(1) -

For each neighbor i in the set of vertices: → O(n).

Edge<V, T> edge = adjacencyMatrix[start][i] → O(1).

If (edge is not null and !visited[i]) then: → O(1) - Condition check.

result.add(vertices.get(i).toString()) → O(1) - Add neighbor to list.

dfsRecursiveWithNeighbors(i, visited, result)

End Function

| Tipo | Variable | Cantidad de Valores Atómicos |
| --- | --- | --- |
| Entrada | start  visited  result | 1  1  1 |
| Auxiliar | edge  i | 1  n |
| Salida | - | - |

Spatial Complexity Total = Input + Auxiliary + Output = 1 + 1 + 1 + 1 + n = O(n)

Spatial Complexity Auxiliary = 1 + n = O(n)

Spatial Complexity Auxiliary + Output = 1 + n = O(n)

**Diseño:**

| **Nombre** | Graph 1 |
| --- | --- |
| **Objeto abstracto** |  |
| **Invariante** |  |

| **Operación** | **Entradas** | **Salidas** | **Tipo de operación** | **Especificación** |
| --- | --- | --- | --- | --- |
| agregarVertice() | V vertice |  | Modifier | Adds a vertex to the graph. |
| agregarArista() | V v, V w, T peso |  | Modifier | Adds a weighted edge between vertices v and w with the given weight. |
| imprimirGrafo() |  |  | Analyzer | Prints the edges of the graph with their respective weights. |
| conectarAleatoriamente() | int numVertices, int maxAristas, y int maxPeso |  | Modifier | It randomly connects the vertices of the graph, generating edges with random weights. |
| obtenerVecinosBFS() | V inicio | List<V> | Analyzer | Performs a BFS traversal from the start vertex and returns the list of reached neighbors. |

| **Nombre** | Graph 2 |
| --- | --- |
| **Objeto abstracto** |  |
| **Invariante** |  |

| **Operación** | **Entradas** | **Salidas** | **Tipo de operación** | **Especificación** |
| --- | --- | --- | --- | --- |
| agregarVertic() | V vertice |  | Modifier | Adds a vertex to the graph if it does not exist, updating the adjacency matrix. |
| agregarArista() | V v, V w, T peso |  | Modifier | Adds a weighted edge between vertices v and w with the given weight, updating the adjacency matrix. |
| imprimirGrafo() |  |  | Analyzer | Imprime las aristas del grafo con sus respectivos pesos. |
| conectarAleatoriamente() | int numVertices, int maxAristas, int maxPeso |  | Modifier | It randomly connects the vertices of the graph, generating edges with random weights and updating the adjacency matrix. |
| adyacenciaExistente() | V v, V w | Boolean | Analyzer | Checks if there is an edge between vertices v and w in the adjacency matrix. |
| dfsConVecinos() | V inicio | List<String> | Analyzer | Performs a DFS traversal from the start vertex and returns the list of vertices reached. |
| dijkstra() | V inicio, V destino | int | Analyzer | Apply Dijkstra's algorithm to calculate the shortest distance between start and destination. Returns -1 if there is no path. |
| actualizarMatriz() |  |  | Modifier | Updates the adjacency matrix when adding a new vertex to the graph. |