**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 and it should work over any type of graph (adjacency matrix or adjacency list).

A piping game has been chosen in which players must efficiently connect a fountain with a drain. Both elements (source and drain) appear randomly on a game board, which serves as a canvas for creating the piping system. The game must be able to generate a board with at least one valid solution. It must allow the player to place pipes on the squares wherever he/she wishes, except on the randomly generated blocked squares. If the player wishes to clear the board, there shall also be an option to allow this. It should also allow the player to validate the proposed solutions for connecting the fountain and the drain and calculate a score based on the path chosen. Finally, it must have an option that allows the player to exit the game to be shown a solution that could have been chosen for that game.

**Step 1: Problem Identification**

Identification of needs and symptoms (only requirements that could be addressed from different solutions were chosen):

* The game must allow the player to configure the graph type.
* The game must allow the player to initialize with a valid board.
* The game must allow the player to place pipes on the board.
* The game must allow the player to give up and be shown a valid solution.
* The game must allow the player to clear the board.
* The game must allow the player to validate his path from the source to the drain and calculate the score according to the chosen path.

*(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.

*Minimum Spanning Tree (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**

Requirements were chosen that can be addressed from a variety of creative solutions:

* The game must allow the player to initialize with a valid board.
* The game must allow the player to give up and be shown a valid solution.
* The game must allow the player to validate his path from the source to the drain and calculate the score according to the chosen path.

3.1. Generate a valid board (At least one path from the source to the drain)

* BFS (Breadth-First Search)
* DFS (Depth-First Search)
* Dijkstra's Algorithm
* Floyd-Warshall Algorithm
* Prim's Algorithm
* Kruskal's Algorithm

3.2. Give up and show a valid solution (The shortest path from the source to the drain)

* BFS (Breadth-First Search)
* DFS (Depth-First Search)
* Dijkstra's Algorithm
* Floyd-Warshall Algorithm
* Prim's Algorithm:
* Kruskal's Algorithm

3.3.1. Validate solution (The source and the drain are properly connected)

* BFS (Breadth-First Search)
* DFS (Depth-First Search)
* Dijkstra's Algorithm
* Floyd-Warshall Algorithm
* Prim's Algorithm:
* Kruskal's Algorithm

3.3.2. Calculate the score according to the chosen path (Compare it with the shortest path)

* BFS (Breadth-First Search)
* DFS (Depth-First Search)
* Dijkstra's Algorithm
* Floyd-Warshall Algorithm
* Prim's Algorithm:
* Kruskal's Algorithm

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

4.1. Generate a valid board (At least one path from the source to the drain)

* BFS (Breadth-First Search): Employing BFS is instrumental in generating a valid board, ensuring the existence of at least one path from the source to the drain. Although particularly effective in unweighted graphs, it may not be the most efficient choice for dense graphs or those with heavy-weighted edges.
* DFS (Depth-First Search): Similar to BFS, DFS can contribute to generating a valid board with at least one path from the source to the drain. While efficient for large graphs, it doesn't guarantee the shortest path and may lead to infinite cycles in non-acyclic graphs.
* Dijkstra's Algorithm: This algorithm proves valuable in generating a valid board with an optimal path from the source to the drain, especially in weighted graphs with non-negative weights. However, it may be time-consuming in large graphs and does not handle negative weights well.
* Floyd-Warshall Algorithm: Particularly useful for finding all-pair shortest paths, including a path from the source to the drain. However, it can be computationally expensive in large graphs and does not handle negative weights well.
* Prim's Algorithm: This algorithm can be applied to create an efficient structure of connections on the board, optimizing connectivity in a pipe network. However, it may not handle graphs with negative-weight edges well.
* Kruskal's Algorithm: Similar to Prim's, Kruskal's algorithm can be applied for efficient pipe connections and an optimized board structure. However, it may be less efficient in dense graphs.

4.2. Give up and show a valid solution (The shortest path from the source to the drain)

* BFS (Breadth-First Search): Utilizing BFS to find the shortest path ensures a valid and optimal solution. However, its efficiency may be compromised in dense graphs or those with heavy-weighted edges.
* DFS (Depth-First Search): DFS can find a path, though not necessarily the shortest. It can be employed to display any valid solution but may not be the most efficient choice for finding the shortest path.
* Dijkstra's Algorithm: Guarantees the shortest path if weights are involved, providing an optimal solution. However, it may be time-consuming in large graphs and does not handle negative weights well.
* Floyd-Warshall Algorithm: Finds the shortest path between the source and the drain, ensuring an optimal solution, but it can be computationally expensive in large graphs and does not handle negative weights well.
* Prim's Algorithm: Can be adapted to show an efficient solution, optimizing connectivity in the pipe network. However, it may not handle graphs with negative-weight edges well.
* Kruskal's Algorithm: Similar to Prim's, Kruskal's algorithm could be adapted to show an efficient solution, though it may be less efficient in dense graphs.

4.3.1 Validate the solution (The source and the drain are properly connected)

* BFS (Breadth-First Search): Useful for verifying the connectivity between the source and the drain, especially in unweighted graphs. However, its efficiency may be compromised in dense graphs or those with heavy-weighted edges.
* DFS (Depth-First Search): Useful for checking the connectivity between the source and the drain, particularly in large graphs. However, it doesn't guarantee finding the shortest path and may lead to infinite cycles in non-acyclic graphs.
* Dijkstra's Algorithm: Inherently ensures connectivity when it finds the shortest path. However, it may be time-consuming in large graphs and does not handle negative weights well.
* Floyd-Warshall Algorithm: By determining all-pair shortest paths, it indirectly verifies connectivity. However, it can be computationally expensive in large graphs and does not handle negative weights well.
* Prim's Algorithm: Can be adapted to verify the connectivity of the board, optimizing connectivity in the pipe network. However, it may not handle graphs with negative-weight edges well.
* Kruskal's Algorithm: Similar to Prim's, Kruskal's algorithm could be adapted to verify the connectivity of the board, though it may be less efficient in dense graphs.

4.3.2 Calculate the score according to the chosen path (Compare it with the shortest path)

* BFS (Breadth-First Search): Utilize BFS to find the chosen path, comparing it with the shortest path for score calculation. Its efficiency may vary in dense graphs or those with heavy-weighted edges.
* DFS (Depth-First Search): Similar to BFS, DFS finds a path, but not necessarily the shortest. Score calculation involves comparing it with the shortest path. It may not be the most efficient choice for finding the shortest path.
* Dijkstra's Algorithm: Finds the chosen path and provides the shortest path, making score calculation straightforward. However, it may be time-consuming in large graphs and does not handle negative weights well.
* Floyd-Warshall Algorithm: Provides the shortest path, allowing easy comparison with the chosen path for score calculation. However, it can be computationally expensive in large graphs and does not handle negative weights well.
* Prim's Algorithm: Can be adapted to calculate the score based on the structure of the board, optimizing connectivity in the pipe network. However, it may not handle graphs with negative-weight edges well.
* Kruskal's Algorithm: Similar to Prim's, Kruskal's algorithm could be adapted to calculate the score based on the structure of the board, though it may be less efficient in dense graphs.

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

This step will be performed to find the best solution to implement, creating a numerical evaluation system based on the following criteria:

1. Efficiency
2. Usability
3. Maintainability
4. Scalability

Each point will be rated from 1 to 5, with 1 being very poor and 5 being excellent. At the end of each case, a sum will be made, and the higher the value, the more convenient it will be to use.

5.1. Generate a valid board.

* Breadth-First Search (BFS):
  + Efficiency: Effective for pathfinding. (5)
  + Usability: Suitable for controlled expansion. (4)
  + Maintainability: Reasonable maintenance. (3)
  + Scalability: Scalable, but not the most efficient in dense graphs. (3)

Total: 15

* Depth-First Search (DFS):
  + Efficiency: Effective but doesn't guarantee the shortest path. (3)
  + Usability: Useful for pathfinding and connected components. (4)
  + Maintainability: Reasonable maintenance. (3)
  + Scalability: Efficient for large graphs. (4)

Total: 14

* Dijkstra's Algorithm:
  + Efficiency: Efficient just for finding the optimal path. (3)
  + Usability: Suitable for pipe networks. (4)
  + Maintainability: Moderate maintenance. (3)
  + Scalability: Can be time-consuming in large graphs. (3)

Total: 13

* Floyd-Warshall Algorithm
  + Efficiency: Effective but for finding all-pair shortest paths. (3)
  + Usability: Useful for determining connectivity. (4)
  + Maintainability: Moderate maintenance. (3)
  + Scalability: Computationally expensive in large graphs. (2)

Total: 12

* Prim's Algorithm:
  + Efficiency: Efficient but for creating an efficient structure. (3)
  + Usability: Suitable for optimizing connectivity. (4)
  + Maintainability: Moderate maintenance. (3)
  + Scalability: May not handle negative-weight edges well. (3)

Total: 13

* Kruskal's Algorithm:
  + Efficiency: Efficient for pipe connections. (4)
  + Usability: Similar to Prim's. (4)
  + Maintainability: Moderate maintenance. (3)
  + Scalability: Less efficient in dense graphs. (2)

Total: 13

5.2. Give up and show a valid solution.

* Breadth-First Search (BFS):
  + Efficiency: Effective for finding the shortest path. (4)
  + Usability: May struggle in dense graphs. (3)
  + Maintainability: Reasonable maintenance. (3)
  + Scalability: Scalable, but not the most efficient in dense graphs. (3)

Total: 13

* Depth-First Search (DFS):
  + Efficiency: May not find the shortest path. (2)
  + Usability: Useful for displaying any valid solution. (4)
  + Maintainability: Reasonable maintenance. (3)
  + Scalability: Efficient for large graphs. (4)

Total: 13

* Dijkstra's Algorithm:
  + Efficiency: Guarantees the shortest path. (5)
  + Usability: Suitable for optimal solutions. (4)
  + Maintainability: Moderate maintenance. (3)
  + Scalability: Can be time-consuming in large graphs. (3)

Total: 15

* Floyd-Warshall Algorithm:
  + Efficiency: Effective for optimal solutions. (4)
  + Usability: Finds the shortest path (4)
  + Maintainability: Moderate maintenance. (3)
  + Scalability: Computationally expensive in large graphs. (2)

Total: 13

* Prim's Algorithm:
  + Efficiency: Can be adapted to show an efficient solution. (4)
  + Usability: May not handle negative-weight edges well. (3)
  + Maintainability: Moderate maintenance. (3)
  + Scalability: May not handle negative-weight edges well. (3)

Total: 13

* Kruskal's Algorithm:
  + Efficiency: Can be adapted to show an efficient solution. (4)
  + Usability: May be less efficient in dense graphs. (2)
  + Maintainability: Moderate maintenance. (3)
  + Scalability: May be less efficient in dense graphs. (2)

Total: 11

5.3.1. Validate solution.

* Breadth-First Search (BFS):
  + Efficiency: Effective for verifying connectivity. (5)
  + Usability: May struggle in dense graphs. (3)
  + Maintainability: Reasonable maintenance. (3)
  + Scalability: Scalable, but not the most efficient in dense graphs. (3)

Total: 14

* Depth-First Search (DFS):
  + Efficiency: Useful for checking connectivity. (4)
  + Usability: May lead to infinite cycles. (2)
  + Maintainability: Reasonable maintenance. (3)
  + Scalability: Efficient for large graphs. (4)

Total: 13

* Dijkstra's Algorithm:
  + Efficiency: Inherently ensures connectivity. (3)
  + Usability: Suitable for pipe networks. (4)
  + Maintainability: Moderate maintenance. (3)
  + Scalability: Can be time-consuming in large graphs. (3)

Total: 13

* Floyd-Warshall Algorithm:
  + Efficiency: Indirectly verifies connectivity. (3)
  + Usability: Useful for determining connectivity. (3)
  + Maintainability: Moderate maintenance. (3)
  + Scalability: Computationally expensive in large graphs. (2)

Total: 11

* Prim's Algorithm:
  + Efficiency: Can be adapted to verify connectivity. (3)
  + Usability: May not handle negative-weight edges well. (3)
  + Maintainability: Moderate maintenance. (3)
  + Scalability: May not handle negative-weight edges well. (3)

Total: 12

* Kruskal's Algorithm:
  + Efficiency: Can be adapted to verify connectivity. (3)
  + Usability: Similar to Prim's. (3)
  + Maintainability: Moderate maintenance. (3)
  + Scalability: May be less efficient in dense graphs. (2)

Total: 11

5.3.2. Calculate the score according to the chosen path.

* Breadth-First Search (BFS):
  + Efficiency: Utilize BFS for the chosen path. (3)
  + Usability: May struggle in dense graphs. (3)
  + Maintainability: Reasonable maintenance. (3)
  + Scalability: Scalable, but not the most efficient in dense graphs. (3)

Total: 12

* Depth-First Search (DFS):
  + Efficiency: May not find the shortest path. (2)
  + Usability: May not be the most efficient choice. (2)
  + Maintainability: Reasonable maintenance. (3)
  + Scalability: Efficient for large graphs. (4)

Total: 11

* Dijkstra's Algorithm:
  + Efficiency: Provides the shortest path. (5)
  + Usability: Suitable for optimal solutions. (4)
  + Maintainability: Moderate maintenance. (3)
  + Scalability: Can be time-consuming in large graphs. (3)
  + Total: 15
* Floyd-Warshall Algorithm:
  + Efficiency: Effective for optimal solutions. (4)
  + Usability: Provides the shortest path (4)
  + Maintainability: Moderate maintenance. (3)
  + Scalability: Computationally expensive in large graphs. (2)
  + Total: 13
* Prim's Algorithm:
  + Efficiency: Can be adapted to calculate the score. (4)
  + Usability: May not handle negative-weight edges well. (3)
  + Maintainability: Moderate maintenance. (3)
  + Scalability: May not handle negative-weight edges well. (3)

Total: 13

* Kruskal's Algorithm:
  + Efficiency: Can be adapted to calculate the score. (4)
  + Usability: Similar to Prim's. (3)
  + Maintainability: Moderate maintenance. (3)
  + Scalability: May be less efficient in dense graphs. (2)

Total: 12

In summary, based on the evaluation criteria provided, the BFS algorithm is the most suitable for requirements 5.1 and 5.3.1:

5.1. Generate a valid board.

Reasoning: BFS proved effective in generating a valid board, ensuring at least one path from the source to the drain. Its controlled expansion approach from an initial point translates to efficiency in terms of generating valid boards.

5.3.1. Validate solution.

Reasoning: BFS again proves to be the right choice for verifying connectivity between the source and the drain. Although it may have limitations in dense graphs, its overall efficiency and ability to verify connectivity are significant advantages.

On the other hand, Dijkstra's Algorithm is the most suitable for requirements 5.2, and 5.3:

5.2. Give up and show a valid solution.

Dijkstra guarantees the generation of a valid solution with the shortest path from the source to the drain. Its ability to find optimal paths and its efficiency make it the right choice for this task.

5.3.2. Calculate the score according to the chosen path.

Dijkstra is selected to calculate the score due to its ability to find the shortest path ((the one to be compared with the player's solution). This feature simplifies score calculation and ensures optimal solutions.

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

Problem Specification (in terms of input/output):

Problem: Piping game

Inputs: Graph type configuration (Adjacency list or adjacency matrix)

Outputs: The program should provide output reflecting the game state, results of applied algorithms, and any relevant information.

Considerations:

The game must be of one or more players, must be modeled using graphs, with a minimum of 50 vertices and 50 edges, and must apply at least two (2) of the graph algorithms: Paths over Graphs (BFS, DFS), Minimum Weight Paths (Dijkstra, Floyd-Warshall), Minimum Cover Tree -MST- (Prim, Kruskal).

The minimum requirements are the following:

* Develop two versions of graph (your solution must work seamlessly with both versions, i.e., the program must support the change of the implementation used at any time and work well regardless of which one is being used).
* The program must have a graphical user interface that allows you to use the functionalities that meet the requirements of the problem.

*(Class diagram is in another document called "Class diagram")*

Pseudocode of the algorithms to be used:

1. BFS

1 method BFS(Graph,origin):

2 we create a queue Q

3 add origin to queue Q

4 mark origin as visited

5 While Q is not empty:

6 we draw an element from queue Q called V

7 for each vertex W adjacent to V in the Graph:

8 if W has not been visited:

9 we mark W as visited

10 insert W into queue Q

2. Dijkstra

1 function Dijkstra(Graph, source):

2

3 for each vertex v in Graph.Vertices:

4 dist[v] ← INFINITY

5 prev[v] ← UNDEFINED

6 add v to Q

7 dist[source] ← 0

8

9 while Q is not empty:

10 u ← vertex in Q with min dist[u]

11 remove u from Q

12

13 for each neighbor v of u still in Q:

14 alt ← dist[u] + Graph.Edges(u, v)

15 if alt < dist[v]:

16 dist[v] ← alt

17 prev[v] ← u

18

19 return dist[], prev[]

**Step 7. Implementation of the Design**

Implementation in a programming language of a pipeline game with graphical interface that allows the player to:

* Configure the graph type:
* Initialize a game with a valid board.
* Place pipes on the board.
* Give up and be shown a valid solution.
* Clear the board.
* Validate his path from the source to the drain and calculate the score according to the chosen path.

**Sources:**

* <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-yprim/#:~: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>.