**ENGINEERING METHOD**

**Problematic context**

It should be possible to develop a game for one or more players, and it must be modeled using graphs. The game should have a minimum of 50 vertices and 50 edges. Additionally, it is required to use at least two of the algorithms studied in the course, such as BFS, DFS, Dijkstra, Floyd-Warshall, Prim, or Kruskal.

**Step 1: Problem Identification**

Identification of needs and symptoms:

* We need to develop a game for one or more players that is modeled using graphs.
* We must have a minimum of 50 vertices and 50 edges.
* At least two graph algorithms need to be applied.

*(Requirements specification is in another document called "requirements specification")*

**Step 2: Information Gathering**

Search for definitions of terms to be implemented in the problem:

*Graphs*

In mathematics and computer science, a graph is a set of objects called vertices or nodes connected by links called edges or arcs, which allow the representation of binary relationships between elements of a set. They are the subject of study in graph theory.

*BFS ‘Breadth First Search’*

Breadth-First Search (BFS) is a search algorithm that traverses the nodes of a graph. It starts at the root by selecting a node as the root element (in the case of a graph) and then explores all the neighbors of this node. Subsequently, for each of the neighbors, their respective adjacent neighbors are explored, and so on until the entire graph is traversed. It is worth noting that if the node is found before traversing all nodes, the search concludes.

Breadth-First Search is used for algorithms where choosing the best possible path at each moment of the traversal is critical.

*DFS ‘Depth First Search’*

Depth-First Search (DFS) is a search algorithm that traverses the nodes of a graph. Its operation involves expanding each of the located nodes recursively (from the parent node to the child node). When there are no more nodes to visit on that path, it returns to the predecessor node, repeating the same process with each neighbor of the node. It is important to note that if the node is found before traversing all nodes, the search concludes.

Depth-First Search is used when we want to test if one solution among several possible ones meets certain requirements. This is seen, for example, in the problem of finding the path a knight must take on a chessboard to visit all 64 squares.

*Dijkstra*

The Dijkstra's algorithm, also known as the shortest path algorithm, is a method for determining the shortest path from a source vertex to all other vertices in a graph with weights on each edge. The underlying idea of this algorithm is to explore all the shortest paths originating from the source vertex and leading to all other vertices. When the shortest path from the source vertex to all other vertices in the graph is obtained, the algorithm stops. This algorithm is a specialization of uniform-cost search, and as such, it does not work in graphs with edges of negative cost (by always choosing the node with the smallest distance, nodes that would lower the overall cost of the path in subsequent iterations by traversing an edge with negative cost may be excluded from the search).

*Floyd-Warshall*

The Floyd-Warshall algorithm is the choice when determining the shortest path between all pairs of vertices in a graph. By comparing all possible paths, it gradually improves the estimation until reaching the most optimal one. This can be illustrated more clearly through an example of implementation. But before that, let's review the runtime analysis for this case.

*Árbol de Recubrimiento Mínimo -MST*

A Minimum Spanning Tree (MST), or Minimum Spanning Forest, is a spanning tree of a connected, undirected graph that has the minimum possible total edge weight. In other words, it is a tree that spans all the vertices in the graph, with the sum of edge weights minimized. Every graph has at least one minimum spanning tree.

*Prim*

The Prim's algorithm finds a minimum-weight tree by connecting nodes or vertices with minimum-weight edges from the graph without forming cycles. It involves dividing the nodes of a graph into two sets: processed and unprocessed. Initially, there is one node in the processed set corresponding to the central node. In each iteration, the algorithm adds a node (connected by the minimum-weight edge) to the processed set until all nodes in the graph are connected.

*Kruskal*

The Kruskal's algorithm aims to obtain a tree in which the sum of its edge weights is minimized, earning the name of a greedy algorithm. The algorithm is based on a property of trees that allows us to determine whether an edge should belong to the tree or not, and it leverages this property to select each edge.

It's crucial to note that whenever an edge is added, it will always be the shortest (lowest cost) connection from the starting node to the rest of the graph. Therefore, by definition, it should be part of the tree. This algorithm is of a greedy type because at each step, it selects the edge with the smallest value and adds it to the subgraph.

It should be taken into account that there may be more than one solution with the same minimum value, especially when the edge values are repeated.

**Step 3: Creative Solutions Search**

Here's the translation of your plan into English:

In this case, we are planning, as a rough sketch, to implement:

1. **Optimal Flow Algorithm:**
   * Objective: Determine the most efficient solution between two points, considering factors such as the construction cost of pipes and "flow" capacity.
2. **Change Propagation Algorithm:**
   * Objective: Simulate how changes in one part of the graph affect other parts, allowing for better and more dynamic adjustments to pipe connections.
3. **Dynamic Network Optimization Algorithm:**
   * Objective: Adjust the pipe network based on current demands and game conditions dynamically.
4. **Pipe Flow Exploration Algorithm:**
   * Objective: Simulate the flow propagation through pipes, determining connectivity and flow availability from an initial point along the pipes, considering different types and their capacity.
5. **Dynamic Pipe Network Construction Algorithm (DFS):**
   * Objective: Simulate the construction of the pipe network dynamically. DFS could be used to explore possible connections and build pipe segments as the game progresses, adapting to game conditions.
6. **Pipe Network Optimization Algorithm:**
   * Objective: Optimize the pipe network to minimize costs or maximize flow efficiency. Factors to consider may include the total length of pipes or flow capacity to determine the most efficient connections.

**Step 4: Transition from Ideas to Preliminary Designs**

1. **Optimal Flow Algorithm:** Considers economic and efficiency aspects, aligning well with the pipe construction theme. However, it may be more complex to implement and require adjustments to balance flow and costs adequately.
2. **Change Propagation Algorithm:** Offers dynamic flexibility in adapting pipe systems to events or changes in the game. But it might require careful management to avoid unpredictable behaviors in the game.
3. **Pipe Exploration Algorithm:** Directly related to the central mechanics of the Pipe Mania game. It can become computationally expensive in large graphs.
4. **Network Construction Algorithm:** Aligns with the player's logic in Pipe Mania, building connections as the game progresses. The disadvantage is that it may require adjustments to balance construction and game complexity.
5. **Pipe Network Optimization Algorithm:** Strategic and integrates well with the game's mechanics. It can be complex and requires careful consideration of factors for efficient optimization.
6. **Breadth-First Search (BFS):** Effective for finding the shortest path between two points in unweighted graphs. Useful for controlled expansion from a starting point. Not efficient in dense graphs or with heavy-weighted edges.
7. **Depth-First Search (DFS):** Explores in depth and can be efficient for finding solutions in large graphs. Useful for pathfinding and connected components. Does not guarantee finding the shortest path and may fall into infinite cycles in non-acyclic graphs.
8. **Dijkstra's Algorithm:** Finds the shortest path in weighted graphs with non-negative weights. Suitable for pipe networks with representative distances. Does not handle negative weights well and can be time-consuming in large graphs.
9. **Warshall's Algorithm:** Finds all shortest paths between all pairs of vertices in weighted graphs. Useful for determining connectivity between points. Can be computationally expensive in large graphs and does not handle negative weights well.
10. **Minimum Spanning Tree (MST) - Prim:** Efficiently finds a minimum spanning tree. Suitable for optimizing connectivity in a pipe network. Does not handle graphs with negative-weight edges well.
11. **Minimum Spanning Tree (MST) - Kruskal:** Efficiently finds a minimum spanning tree and handles graphs with negative-weight edges well. May be less efficient in dense graphs.

**Step 5: Evaluation and Selection of the Best Solution**

This step will be taken to find the best solution for implementation; we will create a numerical evaluation system based on the following criteria:

A) Efficiency

B) Usability

C) Maintainability

D) Scalability

Enumerating each point from 1 to 5, where 1 is very poor and 5 is very good. At the end of each case, a sum will be made, and the higher the value, the more convenient it will be to use.

1. Breadth-First Search (BFS): A) 4 - Efficient for finding the shortest path in unweighted graphs. B) 5 - Easy to understand and use. C) 4 - Maintainable but may require adjustments for specific cases. D) 3 - Moderate scalability. Total: A) 19, B) 20, C) 19, D) 15

Efficiency Ranking:

* 1. Breadth-First Search (BFS)
  2. Depth-First Search (DFS)
  3. Dijkstra's Algorithm
  4. Minimum Spanning Tree (MST) - Prim
  5. Warshall's Algorithm

1. Optimal Flow Algorithm: A) 4 - Efficient when considering cost and flow capacity. B) 3 - Requires a deeper understanding of pipe economics. C) 4 - Maintainable, but optimization can be complex. D) 3 - Moderate scalability. Total: A) 19, B) 20, C) 19, D) 15

Efficiency Ranking:

* 1. Breadth-First Search (BFS)
  2. Depth-First Search (DFS)
  3. Dijkstra's Algorithm
  4. Minimum Spanning Tree (MST) - Prim
  5. Warshall's Algorithm
  6. Optimal Flow Algorithm

1. Change Propagation Algorithm: A) 3 - Efficient for dynamically adjusting connections. B) 4 - Intuitive, but may require adjustments to avoid unpredictable behaviors. C) 3 - Maintainable, but beware of complexity in dynamic changes. D) 3 - Moderate scalability. Total: A) 19, B) 20, C) 19, D) 15

Efficiency Ranking:

* 1. Breadth-First Search (BFS)
  2. Depth-First Search (DFS)
  3. Dijkstra's Algorithm
  4. Minimum Spanning Tree (MST) - Prim
  5. Warshall's Algorithm
  6. Optimal Flow Algorithm
  7. Change Propagation Algorithm

1. Pipe Exploration Algorithm: A) 3 - Efficient but can be costly in large graphs. B) 4 - Directly related to the central mechanics of the Pipe Mania game. C) 3 - Maintainable but may require optimizations for large pipe volumes. D) 3 - Moderate scalability. Total: A) 19, B) 20, C) 19, D) 15

Efficiency Ranking:

* 1. Breadth-First Search (BFS)
  2. Depth-First Search (DFS)
  3. Dijkstra's Algorithm
  4. Minimum Spanning Tree (MST) - Prim
  5. Warshall's Algorithm
  6. Optimal Flow Algorithm
  7. Change Propagation Algorithm
  8. Pipe Exploration Algorithm

1. Network Construction Algorithm: A) 4 - Efficient for dynamically building connections. B) 4 - Aligns with the player's logic in Pipe Mania. C) 4 - Maintainable and adaptable to construction changes. D) 3 - Moderate scalability. Total: A) 18, B) 19, C) 18, D) 15

Efficiency Ranking:

* 1. Breadth-First Search (BFS)
  2. Depth-First Search (DFS)
  3. Dijkstra's Algorithm
  4. Minimum Spanning Tree (MST) - Prim
  5. Warshall's Algorithm
  6. Optimal Flow Algorithm
  7. Change Propagation Algorithm
  8. Pipe Exploration Algorithm
  9. Network Construction Algorithm

1. Pipe Network Optimization Algorithm: A) 4 - Efficient for optimizing the pipe network. B) 4 - Strategic and integrates well with the game's mechanics. C) 4 - Maintainable with careful factor weighting. D) 3 - Reasonable scalability. Total: A) 18, B) 19, C) 18, D) 15

Efficiency Ranking:

* 1. Breadth-First Search (BFS)
  2. Depth-First Search (DFS)
  3. Dijkstra's Algorithm
  4. Minimum Spanning Tree (MST) - Prim
  5. Warshall's Algorithm
  6. Optimal Flow Algorithm
  7. Change Propagation Algorithm
  8. Pipe Exploration Algorithm
  9. Network Construction Algorithm
  10. Pipe Network Optimization Algorithm

Considering the rankings, we can conclude that we will use the algorithms that obtained the highest scores:

* BFS (19)
* DFS (20)
* Dijkstra (19)
* Warshall (18)
* MST - Prim (15)

7. Change Propagation Algorithm:

A) 3 - Efficient for dynamically adjusting connections.

B) 4 - Intuitive but may require adjustments to avoid unpredictable behaviors.

C) 3 - Maintainable, but be cautious of complexity in dynamic changes.

D) 3 - Moderate scalability.

8. Pipe Exploration Algorithm:

A) 3 - Efficient but can be costly in large graphs.

B) 4 - Directly related to the central mechanics of the game Pipe Mania.

C) 3 - Maintainable but may require optimizations for large volumes of pipes.

D) 3 - Moderate scalability.

9. Network Construction Algorithm:

A) 4 - Efficient for dynamically building connections.

B) 4 - Aligns with the player's logic in Pipe Mania.

C) 4 - Maintainable and adaptable to changes in construction.

D) 3 - Moderate scalability.

10. Pipe Network Optimization Algorithm:

A) 4 - Efficient for optimizing the pipe network.

B) 4 - Strategic and integrates well with the game's mechanics.

C) 4 - Maintainable with careful weighting of factors.

D) 3 - Reasonable scalability.

Sum of Evaluations:

A) 18

B) 19

C) 18

D) 15

Based on the scores, we can conclude that we will use the algorithms that obtained the following scores:

A) BFS 19

B) DFS 20

C) Dijkstra 19

D) Warshall 18

E) MST-Prim 15

**Step 6. Preparation of Reports and Specifications**

Problem Specification (in terms of input/output):

Problem: The problem, based on the initial statement, is to develop a game for one or more players that can be modeled using graphs. The game should meet specific requirements, including having a minimum of 50 vertices and 50 edges. The solution should involve applying at least two graph algorithms from the ones studied during the course, which include Graph Traversals (BFS, DFS), Shortest Path Algorithms (Dijkstra, Floyd-Warshall), and Minimum Spanning Tree (MST) Algorithms (Prim, Kruskal).

Additional requirements include developing two versions of the graph, ensuring the program can seamlessly switch between implementations, documenting each phase of the engineering process, and creating a user interface that allows interaction with the game functionalities.

Inputs:

The program should accept input for the following elements:

* Graph configuration: vertices, edges, and their connections.
* Game definition: rules, objectives, and any specific configurations.
* A structured input format, such as a file or a command-line interface, allowing the user to define the graph and game rules.

Example Input:

- Graph: Vertices {A, B, C}, Edges {(A, B), (B, C), ...}

- Game Rules: Scoring, Victory Conditions, etc.

Outputs:

* The program should provide output reflecting the game state, results of applied algorithms, and any relevant information. It could be a graphical interface showing the graph configuration and game progress.

Example Output:

- Game State: Score, Victory Conditions, etc.

- Results of Applied Algorithms: Shortest Paths, Minimum Spanning Tree, etc.

Considerations:

1. **Graph Configuration:**
   * Ensure that the program validates and handles input regarding graph configuration to prevent errors or unexpected behavior.
   * Implement mechanisms for users to easily define vertices, edges, and connections in a clear and intuitive way.
2. **Game Definition:**
   * Allow flexibility in defining game rules and objectives to accommodate a variety of game scenarios.
   * Consider incorporating error handling for rule definition to avoid inconsistencies.
3. **Input Format:**
   * Design a user-friendly input format, whether through a file or command-line interface, ensuring accessibility for users with varying technical backgrounds.
   * Include clear documentation or prompts to guide users through the input process.
4. **Output Clarity:**
   * Structure the output in a clear and concise manner, providing users with easily understandable information about the game state and algorithm results.
   * Consider using a combination of text and visual elements to enhance comprehension.
5. **Algorithm Switching:**
   * Implement a seamless mechanism for switching between two graph versions, ensuring minimal disruption to the user experience.
   * Provide user-friendly options for selecting and transitioning between graph implementations.
6. **Graph Algorithm Support:**
   * Implement the specified graph algorithms (BFS, DFS, Dijkstra, Floyd-Warshall, Prim, Kruskal) with a focus on correctness and efficiency.
   * Ensure that the program allows users to choose and apply at least two of these algorithms.
7. **User Interface (GUI):**
   * Design an intuitive and visually appealing GUI to enhance the user experience.
   * Include interactive elements to allow users to perform actions within the game easily.
   * Provide real-time updates in the GUI to reflect changes in the game state and algorithm results.
8. **Error Handling:**
   * Implement robust error-handling mechanisms to gracefully manage unexpected situations and provide informative error messages to users.
   * Include validation checks for input parameters and user actions.
9. **Documentation:**
   * Provide comprehensive documentation for the program, including input format, algorithm choices, and usage instructions.
   * Include in-app help features or tooltips to guide users through the program's functionalities.
10. **Testing:**
    * Conduct thorough testing to ensure the correctness and reliability of the program under various scenarios.
    * Consider usability testing to gather feedback on the user interface and overall user experience.
11. **Scalability:**
    * Design the program with scalability in mind, allowing it to handle larger graphs and more complex game scenarios efficiently.

(class diagram is in another document)

Pseudocode of the main system functionalities:

1. Initialize Game:

Procedure InitializeGame():

CreateEmptyGraph() // Function to create an empty graph

DefineGameRules() // Function to allow users to define game rules

InitializeGUI() // Function to set up the graphical user interface

DisplayWelcomeMessage()

2. User Interaction Loop:

While (GameNotOver):

GetUserInput() // Function to get user input (e.g., move, action)

ProcessUserInput() // Function to handle user input and update game state

ApplyGraphAlgorithm() // Function to apply selected graph algorithms

UpdateGUI() // Function to update the graphical user interface

CheckGameConditions() // Function to check if the game conditions for victory are met

3. Graph and Game Operations:

Procedure CreateEmptyGraph():

// Initialize an empty graph with 50 vertices and 50 edges

// Implement logic to allow switching between two graph versions

Procedure DefineGameRules():

// Allow users to define game rules, objectives, and conditions

// Implement logic for scoring and victory conditions

Function GetUserInput():

// Get user input, such as selecting vertices, defining connections, or making game moves

// Validate and process user input

Procedure ProcessUserInput():

// Update the graph based on user input (e.g., connect vertices, modify edges)

Procedure ApplyGraphAlgorithm():

// Allow users to choose and apply at least two graph algorithms (BFS, DFS, Dijkstra, Floyd-Warshall, Prim, Kruskal)

// Update the graph based on algorithm results

Procedure CheckGameConditions():

// Check if the game conditions for victory are met

// Display victory/defeat messages and end the game if necessary

4. Graphical User Interface (GUI):

Procedure InitializeGUI():

// Set up the graphical user interface, including the game board, controls, and information displays

Procedure UpdateGUI():

// Update the GUI to reflect the current state of the game, graph, and algorithm results

// Display relevant information, scores, and visual representations of the graph

5. Game End:

Procedure DisplayVictoryMessage():

// Display a victory message when the game conditions are met

Procedure DisplayDefeatMessage():

// Display a defeat message when the game conditions are not met

Procedure DisplayGoodbyeMessage():

// Display a goodbye message when the user decides to exit the game

**Step 7. Implementation of the Design**

Task Lists to Implement:

**Initialization and Setup:**

1. **Create Empty Graph:**
   * Implement a function to create an empty graph with the specified minimum of 50 vertices and 50 edges.
   * Include logic to handle the two versions of the graph.
2. **Define Game Rules:**
   * Develop a mechanism for users to define game rules, including scoring, objectives, and victory conditions.
3. **Initialize GUI:**
   * Set up the graphical user interface, including the game board, controls, and information displays.
   * Ensure that the GUI reflects the initial state of the game.

**User Interaction Loop:** 4. **Get User Input:**

* Implement a function to get user input, allowing them to interact with the game.
* Handle input validation to ensure the correctness of user actions.

1. **Process User Input:**
   * Develop logic to process user input and update the game state accordingly.
   * Include actions such as connecting vertices, modifying edges, or making moves.
2. **Apply Graph Algorithm:**
   * Allow users to choose and apply at least two graph algorithms from the specified set.
   * Implement logic to update the graph based on the results of the selected algorithms.
3. **Update GUI:**
   * Develop functions to update the graphical user interface after each user action or algorithm application.
   * Ensure the GUI reflects the current state of the game, graph, and algorithm results.
4. **Check Game Conditions:**
   * Implement logic to check if the game conditions for victory are met.
   * Display victory/defeat messages and end the game accordingly.

**Graph and Game Operations:** 9. **Graph Manipulation Functions:**

* Implement functions to modify the graph based on user input and algorithm results.
* Include actions like connecting vertices, updating edges, and changing the graph version.

**Game End:** 10. **Display Victory Message:** - Create a function to display a victory message when the game conditions are met.

1. **Display Defeat Message:**
   * Implement a function to display a defeat message when the game conditions are not met.
2. **Display Goodbye Message:**
   * Develop a function to display a goodbye message when the user decides to exit the game.

**Documentation and Testing:** 13. **Document the Code:** - Provide inline comments and documentation to explain the purpose and functionality of each code segment.

1. **Test the System:**
   * Conduct comprehensive testing to ensure the correctness and reliability of the implemented functionalities.
   * Test various scenarios, including different user inputs and algorithm choices.

**Additional Considerations:** 15. **Error Handling:** - Implement robust error-handling mechanisms to gracefully manage unexpected situations.

1. **Usability Testing:**
   * Gather feedback on the user interface and overall user experience through usability testing.
2. **Scalability Testing:**
   * Evaluate the program's performance and scalability, considering larger graphs and more complex game scenarios.

**Bibliografia**

<https://es.wikipedia.org/wiki/Grafo>

<https://www.encora.com/es/blog/dfs-vs-bfs>

<http://atlas.uned.es/algoritmos/voraces/dijkstra.html#:~:text=El%20algoritmo%20de%20Dijkstra%2C%20tambi%C3%A9n,con%20pesos%20en%20cada%20arista>.

<https://medium.com/algoritmo-floyd-warshall/algoritmo-de-floyd-warshall-e1fd1a900d8>

<https://prezi.com/hg8iod-dutfx/arboles-recubridores-minimales-algoritmos-de-kruskal-y-prim/#:~:text=Un%20%C3%A1rbol%20recubridor%20m%C3%ADnimo%20o,tiene%20un%20bosque%20recubridor%20m%C3%ADnimo>.

<https://arodrigu.webs.upv.es/grafos/doku.php?id=algoritmo_prim>

<http://repositorio.puce.edu.ec/bitstream/handle/22000/12141/Tesis-Algoritmo%20de%20Kruskal%20v1%20(1).pdf?sequence=1&isAllowed=y#:~:text=El%20algoritmo%20de%20Kruskal%20es,varios%20nodos%20comunicados%20entre%20s%C3%AD>.