**THE ENGINEERING DESIGN METHOD**

**Context of the problem**

The context of this problem is framed in the field of software development and game design. The goal is to create an interactive game based on a graph, a mathematical data structure consisting of vertices and edges. In this case, a graph with a minimum of 50 vertices and 50 edges is required, which implies a play structure of some complexity. This graph should be the backbone of the game and allow the application of at least two graph algorithms, which can be selected from a specific set, such as Graph Paths (BFS, DFS), Minimum Weight Paths (Dijkstra, Floyd-Warshall) and Minimum Covering Tree (Prim, Kruskal). each of which must follow a complete process.

In this context, the game in development is a three-dimensional experience set in space, where the player takes on the role of a space pilot in search of a distant planet. To reach this destination, the user must navigate through a complex network of paths and routes represented by the graph mentioned above. The graph acts as the interconnected network of star systems, with each vertex representing a key point in space, such as space stations or asteroids. Edges, on the other hand, symbolize the navigable routes between these points.

The main objective of the game is to collect spaceships scattered at different points in the graph to strengthen the player's fleet and increase their chances of success. However, the journey will not be easy, as the space is populated by hostile enemies that the player must dodge or face strategically. The complexity of the graph and its application in the game adds an additional level of challenge, as the player must carefully plan their movements to optimize ship gathering and avoid unnecessary confrontations

**PHASE 1: PROBLEM IDENTIFICATION**

Problem: The problem centers on the development of a two-dimensional interactive game set in space, where the player must reach a planet by collecting scattered spaceships.

Needs:

* Space Environment Design
* Spacecraft Modeling
* Harvesting Mechanics
* Objective of the Game
* Intuitive User Interface
* Visual and Sound Effects
* Adaptability of the Difficulty Level
* Testing & Feedback
* Game Narrative

**SOFTWARE ENGINEERING PROBLEM SPECIFICATION TABLE**

|  |  |
| --- | --- |
| CUSTOMER |  |
| USER |  |
| FUNCTIONAL REQUIREMENTS | 1. Space Exploration 2. Spacecraft Collection 3. Starting Location 4. Avatar Mobility 5. Space Combat 6. Strategic Navigation |
| CONTEXT OF THE PROBLEM |  |
| NON-FUNCTIONAL REQUIREMENTS | 1. Yield 2. Compatibility 3. Visual Aesthetics 4. Ambient Sound 5. Intuitive Controls: 6. Safety 7. Stability & Performance |

| Name or identifier | 1. **Space Exploration** | | |
| --- | --- | --- | --- |
| Summary | The game should allow the player to explore a three-dimensional space environment, interacting with various elements such as planets, asteroids, and space stations. | | |
| Tickets | Ticket name | Type of data | Selection or repetition condition |
| Controls | Char |  |
| Outcome or post-condition | The player is immersed in a visually detailed space environment and can explore freely, observing planets, asteroids, and space stations in real-time. | | |

| Name or identifier | 1. **Spacecraft Collection** |
| --- | --- |
| Summary | The game should provide the player with the task of collecting spaceships scattered in the environment to strengthen their fleet |
| Outcome or post-condition | Upon collection, the ship is added to the player's inventory, upgrading their fleet |

| Name or identifier | 1. **Starting Location** |
| --- | --- |
| Summary | The Game must establish a starting location for the player within the spatial environment at the start of the game. |
| Outcome or post-condition | The player's avatar is positioned at the designated starting location, ready to begin exploration. The initial narrative context has been established |

| Name or identifier | 1. **Avatar Mobility** |
| --- | --- |
| Summary | The game should allow the player to move fluidly and control their avatar in three-dimensional space. |
| Outcome or post-condition | The avatar moves smoothly in the direction specified by the player, with speed and direction adjustments based on the player's preference. |

| Name or identifier | 1. **Space Combat** |
| --- | --- |
| Summary | The game should introduce a combat system that allows the player to face hostile enemies during space exploration. |
| Outcome or post-condition | After performing combat actions, the visual and auditory state of the game reflects the outcome of the match. It can include destroying enemies or changing the health of ships |

| Name or identifier | 1. **Strategic Navigation** |
| --- | --- |
| Summary | The game should facilitate the player's ability to plan strategic routes through the graph, taking into account the location of spaceships, enemies, and obstacles. |
| Outcome or post-condition | The player has selected a strategic path through the graph, and the game provides clear visualizations of the path and possibly adjusts the player's position accordingly. |

**PHASE 2: GATHERING THE NECESSARY INFORMATION**

**Adjacency matrix The adjacency matrix is a** square matrix that is used as a way to represent binary relationships.

1. For each edge that joins two nodes, [1 is added](https://es.wikipedia.org/wiki/Uno)  to the value that is currently at the corresponding location in the array.
   * If such an edge is a [loop](https://es.wikipedia.org/wiki/Bucle_(teor%C3%ADa_de_grafos)) and the graph  [is undirected](https://es.wikipedia.org/wiki/Grafo#Grafo_no_dirigido), then 1 or 2 is added (depending on the convention used).
   * If the graph is [weighted](https://es.wikipedia.org/wiki/Grafo_ponderado), then instead of a 1 the weight of the respective edge is added.

Finally, a matrix is obtained that represents the number of edges (relationships) between each pair of nodes (elements).

There is a unique adjacency matrix for each graph (without considering [permutations](https://es.wikipedia.org/wiki/Permutaci%C3%B3n) of rows or columns), and vice versa.

* For an undirected graph, the adjacency matrix is symmetric.
* The number of paths *Ci,j*(*k), traversing*  k edges from node *i* to node *j*, is given by an element of the k-th power of the adjacency matrix:

**Graph** A graph is a set of objects called vertices or nodes joined by links called edges or arcs, which allow binary relationships between elements of a set to be represented

**Adjacendy list** An adjacency list is a representation of all the [edges](https://es.wikipedia.org/wiki/Arista_(teor%C3%ADa_de_grafos)) or arcs of a [graph](https://es.wikipedia.org/wiki/Grafo) using a [list](https://es.wikipedia.org/wiki/Lista_(inform%C3%A1tica)). If the graph is undirected, each input is a [set](https://es.wikipedia.org/wiki/Conjunto) or [multiset](https://es.wikipedia.org/wiki/Multiconjunto) of two [vertices](https://es.wikipedia.org/wiki/V%C3%A9rtice_(teor%C3%ADa_de_grafos)) containing the two ends of the corresponding edge. If the graph is directed, each input is a [tuple](https://es.wikipedia.org/wiki/Tupla) of two nodes, one denoting the source node and the other denoting the destination node of the corresponding arc. Typically, adjacent lists are not sorted

**BFS**  is a search algorithm that traverses the nodes of a graph, starting at the root (choosing some node as a root element in the case of a graph), and then explores all the neighbors of this node. Intuitively, you start at the root (choosing some node as the root element in the case of a graph) and explore all the neighbors of this node. Then for each of the neighbors their respective adjacent neighbors are explored, and so on until the entire tree is traversed.

**DFS** is an uninformed search [algorithm](https://es.wikipedia.org/wiki/B%C3%BAsquedas_no_informadas) used to traverse all nodes in a graph  [or](https://es.wikipedia.org/wiki/Grafo) tree (graph theory) [in an](https://es.wikipedia.org/wiki/%C3%81rbol_(teor%C3%ADa_de_grafos))  orderly, but not uniform, manner. Its operation consists of expanding each and every one of the nodes that it locates, on a recurring basis, on a specific path. When there are no more nodes left to visit on that path, it returns ([Backtracking](https://es.wikipedia.org/wiki/Backtracking)), so it repeats the same process with each of the siblings of the already processed node.

**Dijkstra algorithm** is an [algorithm](https://es.wikipedia.org/wiki/Algoritmo) for determining the [shortest path,](https://es.wikipedia.org/wiki/Problema_del_camino_m%C3%A1s_corto) given an origin [vertex](https://es.wikipedia.org/wiki/V%C3%A9rtice_(teor%C3%ADa_de_grafos)), to the rest of the vertices in a [graph](https://es.wikipedia.org/wiki/Grafo) that has weights on each [edge](https://es.wikipedia.org/wiki/Arista_(teor%C3%ADa_de_grafos)). The idea behind this algorithm is to explore all the shortest paths that start from the source vertex and lead to all the other vertices; When the shortest path is obtained from the origin vertex to the rest of the vertices that make up the graph, the algorithm stops. This is a specialization of uniform-cost search  [and, as such, does not work on graphs with negative-cost edges (by always choosing the node with the shortest distance, nodes that in future iterations would lower the overall cost of the path by passing through an edge-negative-cost can be excluded from the search).](https://es.wikipedia.org/wiki/B%C3%BAsqueda_de_costo_uniforme)

**Floyd-Warshall algorithm**

1. **Warshall' s algorithm:** Warshall's algorithm is an example of a Boolean algorithm. From an initial table composed of 0's (there is no initial correspondence in the graph) and 1's (there is a correspondence, called an "arrow", between nodes), he obtains a new matrix called "Transitive Closure Matrix" in which all the possible unions between nodes are shown, directly or indirectly. That is, if there is no "arrow" from "A" to "B", it is possible that there is from "A" to "C" and then from "C" to "B". This result will then be dumped into the final matrix.
2. **Floyd's algorithm: Floyd's algorithm** is very similar, but it works with weighted graphs. That is, the value of the "arrow" we represent in the matrix can be any real or infinite number. Infinity indicates that there is no union between the nodes. This time, the result will be a matrix where the minimum distances between nodes will be represented, selecting the most convenient paths according to their weighting ("weight").

The **Floyd-Warshall** algorithm compares all possible paths through the graph between each pair of vertices. The algorithm is able to do this with only V³ comparisons (this is remarkable considering that there can be up to 2 edges in the graph, and that every combination of edges is tested). It does this by gradually improving an estimate of the shortest path between two vertices, until it is known that the estimate is optimal.

**Prim algorithm Prim's algorithm** allows you to find a minimum overlain tree of a graph. In other words, the algorithm finds a subset of edges that form a tree with all vertices, where the total value of all edges in the tree is the minimum possible.

**Kruskal's algorithm is a** process that allows all the nodes of a graph to be joined to form a tree, taking into account the weight of the edges and whose total cost is as low as possible

**PHASE 3: FINDING CREATIVE SOLUTIONS**

In this problem, you can set solutions such as:

Brainstorming:

• Create a visually stunning space environment with realistic details of planets, space stations, and asteroids.

• Implement visual and sound effects that bring space to life, such as nebulae, shooting stars, and authentic space sounds.

• Develop a graph generator that creates complex and challenging lattices for spatial navigation.

• Ensure that each vertex of the graph uniquely represents a point of interest in space.

• Allow the player to interact with spaceships through intuitive controls for fleet gathering and upgrading.

• Design animations and visual effects to highlight successful spacecraft harvesting.

• Implement an artificial intelligence system for enemies that dynamically reacts to the player's actions.

• Ensure varied tactics from enemies, such as chases, ambushes, and evasions.

• Develop a user interface that allows the player to strategically plan their route through the graph.

• Provide visual information on the location of spacecraft, enemies, and obstacles to make informed decisions.

• Implement a system that allows the player to upgrade and customize their spaceships.

• Include cosmetic customization options and functional enhancements that affect exploration and combat performance.

• Create a scoring system that rewards not only ship collection, but also navigation efficiency and combat skill.

• Ensure that scores reflect the complexity of the actions taken by the player.

• Design specific missions and objectives that guide the player through the space environment.

• Provide meaningful rewards for successful completion of missions.

• Implement visual messages and alerts that inform the player about key events, such as the detection of nearby enemies or the availability of harvestable ships.

• Ensure that feedback is clear and non-intrusive so as not to interrupt the immersion in the game.

• Establish a constant process of unit testing and integration for each component of the game.

• Collect player feedback and make iterative updates to improve gameplay and fix potential issues.

**PHASE 4: TRANSITIONING IDEAS TO PRELIMINARY DESIGNS**

In this phase, the preliminary designs of the Game will be carried out, based on the creative solutions identified in Phase 3:

* **Visual Design of the Spatial Environment:**
  + Use graphic design tools to create visual sketches of the spatial environment. Define key colors, textures, and visuals.
* **Interaction with Spacecraft:**
  + Design the interface for interaction with spaceships, keeping in mind the collection and upgrading of the fleet. Create UI sketches for these actions.
* **Strategic Planning in the Graph:**
  + Create visual outlines for strategic planning. Define how the graph will be displayed and what information will be visible to the player.
* **Artificial Intelligence for Enemies:**
  + Define key enemy AI tactics and behaviors. Describe how they will interact with the player and each other.
* **Ship Upgrade and Customization:**
  + Specifies ship customization and upgrade options. Create sketches or diagrams to show how players will interact with these features.
* **Dynamic Scoring System:**
  + Defines how the player's score will be calculated in real-time. Specifies the events that will contribute to the score.
* **Development of Missions and Objectives:**
  + Design missions and objectives, establishing the criteria for their fulfillment and the associated rewards. Define the interface to display these missions.
* **Initial Prototyping:**
  + Create initial prototypes using game development tools. These prototypes should provide a basic functional representation of the gameplay mechanics.
* **Internal Testing:**
  + Conduct internal testing to evaluate gameplay, identify potential issues, and adjust the interface and mechanics as needed.
* **Iterative Settings:**
  + Based on feedback from internal testing, make iterative adjustments to the mechanics and interface to improve the user experience.

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

**Definition of evaluation criteria**

To evaluate and select the best solution, we will define criteria based on the needs and requirements identified in the previous phases:

**Criterion A. User Experience (Gameplay):**

* **[4] Exceptional Gameplay:**
  + The solution offers a highly immersive and satisfying user experience, with intuitive controls and a friendly learning curve.
* **[3] Better Than Satisfying Gameplay:**
  + The user experience is good, although there may be some areas for improvement in terms of controls or game fluidity.
* **[2] Acceptable Gameplay:**
  + The solution meets the minimum gameplay requirements, but might be less engaging or more difficult to understand.
* **[1] Limited Gameplay:**
  + The user experience is unsatisfactory, with significant issues in gameplay.

**Criterion B. Complexity of the Graph and Strategic Navigation:**

* **[4] Optimal Complexity:**
  + The graph offers adequate complexity for strategic navigation, challenging the player without being overwhelming.
* **[3] Moderate complexity:**
  + The graph provides a good balance between challenge and accessibility, although there may be areas where complexity can be adjusted.
* **[2] Minor Complexity:**
  + The complexity of the graph is lower than expected, which can negatively impact the strategic experience.
* **[1] Excessive complexity:**
  + The complexity of the graph significantly hinders strategic navigation, negatively affecting gameplay.

**Criterion C. Variety and Challenge of Space Combat:**

* **[4] Optimal Variety and Challenge:**
  + The combat system offers a wide variety of tactics and challenges, keeping the player engaged and entertained.
* **[3] Suitable Variety and Challenge:**
  + Combat is satisfying, but there may be areas where more variety or challenge can be added.
* **[2] Limited Variety and Challenge:**
  + The combat system may lack variety or present insufficient challenges to keep the player engaged.
* **[1] Unsatisfactory Variety and Challenge:**
  + The lack of variety and challenge in combat negatively affects the user experience.

**Criterion D. Systems Integration (Ships, Upgrades, Enemies):**

* **[4] Optimal Integration:**
  + Ship, upgrade, and enemy systems are integrated consistently, offering a harmonious and complete gameplay experience.
* **[3] Successful Integration:**
  + Systems are properly integrated, although there may be areas where transitions or interactions between them can be improved.
* **[2] Basic Integration:**
  + System integration is basic, with potential difficulties in game coherence.
* **[1] Unsatisfactory integration:**
  + Significant issues in system integration negatively affect the game experience

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Criterion A** | **Criterion B** | **Criterion C** | **Criterion D** | **Total** |
| **Visual Design of the Spatial Environment** | 4 | 3 | 4 | 2 | 13 |
| **Interaction with Spacecraft** | 3 | 3 | 4 | 2 | 12 |
| **Strategic Planning in the Graph** | 4 | 4 | 4 | 2 | 14 |
| **Artificial Intelligence for Enemies** | 4 | 4 | 4 | 2 | 14 |
| **Ship Upgrade & Customization** | 3 | 2 | 4 | 2 | 11 |
| **Dynamic Scoring System** | 4 | 3 | 4 | 2 | 13 |
| **Development of Missions and Objectives** | 3 | 3 | 3 | 2 | 11 |
| **Initial Prototyping** | 2 | 2 | 2 | 2 | 8 |
| **Internal Testing** | 3 | 3 | 3 | 2 | 11 |
| **Iterative Tweaks** | 3 | 3 | 4 | 2 | 12 |

All the proposed solutions seem feasible according to the defined criteria. This suggests that each solution has its merits and can be considered based on the specific needs and individual priorities of the game.

**PHASE 6. REPORTING & SPECIFICATIONS**

**Problem: Develop an interactive space game focused on exploring and collecting ships in a three-dimensional environment.**

PHASE 6. REPORTING & SPECIFICATIONS

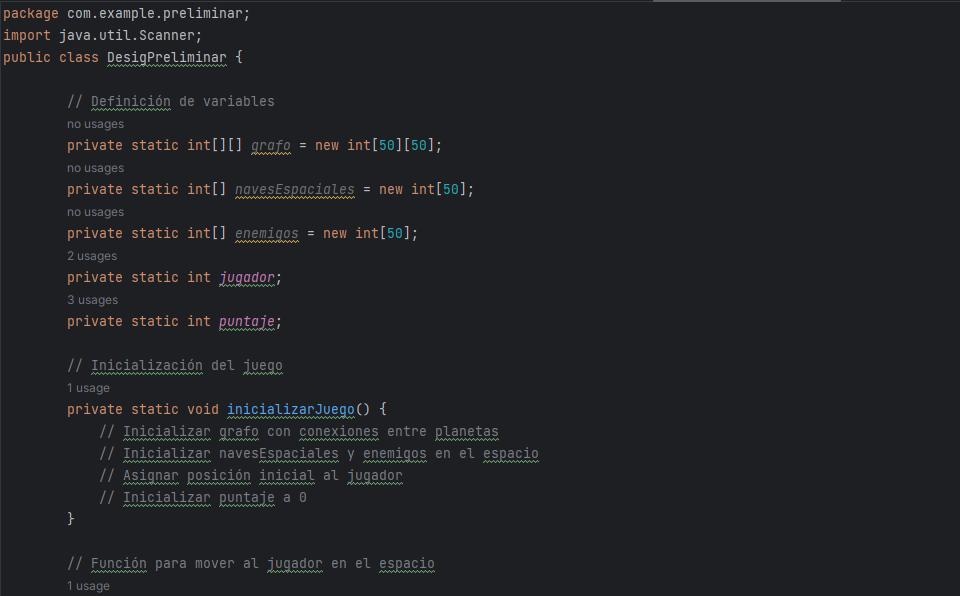
**Tickets:**

1. **Game Settings:**
   * Initial parameters, such as space size, number and types of ships, and initial layout.
2. **User Commands:**
   * Instructions for the player, including text commands to add, upgrade, or customize ships, as well as navigate space and engage enemies.
3. **Strategic Consultations:**
   * Commands to query strategic information, such as graph visualization, mission details, and ship inventory status.

**Outputs:**

1. **Graphical User Interface (GUI):**
   * Visual representation of the space environment, including planets, stars, player ships, and enemies.
2. **List of Ships and Upgrades:**
   * A screen that lists the available ships, their stats, and possible upgrades.
3. **Specific Ship Details:**
   * Detailed information about a specific ship, including its type, capabilities, and applied upgrades.
4. **Data Persistence:**
   * Automatic saving of player progress, including ships collected, upgrades applied, and quests completed.
5. **System Messages:**
   * Informational messages for the player, such as confirmation of actions taken, event alerts, and error messages in case of incorrect entries.
6. **Undo/Redo System:**
   * Ability to undo and redo actions, providing a more flexible user experience.

**Preliminary design:**



Text

Auto-generated description

Text

Auto-generated description

Text

Auto-generated description

|  |
| --- |
| TAD <Grafo> |
| Graph = {nodes, edges, directed} |
| Primitive Operations:  - CreateGraph(): -> Graph  - AddNode(node): -> void  - Add Edge(Origin, Destination, Weight): -> void  - Get Nodes(): -> nodes  - Get Edges(): -> edges  - IsTargeted(): -> targeted  - ExistsArista (origin, destination): -> bool  - GetNeighbors(node): -> neighbors |

|  |
| --- |
| CreateGraph()  "Create a new empty graph."  {Pre: None}  {Pos: Created Graph} |

|  |
| --- |
| AddNode(Node)  "Add a node to the graph."  {Pre: node}  {Pos: Node added to graph} |

|  |
| --- |
| Add Edge(Origin, Destination, Weight)  "Add an edge between the source and destination nodes with the given weight."  {Pre: Origin, Destination, Weight}  {Pos: Edge added to graph} |

|  |
| --- |
| GetNodes()  "Returns the list of nodes in the graph."  {Pre: None}  {Pos: Node List} |

|  |
| --- |
| GetEdges()  "Returns the list of edges in the graph."  {Pre: None}  {Pos: List of edges} |

|  |
| --- |
| IsDirected()  "Indicates whether the graph is directed or not."  {Pre: None}  {Pos: Boolean value |

|  |
| --- |
| There isArista (origin, destination)  "Check if there is an edge between the source and destination nodes."  {Pre: Origin, Destination}  {Pos: Boolean value} |

|  |
| --- |
| GetNeighbors(node)  "Returns the list of neighboring nodes to the given node."  {Pre: node}  {Pos: Neighbor Node List} |

|  |
| --- |
| TAD <Adjacency Matrix> |
| MatrixAdjacency = {nodes, edges, array} |
| Inv: matrix[i][j] = 1 if there is an edge between node i and node j, 0 otherwise |
| Primitive Operations:  - Create MatrixAdjacency(n): -> MatrixAdjacency  - Add Edge(source, destination): -> void  - Get Edges(): -> edges  - GetMatrix(): -> matrix |

**Max Adjacency Matrix**

|  |
| --- |
| CreateMatrixAdjacency(n)  "Create an adjacency array with n nodes."  {Pre: n}  {Pos: Adjacency Matrix Created} |

|  |
| --- |
| AddEdge(source, destination)  "Add an edge between the source and destination nodes in the array#.  {Pre: Origin, Destination}  {Pos: Updated matrix with edge} |

|  |
| --- |
| GetEdges()  "Returns the total number of edges in the graph."  {Pre: None}  {Pos: Total number of edges} |

|  |
| --- |
| GetMatrix()  "The adjacency matrix returns."  {Pre: None}  {Pos: Adjacency Matrix} |

|  |
| --- |
| TAD <Adjacency List> |
| Adjacency list = {nodes, list} |
| Inv: list[i] contains the nodes adjacent to node i |
| Primitive Operations:  - CreateAdjacencyList(n): -> Adjacency List  - Add Edge(source, destination): -> void  - GetNeighbors(node): -> list |

**Max Adjacency List**

|  |
| --- |
| CreateAdjacencyList  "Create an adjacency list with n nodes."  {Pre: n}  {Pos: Adjacency List Created} |

|  |
| --- |
| AddEdge(source, destination)  "Adds a ridge between the source and destination nodes in the list."  {Pre: Origin, Destination}  {Pos: Updated list with edge} |

|  |
| --- |
| GetNeighbors(node)  "Returns the list of nodes adjacent to the given node."  {Pre: node}  {Pos: Adjacent Node List} |

|  |
| --- |
| TAD <BFS> |
| BFS = {graph, visited, queue} |
| Primitive Operations:  - StartBFS(graph): -> BFS  - PerformBFS(start): -> void  - GetRoute(): -> route |

**Max BFS**

|  |
| --- |
| StartBFS(graph)  "Initialize the BFS algorithm with the given graph."  {Pre: graph}  {Pos: Initialized BFS Algorithm} |

|  |
| --- |
| PerformBFS(Start)  "Perform the BFS journey starting from the home node."  {Pre: start}  {Pos: BFS Tour Taken} |

|  |
| --- |
| GetTour()  "Return the BFS tour made."  {Pre: None}  {Pos: BFS Tour} |

|  |
| --- |
| TAD <DFS> |
| DFS = {graph, visited} |
| Primitive Operations:  - StartDFS(graph): -> DFS  - PerformDFS(start): -> void  - GetRoute(): -> route |

**Max DFS**

|  |
| --- |
| StartDFS(graph)  "Initialize the DFS algorithm with the given graph."  {Pre: graph}  {Pos: Initialized DFS Algorithm} |

|  |
| --- |
| Perform DFS(Start)  "Perform the DFS traversal starting from the home node."  {Pre: start}  {Pos: DFS Tour Taken} |

|  |
| --- |
| GetTour()  "Return the DFS route taken."  {Pre: None}  {Pos: DFS Tour} |

|  |
| --- |
| TAD <Dijkstra> |
| Dijkstra = {graph, distances, visited} |
| - InitiateDijkstra(graph): -> Dijkstra  - CalculateDistances(origin): -> void  - GetPath(destination): -> path  - GetDistance(destination): -> distance |

**Max Dijkstra**

|  |
| --- |
| InitiateDijkstra(graph)  "Initialize the Dijkstra algorithm with the given graph."  {Pre: graph}  {Pos: Initialized Dijkstra Algorithm} |

|  |
| --- |
| CalculateDistances (Origin)  "Calculates the shortest distances from the source node."  {Pre: origin}  {Pos: Calculated Distances} |

|  |
| --- |
| GetWay(destination)  "Return the shortest path to the destination node."  {Pre: destination}  {Pos: Shortest Way} |

|  |
| --- |
| GetDistance (destination)  "Returns the shortest distance to the destination node."  {Pre: destination}  {Pos: Shorter Distance} |

|  |
| --- |
| CAS <Floyd-Warshall> |
| FloydWarshall = {graph, distances} |
| Primitive Operations:  - StartFloydWarshall(graph): -> FloydWarshall  - CalculateDistances(): -> void  - GetDistance (origin, destination): -> distance |

|  |
| --- |
| StartFloydWarshall(graph)  "Initialize the Floyd-Warshall algorithm with the given graph."  {Pre: graph}  {Pos: Floyd-Warshall algorithm initialized} |

|  |
| --- |
| CalculateDistances()  "Calculate all the distances between each pair of nodes."  {Pre: None}  {Pos: Calculated Distances} |

|  |
| --- |
| GetDistance (Origin, Destination)  "Returns the distance between the source node and the destination node."  {Pre: Origin, Destination}  {Pos: Distance Between Nodes} |

# References

Singer, I. (n.d.). *Polytechnic School*. Retrieved from https://www.cartagena99.com/recursos/alumnos/apuntes/prog2-tema2-Pilas-1617.pdf

Ingeneriaudb. (n.d.). Retrieved from https://www.udb.edu.sv/udb\_files/recursos\_guias/informatica-ingenieria/programacion-con-estructuras-de-datos/2020/i/guia-8.pdf

keepcoding. (n.d.). *keepcoding.* Retrieved from https://keepcoding.io/blog/tecnicas-lifo-y-fifo-gestion-de-colas/

Lenis, A. (n.d.). Retrieved from https://blog.hubspot.es/website/interfaz-usuario#que-es

OpenAccess. (n.d.). Retrieved from https://openaccess.uoc.edu/bitstream/10609/9122/1/Intro\_TADs.pdf

pcsummary. (n.d.). *pcsummary*. Retrieved from https://www.pcresumen.com/menu-software/35-lenguajes-de-programacion/java/98-la-clase-priorityqueue-y-la-interfaz-queue#:~:text=PriorityQueue%20es%20una%20de%20las,suministra%20a%20trav%C3%A9s%20del%20constructor

sedici.unlp.edu.ar. (n.d.). Retrieved from http://sedici.unlp.edu.ar/bitstream/handle/10915/33386/Documento\_completo.pdf?sequence=3