MEMBERS:
- Maria Alejandra Mantilla Coral - A00395792
- Rafaela Sofía Ruiz Pizarro - A00395368
- Andrés David Parra García - A00395676

Engineering method

1. Problem identification

Needs and symptoms:

1. Need: The gaming company requires a program to find the shortest path in a maze.
   Symptoms: The company wants to develop a game where the player needs to navigate through a maze from an initial position to a final position. They need a program that can calculate the shortest path between these two positions.
2. Need: The program should handle mazes of varying sizes and configurations.
   Symptoms: The company wants the program to be flexible in terms of maze size, which is represented by a matrix. The matrix can have different numbers of rows and columns, allowing for mazes of various shapes and sizes.
3. Need: The program should accurately determine if a path exists between the start and end positions.
   Symptoms: The program should be able to identify cases where there is no possible path from the initial position to the final position. In such cases, the program should output "-1" to indicate the absence of a valid path.
4. Need: The program should output the minimum number of moves required to reach the end position.
   Symptoms: The gaming company wants to provide players with information about the shortest path. The program should calculate and output the minimum number of moves required to navigate from the initial position to the final position in the maze.
5. Need: The program should handle obstacles and varying weights in the maze.
   Symptoms: The maze may contain obstacles or cells with different weights, representing the difficulty or cost of traversing those cells. The program should account for these obstacles and weights while calculating the shortest path.

Problem definition:

The gaming company requires software that tells the player:

- The entrance that leads to pass the maze.
- The shortest way to pass throughout the maze (if there's a way).
- The number of minimum movements to pass the maze.

2. Collection of Information

Information to load the graph:

- The maze is represented by a matrix of size NxM.
- Each cell in the matrix represents the weight or cost of passing through that cell.

- Negative values indicate cells that cannot be passed through.
- Multiple starting positions are provided.

Algorithms needed:

- Dijkstra's algorithm: Is a popular algorithm used to find the shortest path between two nodes in a weighted graph. It guarantees the shortest path when all edge weights are non-negative. The algorithm maintains a priority queue of vertices, starting from the source node, and iteratively explores the vertices with the smallest tentative distance. It gradually builds the shortest path tree from the source to all other reachable nodes in the graph.
- BFS algorithm: BFS (Breadth-First Search) algorithm is a simple graph traversal algorithm that explores all the vertices of a graph in breadth-first order, starting from a given source vertex. It is often used to find the shortest path in an unweighted graph or to explore the graph level by level. BFS visits all the vertices at the same level before moving to the next level.

3. Search for creative solutions

Solution to the problem of entrance:

Alternative Solution 1: DFS algorithm

The Breadth-First Search (BFS) algorithm is used to determine if there is a path from each starting position to the final position. Additionally, if a path exists, it can calculate the cost of each path using Dijkstra's algorithm. The paths and their costs can be stored in a list or priority queue, allowing the selection of the path with the minimum cost as the solution.

Alternative Solution 2: BFS algorithm

Use BFS algorithm to determine if there is a path from each starting position to the final position. If a path exists, calculate the cost of each path using Dijkstra's algorithm. Store the paths and their costs in a list or priority queue. Select the path with the minimum cost as the solution.

Alternative Solution 3: Flood Fill Algorithm

The flood fill algorithm can be used to determine if there is a path from each starting position to the final one in the maze. It explores the maze by recursively filling the reachable cells from each starting position. Here's an outline of the approach:

1. Create a visited matrix to keep track of visited cells.
2. For each starting position:
    - Perform a flood fill from the starting position to mark all reachable cells.
    - Mark the starting position as visited.
3. Check if the final position is marked as visited for any of the starting positions. If yes, there is a path from at least one starting position to the final position.
4. If the final position is reachable, you can also retrieve the path by backtracking from the final position using the visited matrix.

Solution to the problem of minimum path:

Alternative Solution 1: Dijkstra algorithm

Use Dijkstra's algorithm to find the shortest path from each starting position to the final position. Store the paths and their costs in a list or priority queue. Select the path with the minimum cost as the solution.

Alternative Solution 2: Dynamic programming

Implement a dynamic programming approach using a matrix to store the minimum costs to reach each cell. Iterate through the matrix to calculate the minimum costs to reach the final position from each starting position. Store the paths and their costs in a list or priority queue. Select the path with the minimum cost as the solution.

Alternative Solution 2: Recursive Backtracking

Another alternative solution is to use a recursive backtracking algorithm to explore the maze and find the shortest path. Here's an outline of the approach:

1. Create a recursive function to explore the maze:
    - Accept the current position, destination position, and path.
    - Base case: If the current position is the destination, return the path.
    - Recursive case:
        - Mark the current position as visited.
        - Get the neighboring cells of the current position.
        - For each unvisited neighboring cell:
            - Add the current cell to the path.
            - Recursively call the function with the new position and updated path.
            - If a valid path is returned, return it.
        - Backtrack by removing the current cell from the path and marking it as unvisited.
2. Start the exploration by calling the recursive function with the initial position, destination position, and an empty path.
3. If a valid path is returned, it represents the shortest path. Otherwise, there is no valid path.

4. **Transition from ideas to main designs - Review of ideas:**

Problem A - For determining if there is a path from each starting position to the final one:

Alternative 1: Depth-First Search (DFS):

- Benefits:
  - Relatively easy to implement.
  - Memory-efficient as it does not require storing the entire graph.
- Limitations:
  - May not always find the shortest path.
  - Requires marking visited cells to avoid infinite loops.

Alternative 2: Breadth-First Search (BFS):

- Benefits:
  - Guarantees finding the shortest path in an unweighted graph.
  - Suitable for exploring the graph level by level.
- Limitations:
  - May not be as efficient for weighted graphs, although it can be modified with additional complexity.

Alternative 3: Flood Fill Algorithm:

- Benefits:
  - Can handle obstacles and varying weights in the graph.
  - Provides a way to fill the reachable cells for further analysis.
- Limitations:
  - Requires marking visited cells and checking for obstacles.
  - Backtracking may be required to find the shortest path.

Problem B - For determining the minimum path in the graph maze:

Alternative Solution 1: Dijkstra algorithm:

- Benefits:
  - Offers a high-performance solution with efficient path finding.
  - Provides accurate and correct results for determining the minimum path.
- Limitations:
  - Requires the graph to have non-negative edge weights.
  - May not be suitable for graphs with many vertices or complex structures.

Alternative Solution 2: Dynamic programming:

- Benefits:
  - Provides an efficient and accurate solution for determining the minimum path.
  - Can handle graphs with varying sizes and configurations.
- Limitations:
  - Requires additional memory to store the matrix of minimum costs.
  - May have a higher time complexity for larger graphs.

Alternative Solution 3: Recursive Backtracking:

- Benefits:
  - Offers a simpler implementation compared to other algorithms.
  - Can handle graphs with obstacles and varying weights.
- Limitations:
  - May have a higher time complexity for large or complex mazes.
  - May not guarantee the most efficient path in all cases.}

5. **Evaluation and selection of solutions**

Criteria A: Performance and efficiency:

[4] High performance and efficiency.
[3] Good performance and efficiency.
[2] Moderate performance and efficiency.
[1] Low performance and efficiency.

Criteria B: Accuracy and correctness:

[4] Highly accurate and correct.
[3] Mostly accurate and correct.
[2] Moderately accurate and correct.
[1] Less accurate and correct.

Criteria C: Flexibility and adaptability:

[4] Highly flexible and adaptable to different scenarios.
[3] Moderately flexible and adaptable to some scenarios.
[2] Limited flexibility and adaptability.
[1] Not flexible and adaptable.

Problem	Alternative	Criteria A 40%	Criteria B 30%	Criteria C 30%	Total
Problem A	Alternative 1: DFS	[2]	[2]	[3]	2.3
	Alternative 2: BFS	[3]	[3]	[3]	3.0
	Alternative 3: Flood Fill Algorithm	[2]	[3]	[3]	2.6
Problem B	Alternative 1: Dijkstra algorithm	[4]	[4]	[2]	3.4
	Alternative 2: Dynamic programming	[3]	[4]	[3]	3.3
	Alternative 3: Recursive Backtracking	[2]	[3]	[3]	2.6

Based on the scores for each alternative, the best solutions for determining the minimum path from each starting position to the final position in the graph maze would be:

Problem A: Alternative 2: BFS algorithm

Problem B: Alternative 1: Dijkstra algorithm