**Engineering Method**

**Step 1. Problem Identification**

**I Drive**

Description:

A system needs to be designed to implement a game, which in this case is a kind of maze with enemies. This game will allow the user to play against the CPU (enemies). The players will compete in a grid-based game where each participant can only move to their adjacent vertices. Considering that the game needs to be developed using graph implementation, each square on the map where players can move will be modeled as a vertex in a graph.

The story is as follows: It's 1970, and Dani the Red, an expert bank robber, is conducting a routine heist at the National Mint and Stamp Factory in Spain. The heist is going well; Dani has managed to seize 200 million pesetas. The only thing left is to exit the bank and reach one of his hideouts where he will be safe and complete the heist. You are the chosen driver to take Dani to one of his hideouts. You have your DeLorean and your driving skills. You must be very quick.

Game Structure

Based on the fact that this game needs to be made using graph implementation, each square on the board where players can land will be modeled as a vertex.

Graph Components:

50 Vertices (locations): Each of the 50 vertices represents a location in the city.

50 Edges (connections between locations): The 50 edges represent the streets or paths that connect the locations. Each edge connects two vertices and has a weight that represents the travel time.

To avoid a wasted implementation of the graph structure, where the weight of each edge between the vertices is always one, the game will be modeled with different weights between the vertices. This way, it will provide better playability and allow the algorithms we implement to be efficient.

Game Objective

You must reach one of Dani the Red's hideouts in the best possible time so that he can escape the police and you can earn your 20% cut of the heist's takings.

Gameplay

You can move through any street; during a heist, there are no laws or traffic lights. You can only move to your adjacent vertices.

Victory Conditions

Reach one of the hideouts without getting caught by the police.

*Problem Definition*

The problem is to develop a game that includes graphs with a minimum of 50 vertices and 50 edges, and for its solution, it should be possible to apply at least two (2) graph algorithms, in this case, Floyd-Warshall and BFS. The game needs to allow the player to move to different vertices until they reach the hideout (victory condition) or get caught by a police officer (defeat condition), ending the game.

*Identification of Needs and Symptoms*

Representation of the Game Map with Graphs:

Need for an appropriate structure to represent the game map as a graph. Implementation of vertices to represent locations and edges to model the connections between them. This means it must be possible to add vertices and edges to create a graphical representation of the game board.

Minimum Weight Search Algorithms Between Vertices:

Need for minimum weight search algorithms between vertices to model the police and try to reach the robbers in the most efficient way.

Graph Search Algorithms:

Need for graph search algorithms to determine the routes and movements of the players throughout the map.

*Requirements*

| Client | Marlon |
| --- | --- |
| User | Player of the game |
| Functional requirements | RF1 - Show map  RF2-Allow Movement of player.  RF3-Calculate shortest path.  RF4-Allow victory |
| Problem context | A system needs to be designed to implement a game, which in this case is a kind of maze with enemies. This game will allow the user to play against the CPU (enemies). The players will compete in a grid-based game where each participant can only move to their adjacent vertices. Considering that the game needs to be developed using graph implementation, each square on the map where players can move will be modeled as a vertex in a graph. |
| Non-functional requirements | * Quick Response * graphical interface with JavaFX. |

| Identifier and Name | *RF1 - Show map* | | | |
| --- | --- | --- | --- | --- |
| Resumen | The program must be capable of representing the map with its vertices and edges. | | | |
| Inputs | **Name** | **Data type** | | **Valid values condition** |
| Start | button | |  |
| Result or Postcondition | A graph with 50 vertices and their respective edges will be created and shown to the player as a map. | | | |
| Outputs | **Output Name** | | **Data type** | **Format** |
|  | |  |  |

| Identifier and Name | RF2- Allow Movement of players. | | | |
| --- | --- | --- | --- | --- |
| Resumen | *The program allows a player to move from vertex to vertex by clicking on the adjacent vertex they wish to move to.* | | | |
| Inputs | **Name** | **Data type** | | **Valid values condition** |
| click | button | |  |
| Result or Postcondition | The player changes position to the desired vertex. | | | |
| Outputs | **Output Name** | | **Data type** | **Format** |
|  | |  | . |

| Identifier and Name | RF3-Calculate shortest path. | | | |
| --- | --- | --- | --- | --- |
| Resumen | *The system must allow the player to be chased by the police and attempt to catch them to create a defeat condition and add gameplay to the game. For this purpose, the Floyd-Warshall algorithm is used.* | | | |
| Inputs | **Name** | **Data type** | | **Valid values condition** |
|  |  | |  |
| Result or Postcondition | The police will chase the player, and if the player is caught, they will lose the game. | | | |
| Outputs | **Output Name** | | **Data type** | **Format** |
| msg | | String | “you lose” |

| Identifier and Name | RF4-Allow victory | | | |
| --- | --- | --- | --- | --- |
| Resumen | *The system must allow the player to win the game by clicking on the victory vertex if it is adjacent to the vertex where they are currently located.* | | | |
| Inputs | **Name** | **Data type** | | **Valid values condition** |
| click | button | |  |
| Result or Postcondition | The player will win the game upon reaching the victory vertex. | | | |
| Outputs | **Output Name** | | **Data type** | **Format** |
| msg | | String | you win |

## 

## 

## 

## **Step 2. Gathering Information**

*Information about what is required*

*Definitions*

Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices( V ) and a set of edges( E ). The graph is denoted by G(V, E).

Tomado de:

<https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/#what-is-graph-data-structure>

Graphs:

Directed graph: it's one in which the edges have a unique direction. The edges of a directed graph are graphically represented with arrows. Let V be a non-empty finite set, and let E ⊆ V × V be a binary relation. The ordered pair (V, E) is a directed graph over V, or digraph, where V is the set of vertices or nodes, and E is its set of edges. We write G = (V, E) to denote such a digraph.

Taken from:https://posgrados.inaoep.mx/archivos/PosCsComputacionales/Curso\_Propedeutico/Matematicas\_Discretas/Capitulo\_4\_Grafos.pdf

Undirected: When the direction of the edges doesn't matter, the structure G = (V, E), where E is now a set of unordered pairs over V, meaning the set of edges represents a symmetric binary relation, where if Vj and Vk are any vertices in the vertex set V of a graph, (Vj, Vk) ∈ E implies (Vk, Vj) ∈ E. We say that we have an undirected graph.

taken from: https://posgrados.inaoep.mx/archivos/PosCsComputacionales/Curso\_Propedeutico/Matematicas\_Discretas/Capitulo\_4\_Grafos.pdf

Weighted: a graph where each edge is associated with a value to represent a cost.

Traversals:

Breadth First Search (BFS) is a graph traversal algorithm that explores all the vertices in a graph at the current depth before moving on to the vertices at the next depth level. It starts at a specified vertex and visits all its neighbors before moving on to the next level of neighbors. BFS is commonly used in algorithms for pathfinding, connected components, and shortest path problems in graphs.

The time complexity of BFS and DFS is O(V+E) because they need to visit and examine each vertex and edge of the graph. This makes them linear algorithms, which are generally efficient for traversing graphs.

Taken from:<https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/>

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

taken from: https://www.encora.com/es/blog/dfs-vs-bfs

**The Floyd-Warshall algorithm:**is a fundamental algorithm in computer science and graph theory. It is used to find the shortest paths between all pairs of nodes in a weighted graph. This algorithm is highly efficient and can handle graphs with both positive and negative edge weights, making it a versatile tool for solving a wide range of network and connectivity problems..

tomado de: <https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/>

complexity

The Floyd-Warshall algorithm runs in O(∣V∣^3 ) time. This is because of the three nested for loops that are run after the initialization and population of the distance matrix, M.

taken from: <https://brilliant.org/wiki/floyd-warshall-algorithm/>

Dijkstra's algorithm is a method for determining the shortest path from a given origin vertex to all other vertices in a graph with respect to the weights on each edge. The idea is to explore all the shortest paths starting from the origin vertex and leading to all other vertices. When the shortest path from the origin vertex to all other vertices in the graph is obtained, the algorithm stops.

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

Complexity Dijkstra

Matrix

The given graph G = (V, E) is represented as an adjacency matrix. Here, w[u,v] stores the weight of the edge (u,v).

The priority queue Q is represented as an unordered list.

Let |E| and |V| be the number of edges and vertices of the graph, respectively. Then the time complexity is calculated as follows:

- Adding all |V| vertices to Q takes O(|V|) time.

- Removing the node with the minimum distance takes O(|V|) time, and we only need O(1) to recalculate and update its distance dist[u] and reorder Q. Since we use an adjacency matrix here, we need to loop through |V| vertices to update the distance matrix.

- The time needed for each iteration of the loop is O(|V|), as one vertex is removed from Q per loop.

- Therefore, the total time complexity becomes O(|V|) + O(|V|) × O(|V|) = O(|V|^2).

Using a List:

The given graph G = (V, E) is represented as an adjacency list.

The priority queue Q is represented as a binary heap or a Fibonacci heap.

- It takes O(|V|) time to construct the initial priority queue of |V| vertices.

- With the adjacency list representation, all vertices of the graph can be traversed using BFS. Therefore, iterating over the neighbors of all vertices and updating their distance values during the algorithm execution takes O(|E|) time.

- The time needed for each iteration of the loop is O(|V|), as one vertex is removed from Q per loop.

- The binary heap data structure allows extracting the minimum (removing the node with the minimum distance) and updating an element (recomputing dist[u]) in O(log|V|) time.

- Therefore, the time complexity becomes O(|V|) + O(|E| log|V|) + O(|V| log|V|), which is O((|E|+|V|) log|V|) = O(|E| log|V|), as |E| >= |V| - 1 when G is a connected graph.

Taken from: <https://www.baeldung.com/cs/dijkstra-time-complexity>

**Prim**

This algorithm always starts with a single node and moves through various adjacent nodes to explore all edges connected along the way.

The algorithm begins with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, and the other set contains the vertices not yet included. At each step, it considers all edges that connect the two sets and selects the minimum-weight edge from these edges. After selecting the edge, it moves the other endpoint of the edge to the set containing the MST.

Time Complexity: O(E\*log(E)) where E is the number of edges

Taken from: <https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/>

**Kruskal**

In Kruskal's algorithm, all edges of the given graph are sorted in increasing order. Then, it continues to add new edges and nodes to the MST if the newly added edge does not form a cycle. It selects the minimum-weight edge at the beginning and the maximum-weight edge at the end. Therefore, we can say that it makes a locally optimal choice at each step to find the optimal solution. Thus, this is a greedy algorithm.

Time Complexity: O(E \* logE) or O(E \* logV)

Sorting the edges takes O(E \* logE) time.

After sorting, we iterate through all the edges and apply the union-find algorithm. The find and union operations can take up to O(logV) time at most.

So, the overall complexity is O(E \* logE + E \* logV) time.

The value of E can be at most O(V^2), so O(logV) and O(logE) are equal. Therefore, the overall time complexity is O(E \* logE) or O(E \* logV).

Taken from:<https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/>

**Step 3. Creative Solutions Search**

We will not consider the prim and kruskal algorithms since in this case we do not require their use, so options with Floyd Warshall and Dijkstra will be considered.

**Alternative 1: Floyd-Warshall, Graph, BFS, Queue**

**Floyd-Warshall:**

Calculates the shortest paths between all pairs of vertices in a weighted graph.

Allows police to determine the fastest way to reach the thief, adapting to changes in the thief's position.

**Graph:**

Represents the game map with vertices (locations) and edges (paths). Facilitates visualization and route calculation in the game.

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

Searches for routes and explores the graph from a starting node. Useful for the player to model the map and quickly determine accessible routes.

**Queue:**

Implements BFS to explore the graph level by level. Allows for an ordered and systematic exploration of the graph.

**Alternative 2: Dijkstra, Graph, DFS, HashMap, Stack, Queue, ArrayList**

**Dijkstra:**

Finds the shortest routes from a source node in a weighted graph. Useful for police to determine the shortest route to the thief in real-time.

**Graph:**

Represents the game map with vertices and edges. Provides a clear and manipulable structure of the game map.

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

Explores the graph to find possible escape routes. Allows the player to thoroughly explore the map and find long or complex paths.

**HashMap:**

Stores and quickly accesses information about vertices and edges.Improves the performance of search algorithms and route calculations.

**Stack:**

Stores the traversal during DFS execution.Facilitates handling of backtracking in route search.

**ArrayList:**

Stores adjacency lists for vertices and calculated routes.

Provides a flexible and dynamic structure for handling graph data.

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

**Alternative 1:**

**Floyd Warshall, Graph, BFS, Queue**

**Floyd-Warshall:**

The Floyd-Warshall algorithm is used to calculate the shortest paths between all pairs of vertices in a weighted graph. In the context of the game, this would allow police to efficiently determine the fastest way to reach the thief, reacting to changes in the thief's position and new connections in the graph.

**Graph:**

A graph structure is used to represent the game map, where vertices represent locations in the city and edges represent streets or paths between them. This representation provides a clear visualization of the game map and facilitates route calculation.

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

BFS is used to search for routes and explore the graph from a starting node. This would be useful for the player, as it allows modeling of the game map and quickly determining accessible routes from their current location. Additionally, BFS is efficient for graph exploration, making it useful for navigation in the game.

**Queue:**

Implements BFS to explore the graph level by level. It allows for an ordered and systematic exploration of the graph.

**Alternative 2:**

**Dijkstra, Graph, DFS, HashMap, Stack, Queue, ArrayList**

**Dijkstra:**

Finds the shortest routes from a source node in a weighted graph. This would be useful for police to determine the shortest route from their current location to the thief in real-time.

**Graph:**

Represents the game map with vertices and edges. Provides a clear and manipulable structure of the game map.

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

Explores the graph to find possible escape routes. Allows the player to thoroughly explore the map, ensuring all possible routes are considered and facilitating the search for long or complex paths.

**HashMap:**

Stores and accesses information about vertices and edges quickly, such as edge weights and connections between vertices. Provides efficient access to graph data, improving the performance of search algorithms and route calculations, especially when non-sequential access to information is required.

**Stack:**

Stores the traversal during the execution of DFS. Facilitates the implementation of DFS and the handling of backtracking in route search, allowing the storage of visited nodes.

**ArrayList:**

Stores adjacency lists for vertices, as well as calculated routes and possible paths. Provides a flexible and dynamic structure for handling graph data.

It's worth noting that the best option might be to use Floyd Warshall because the police need to update their position every time the player moves one cell, so if we were to use Dijkstra, it would require an n^2 traversal each time, whereas with Floyd, we have the minimum distance between any pair of vertices from the start.

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

Criteria 1: Efficiency

[4] Very efficient

[3] Efficient

[2] Moderately efficient

[1] Not very efficient

Criteria 2: Flexibility and Scalability

[4] Very flexible and scalable

[3] Flexible and scalable

[2] Moderately flexible and scalable

[1] Not very flexible and scalable

Criteria 3: Ease of Implementation

[4] Very easy to implement

[3] Easy to implement

[2] Moderately easy to implement

[1] Difficult to implement

Criteria 4: Solution Accuracy

[2] Fits perfectly to the solution

[1] Approximate solution

*Evaluation*

|  | Criteria 1 | Criterio 2 | Criterio 3 | Criterio 4 | Total |
| --- | --- | --- | --- | --- | --- |
| Alternativa 1 | 3 | 3 | 3 | 2 | 11 |
| Alternativa2 | 3 | 3 | 2 | 1 | 9 |

*Selection*

In conclusion,alternative 1 is the best option.