**Engineering Method**

**Step 1. Problem Identification**

**Context:**  "Pipemania" is a classic puzzle game that challenges players to build a system of pipes to guide the flow of a liquid from a starting point to a final destination, based on directed graphs with more than 50 nodes and edges. The goal is to connect the strategically provided pipe pieces before the liquid starts flowing, preventing it from spilling and causing a disaster. Which requires careful planning and quick decision-making to achieve success. With its combination of problem-solving and mental speed, Pipemania has entertained players of all ages.

|  |  |
| --- | --- |
| Symptoms & Needs | Description |
| Graphical Interface | In order to make the game easier for the user to understand, the decision will be made to implement a graphical interface that gives them a visual representation of the game world, thus improving their experience and accessibility. |
| Design of the world | In order to enhance the player's immersion, a virtual environment will be developed that simulates a board of pipes that the user will be able to connect from the main to the final |
| Gameplay | The game is conceived as a puzzle, with the player being responsible for prudently managing the pipes to reach the end of the pipe. In addition, you are offered the option to forgo the most direct route (game options) or shortest to complete the game. |
| Algorithms to use | Algorithms need to show something in order to contribute to the player in some way, whether they're looking to win quickly, complete pipes, or any other option. As algorithms to use, you should implement one that gives you the easiest route to get to the end of the pipe. |
| Algorithm Testing | Due to the high complexity of representing the connections between the initial and final pipes, in addition to the complexity of the algorithms that contribute something to the player, these must be tested. |
| Performance Optimization | The system should run in real-time in order to make the gameplay feel as snappy as possible |

The software needs to meet the following characteristics:

* Be a video game
* That works through your own implementation of graphs
* The graph must be implemented in two ways: as a list or as an adjacency matrix
* The software must use at least two graph algorithms, but have 6 functional ones (DFS, BSF, Dijkstra, Kruskal, Floyd Warshall, and Prim).

-

**REQUIREMENTS**

**PROBLEM SPECIFICATION TABLE**

|  |  |
| --- | --- |
| **CLIENT** | Game development applicant. |
| **USER** | Player |
| **FUNCTIONAL REQUERIMENTS** | * R1. The game must allow the player to configure the graph type. * R2. The game must allow the player to initialize with a valid board. * R3. The game must allow the player to place pipes on the board. * R4. The game must allow the player to give up. * R5. The game must allow the player to clear the board. * R6. The game must allow the player to validate his path from the source to the drain. |
| **CONTEXT OF THE PROBLEM** | A piping that challenges players to effectively connect a fountain to a drain is under development. Both elements appear randomly on a game board, which serves as a canvas for creating the piping system. To enhance the game experience, a graph traversal algorithm is applied. This algorithm not only generates boards with valid pipeline solutions, but also validates the solutions proposed by the players. In addition, a minimum path algorithm highlights the shortest route between the source and the drain, providing players with a visual reference and influencing their scores. The game allows flexibility in the implementation of the graph, offering players a choice between adjacency list and adjacency matrix implementations. |
| **NON-FUNCTIONAL REQUERIMENTS** | NFR 1: The game must be implemented on a JavaFX interface. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Identifier or name** | **R1. The game must allow the player to configure the graph type.** | | |
| **Summary** | The game must be playable in either of the two implementations for this case: adjacency matrix graph or adjacency list graph. To this, before starting a game, the player will be asked to indicate graph type. The game will try to generate a game with this choice. | | |
| **Inputs** | **Input name** | **Data type** | **Condition or valid values** |
| Graph type | String | * Adjacency list * Matrix list |
| **General activities required to achieve the results** | * Select graph type. | | |
| **Outcome or Postcondition** | Generation of the game on the selected graph. | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **Identifier or name** | **R2. The game must allow the player to initialize with a valid board.** | | |
| **Summary** | The game must initialize a board with at least one valid solution. First blocked cells will be generated and the source and drain will be placed in a random position on the board. Then the game will use an algorithm to verify that there is a valid path from the source to the drain. If a route exists, the game will be ready for the user. If not, the user will be told that a valid game was not generated and will be asked to try again. | | |
| **General activities required to achieve the results** | * Click on the play button. * Select graph type. | | |
| **Outcome or Postcondition** | Generation of a valid game or error message. | | |
| **Outputs** | **Output name** | **Data type** | **Selection or repetition condition** |
| Error message | String | The generated game has no solution. |

|  |  |
| --- | --- |
| **Identifier or Name** | **R3. The game must allow the player to place pipes on the board** |
| **Summary** | The game should allow the player to place pipes on any cell of the board except those that are blocked. There are six types of pipes and the player can change them by pressing several times on the desired cell. |
| **General activities required to achieve the results** | * Click on the box where the player wants to place the pipe. |
| **Outcome or Postcondition** | An image of the pipe is displayed in the selected cell. |

|  |  |
| --- | --- |
| **Identifier or name** | **R4. The game must allow the player to validate his path from the source to the drain.** |
| **Summary** | The game must have the option for the player to give up. To do this, the player must select the option and the game will ask for confirmation. If he goes ahead the game will eliminate all the pipes present and show the player the shortest route from the source to the drain and the game will end. |
| **General activities required to achieve the results** | * Click on give up button |
| **Outcome or Postcondition** | The shortest route from the source to the drain will be shown and the game will end. |

|  |  |
| --- | --- |
| **Identifier or Name** | **R5. The game must allow the player to clear the board.** |
| **Summary** | The game must allow the player to clear the board whenever he/she wishes. |
| **General activities required to achieve the results** | * Click on clear button |
| **Outcome or Postcondition** | All the pipes that the user had placed on the board are removed. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Identifier or name** | **R6. The game must allow the player to validate his path from the source to the drain.** | | |
| **Summary** | The game must have the option for players to validate the route they took from the source to the drain. To do this, the player must select the option and the game will verify that the route chosen is valid. If so, a message box will be displayed with the summary of the game and the score. If not, the error will be indicated and the board will be cleared for the player to try again. | | |
| **General activities required to achieve the results** | * Click on validate button. | | |
| **Outcome or Postcondition** | An information box is displayed and/or the board is cleared. | | |
| **Outputs** | **Output name** | **Data type** | **Selection or repetition condition** |
| Confirmation message | String | The player has won the game or error, the solution is incorrect. |

**Step 2. Collection of Information**

A pipe board is used as a stage, with the inclusion of at least 50 places with obstacles distributed in various locations, representing the places where the user will be able to place the pipes, in order to connect the main one with the final one.

**Graphs:**

A graph is a data structure that consists of a group of nodes or vertices and edges, the former representing the units of the elements to be represented and the edges representing the connections of these elements. An example of a graph would be a social network, people would be the vertices, and friendship relationships are edges that bring the vertices together.

There are several types of graphs, but the most important ones are directed and undirected graphs: in the first the relationships are not binary, that is, that vertex "A" has a relationship "B" with vertex "C" is not the same as vertex "C" having a relationship "B" with vertex "A"; conversely, in the undirected graph, if there is a relation "B" from "A" to "C", it is always true that there is a relation "B" from "C" to "A".

Another type of graph is graphs with weights or weights, these indicate a certain "resistance" of getting from one vertex to another through an edge. An example of graphs with weights would be roads: to get from one city to another there is a cost, a weight, either in time or gasoline consumption.

There are two main ways to represent a graph: the adjacency list and the adjacency matrix. In the first type, in order to know which nodes can be accessed by node number "i", you have to call the values that are in that position. In the second type you have a matrix that contains the weights (if it is a graph without weights you put 1) to get from the node of the row to the node of the columns, an example would be that when you see the position (i, j) you get a number (the weight) that says if you can get from node i to node j.

**References:**

GeeksforGeeks. (2023). Graph Data Structure and Algorithms. <https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/>

Retrieved from <https://www.javatpoint.com/java-graph>

There are several operations with graphs such as adding, removing, or modifying vertices and edges. Additionally, there are several algorithms for different functions:

**Paths:**

**DFS:** In DFS, you start from a source node and explore as deeply as possible along each branch before backing up. It uses a stack (or recursion) to keep track of which nodes need to be visited.

**BFS:** In BFS, you start from a source node and explore all of its neighbors before moving on to the neighbors of these nodes. It uses a queue to keep track of the nodes that need to be visited. It is often used to find the shortest path between two nodes in an unweighted graph.

In both cases the complexity is O(V + E), where V is the number of nodes and E is the number of edges in the graph.

**References**

*Murillo, J. (2022, 18 julio). DFS vs BFS. Encora.*[*https://www.encora.com/es/blog/dfs-vs-bfs*](https://www.encora.com/es/blog/dfs-vs-bfs)

**Minimum Paths:**

**Floyd Warshall: The Floyd-Warshall** algorithm is used to find the shortest paths between all pairs of nodes in a directed or undirected, weighted or unweighted graph. It works even when there are edges with negative weights, but it doesn't work properly in the presence of negative cycles. The algorithm uses an array to keep track of the minimum distances between all node pairs. The complexity of the algorithm is O(V^3), where V is the number of nodes in the graph.

**Dijkstra: Dijkstra**'s algorithm is used to find the shortest path from one source node to all other nodes in a weighted graph with non-negative edges. It maintains a list of unvisited nodes, and at each step, chooses the node with the minimum known distance from the source node. It is more efficient than Floyd-Warshall for sparse or medium-sized graphs. The complexity of the algorithm is O((V + E) \* log(V)), where V is the number of nodes and E is the number of edges in the graph.

*Best routes selection using Dijkstra and Floyd-Warshall algorithm. (2017, 1 octubre). IEEE Conference Publication | IEEE Xplore.* [*https://ieeexplore.ieee.org/document/8265662*](https://ieeexplore.ieee.org/document/8265662)

**Minimum weight subgraphs:**

**Prim:** It starts with an arbitrary node and repeatedly selects the shortest edge that connects a node in the tree to a node outside it. It can be implemented using a priority queue (heap) to efficiently select the shortest edges. The complexity of the algorithm is O(E + V \* log(V)), where E is the number of edges and V is the number of nodes.

**Kruskal:** Sorts all edges by weight and adds them to the minimum spanning tree in ascending order until all nodes are connected. Use disjoint sets to see if adding an edge cycles the current tree. The complexity of the algorithm is O(E \* log(E)), where E is the number of edges.

*Minimum weight trees: Prim and Kruskal algorithms — discrete math for data science. (n.d.).* [*https://madi.nekomath.com/P5/ArbolPesoMin.html*](https://madi.nekomath.com/P5/ArbolPesoMin.html)

**Step 3. Search for Creative Solutions**

**Graphical Interface:**

The objectives of the graphical interface are:

* Ease of use and understanding by the user
* Graphically display algorithms (Dijkstra)
* Interact with the game elements (the pipes).
* Improve game immersion.

**In this case, we have planned to implement:**

Optimal Flow Algorithm: To determine the most efficient solution between two points, this could take into account factors such as the cost of building the pipes and the "flow" capacity.

**Graph Structure**

Change propagation, as this would simulate how changes in one part of the graph affect other parts, so we can better and more dynamically adjust the pipe connections. We also plan to implement a:

**Dynamic network optimization algorithm:** this would adjust the pipe network based on what is being ordered at the moment and based on the game conditions.

To simulate the propagation of flow through the pipes we plan to use a pipe exploration algorithm, in this way the connectivity and availability of flow can be determined from a starting point along the pipes, considering the different types and their capacity. We also plan to design an algorithm that simulates the construction of the pipeline network dynamically. The DFS could be used to explore possible connections and build pipe segments as it goes, adapting to the conditions of the game.

Finally, we plan to implement an algorithm that optimizes the pipe network to minimize costs or maximize flow efficiency. You could consider factors such as the total length of the pipes or the flow capacity to determine the most efficient connections.

**Game World Appearance:**

The only purpose of the appearance of the world, as it does not directly affect the gameplay, is the appearance and immersion of the user, so the original PipeMania game must be referenced but with an interactive board.

**Crucial Algorithms:**

Two crucial algorithms are available to help the player:

* Dijkstra, who find the way with the lowest cost to the last pipeline
* Kruskal and Prim, which shows the 10 paths with the least minimum overlay tree weight. Remember that players must choose between efficiency and the most suitable place to reach the last pipe.

**Step 4. Transitioning from Ideas to Preliminary Designs**

It should be noted that you have to use a graphical interface, so JavaFX, Java Swing, SWT or AWT were considered. But in the end, JavaFX was chosen.

**On the other hand, it was thought that it could be used with respect to implementation:**

**Optimal Flow Algorithm:** Considers economic and efficiency aspects, adapting well to the theme of pipeline construction, although it may be more complex to implement and require adjustments to properly balance flow and costs.

**Change Propagation Algorithm:** Offers dynamic flexibility in adapting the piping system to events or changes in the game. But it might require careful management to avoid unpredictable behaviors in the game.

**Pipe Exploration Algorithm:** This is directly related to the core mechanics of the game PipeMania. It can become computationally expensive in large graphs.

**Network Building Algorithm:** Aligns with the player's logic in Pipe Mania, building connections as you go The downside is that it may require tweaks to balance the build and complexity of the game.

**Pipe Network Optimization Algorithm:** Strategic and integrates well with the game mechanics.

It can be complex and needs careful weighting of factors for efficient optimization.

**Amplitude Traversal (BFS):** It is effective for finding the shortest path between two points in unweighted graphs. Useful for controlled expansion from a starting point. It is not efficient in dense graphs or graphs with heavy edges.

**In-Depth Walkthrough (DFS):** Explores in depth and can be efficient in finding solutions in large graphs. Useful for finding paths and connected components. It does not guarantee to find the shortest path and can fall into infinite cycles on non-acyclic graphs.

**Dijkstra's algorithm:** Find the shortest path in weighted graphs with non-negative weights. Suitable for pipe networks with representative distances. It doesn't handle negative weights well and can be time-intensive in large graphs.

**Warshall algorithm:** Finds all the shortest paths between all vertex pairs in weighted graphs. Useful for determining connectivity between points. It can be computationally expensive in large graphs and doesn't handle negative weights well.

**Minimum Cover Tree (MST) - Prim**: Find a minimum cover tree efficiently. Suitable for optimizing connectivity in a pipeline network. It does not handle graphs with negative weight edges well.

**Minimum Cover Tree (MST) - Kruskal**: Finds a minimum cover tree efficiently and handles graphs with negative weight edges well. It may be less efficient in dense graphs.

**Step 5: Evaluate and select the best solution**

Depending on the proposed solutions to the problem, some criteria are presented that may be relevant to evaluate the ideas generated in the brainstorming phase for the implementation of the system. It should be noted that the following criteria will only be used to evaluate the algorithms that will solve the needs of the game, not the final design of the interface:

**Criterion A: Efficiency** - This refers to the ability of the system to process and deliver the required data and services quickly and without interruption. Efficiency is measured in terms of speed of response or by the temporal complexity of operations on a scale of 1 to 5, with 1 being very low efficiency and 5 being very high efficiency.

**Criterion B: Scalability** - Refers to the ability of the system to grow and evolve without restriction, and to meet future business or user needs without compromising its quality or stability. It can be rated on a scale of 1 to 5, with 1 being very low scalability and 5 being very high scalability.

**Criterion C: Development Time** – This refers to the time required to complete the development of a specific system or functionality. This criterion is affected by factors such as the complexity of the system and the airline's urgency to have the system in place. It can be evaluated on a scale of 1 to 5, with 1 being too long and 5 being too little time.

Idea evaluation, where the highest score is the best alternative:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Alternative | Criterion A | Criterion B | Criterion C | Total |
| Breadth First Search | 5 | 4 | 5 | 14 |
| Depth First Search | 4 | 4 | 3 | 11 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Alternative | Criterion A | Criterion B | Criterion C | Total |
| Dijkstra Algorithm | 5 | 4 | 4 | 13 |
| Floyd Warshall Algorithm | 3 | 3 | 3 | 9 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Alternative | Criterion A | Criterion B | Criterion C | Total |
| Kruskal Algorithm | 5 | 3 | 3 | 11 |
| Prim 2015 | 3 | 3 | 3 | 9 |

**Based on the above assessment, the best alternatives are:**

- Find an arrival route from the main to the final pipe: Dijkstra's algorithm.

- Determine the main route that passes through all positions: Kruskal's algorithm.