engineering method

Members:
-Pablo Pineda - A00395831
-Daniela Londoño - A00392917
-Isabella Huila - A00394751

Phase 1: Identification of the problem

The problem is to develop a system that can generate the legendary Cretan Labyrinth and solve it using suitable search algorithms. The system must have a starting point and an ending point within the maze so that the agent can start his journey. The system should show the path where it is passing.

customer	video game company
user	People interacting with the game
Functional requirements	**R1**: Generate the maze **R2:**Find the way with the BFS algorithm **R3:**Finding the way with the DFS algorithm **R4:**Show the possible paths inside the maze
Problem context	● The problem is to develop a system that can generate the legendary Cretan Labyrinth and solve it using suitable search algorithms. The system must have a starting point and an ending point within the maze so that the agent can start his journey. The system should show the path where it is passing.
non-functional requirements	R1: The system needs to be efficient using the minimum resources necessary R2: The system needs to be functional in such a way that it fulfills the intended functions

<table>
<tr><td>Name or identifier</td><td colspan="3">R1: Generate the Maze</td></tr>
<tr><td>Summarizes</td><td colspan="3">The system must generate a maze with corridors and walls, which will have a starting point and an end point. Within the labyrinth there will be several paths that will reach the end point.</td></tr>
<tr><td rowspan="2">Inputs</td><td>Input name</td><td>datatype</td><td>Selection or repetition condition</td></tr>
<tr><td>Text file</td><td>txt</td><td></td></tr>
<tr><td>General activities necessary to obtain the results</td><td colspan="3">1. The system must generate a graph which represents the maze
2. The system must define the start point and end point
3. The system must display the interface where the maze will be</td></tr>
<tr><td>Result or postcondition</td><td colspan="3">1. The labyrinth (Graph) generated. (Upload txt file)</td></tr>
<tr><td rowspan="2">Outputs</td><td>Output name</td><td>datatype</td><td>Selection or repetition condition</td></tr>
<tr><td>Labyrinth</td><td>Graphic</td><td>There is some error and the txt file cannot be loaded</td></tr>
</table>

Name or identifier	**R2:**: Find the way with the BFS algorithm		
Summarizes	The system allows using this method to go through all the nodes of the maze and look for the possible exit.		
Inputs	Input name	datatype	Selection or repetition condition
	startpoint	int	
	final point	int	
General activities necessary to obtain the results	1. Going through the maze with the BFS method 2. Get the way to the exit		
Result or postcondition	1. Path index list		
Outputs	Output name	datatype	Selection or repetition condition
	wayout	list	

Name or identifier	**R3**: Find the way with the DFS algorithm		
Summarizes	The system allows using this method to go through all the nodes of the maze and look for the possible exit.		
Inputs	Input name	datatype	Selection or repetition condition
	startpoint	int	
	final point	int	
General activities necessary to obtain the results	1. Walking the maze with the DFS method 2. Get the way to the exit		
Result or postcondition	1. Path index list		
Outputs	Output name	datatype	Selection or repetition condition
	wayout	list	

Name or identifier	**R4**: Show the possible path inside the maze		
Summarizes	The system should allow you to find the path where you went to find the exit, either with the DFS and BFS. This should be displayed in an interface so that the solution path for the maze is marked with a color.		
Inputs			
General activities necessary to obtain the results	1. Get the path traveled 2. Show the way and the solution to the maze.		
Result or postcondition	1. The solution path of the maze		
Outputs	Output name	datatype	Selection or repetition condition
	Path	Animation	

Phase 2: Information Gathering

To solve this problem, we will address the concepts and programming tools that will allow us to generate the maze and go through it. Next, we will address these for the implementation of the system:

Graphic Schema Theory:

It is a branch of mathematics and computer science that studies the properties of graphs. In addition, it analyzes and solves problems related to graphs.

graph:

A graph G = (V, E) is a mathematical structure consisting of two sets V and E. The elements of V are called vertices (or nodes), and the elements of E are called edges. Each edge is associated with a set of one or two vertices, which are called its endpoints.
terminology: It is said that an edge joins its endpoints. A vertex joined by an edge to a vertex V, The vertex v is said to be a neighbor of v.
self loop:he is an artist who unites a single final point to himself.
multi edge:is a collection of two or more edges that have identical endpoints. The multiplicity of edges is the number of edges within the multi-edge.
simple graph:it has no proper loops or multiple edges.
Non-looping graph (or multi-graph):can have multiple edges but not auto-loops.
Graphic (general):can have automatic loops and/or multiple edges.

Tour algorithms:
They are techniques used to visit all the nodes or edges of a graph. These algorithms allow you to explore and search for information within the structure of the graph.

Breadth-first search: Breadth search is one of the simplest algorithms for finding a graph, and the archetype of many important graph algorithms. Breadth-wise search is so named because it expands the boundary between discovered and undiscovered vertices uniformly across the width of the boundary. You can think of it as discovering vertices in waves emanating from the source vertex. That is, starting with s, the algorithm first discovers all the neighbors of s, which have distance 1. It then discovers all vertices with distance 2, then all vertices with distance 3, and so on, until it has discovered all vertices reachable from s

Depth First Search: As its name implies, depth-first search searches < deeper into the graph whenever possible. The depth search explores the edges outside of the most recently discovered v vertex that still has unexplored edges leaving it. Once all the edges of v have been explored, the search <backtracks= to explore the edges leaving the vertex from which v was discovered. This process continues until all vertices reachable from the original source vertex have been discovered. If there are any undiscovered vertices left, the depth-first search selects one of them as a new source, repeating the search from that source. . The algorithm repeats this entire process until it has discovered each vertex.

Shortest path algorithms:
It is the problem that consists of finding a way between two vertices or nodes, in such a way that the sum of the weights of the edges that constitute it is minimal. The shortest path between two vertices is also known as a geodesic.

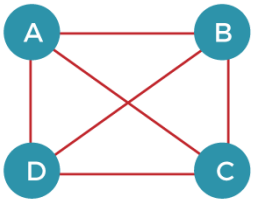
The Floyd-Warshall algorithm: Floyd's algorithm is very similar, but it works with weighted graphs. That is, the value of the "arrow" that we represent in the matrix can be any real or infinite number. Infinity marks that there is no union between the nodes. This time, the result will be a matrix where the minimum distances between nodes will be represented, selecting the most convenient paths according to their weighting (“weight”).

Dijkstra's algorithm:

Dijkstra's algorithm is used in graph theory to find the shortest path between a source node and all other nodes within the weighted graph. The main functionality of the algorithm is to

determine the shortest path from a given source node to all other nodes of an undirected weighted graph. Although it can also be used in a directed graph, it is not really that common to find it. Dijkstra's algorithm does not admit edges with negative weights, however, it works efficiently on graphs with non-negative weights and is especially useful when trying to find the shortest route in communication networks or navigation systems, as this is the case. It would be in the maze.

Minimum Spanning Trees:



The graph above can be represented as G(V, E), where 'V' is the number of vertices and 'E' is the number of edges. The spanning tree in the graph above would be represented as G`(V`, E`). In this case, V` = V means that the number of vertices in the spanning tree would be the same as the number of vertices in the graph, but the number of edges would be different. The number of edges in the spanning tree is the subset of the number of edges in the original graph.

Kruskal's algorithm:

Kruskal's algorithm is an algorithm used in graph theory to find a minimum spanning tree in an undirected weighted graph. It is widely used in network design and optimization problems.
The main functionality of Kruskal's algorithm is to find a minimum spanning tree in a weighted graph. A minimal spanning tree is a subset of the edges of the graph that connects all the nodes and has the smallest possible total weight. This algorithm works as follows.
Kruskal's algorithm works as follows: It orders all the edges of the graph by weight, from smallest to largest, then initializes an empty tree as the result.
As a next step, for each edge in ascending order of weight, if adding it to the tree does not form a cycle, it is added to the resulting tree. Repeat step 3 until all edges have been considered or a minimal spanning tree has been constructed.

Prim's algorithm:

The algorithm is used to find the minimum spanning tree in an undirected weighted graph. The goal of the algorithm is to build a subset of edges that connect all the vertices of the graph by minimizing the sum of the weights of the selected artists. Prim's algorithm works as follows.

First it selects an arbitrary initial vertex and adds it to the minimum spanning tree, then at each iteration an edge of minimum weight is chosen that connects a vertex in the tree with a vertex outside the tree; It then adds the selected edge to the minimum spanning tree and marks the added vertex as visited. The previous two steps are repeated until all vertices are in the minimum spanning tree. Finally, the resulting minimum expiation tree contains all the edges necessary to connect all the vertices of the original graph, minimizing the total cost. In short, this algorithm is based on the idea of growing the tree incrementally, always selecting the edge with the least weight that connects a vertex in the tree with a vertex outside the tree.

Taken from:

- **Graph Theory and Its Applications, third edition - DOKUMEN.PUB**
- **https://dl.ebooksworld.ir/books/Introduction.to.Algorithms.4th.Leiserson.Stein.Rivest.Cormen.MIT.Press.9780262046305.EBooksWorld.ir.pdf**
- **https://www.javatpoint.com/minimum-spanning-tree-introduction**

Phase 3: Search for Creative Solutions

The generation of the maze is going to be done by us, therefore you will not see a specific algorithm to generate it randomly, but we only add a txt file with the graph and within it a matrix that represents the maze.

For the search for the path of the maze, we have the following alternatives:

Alternative 1: Search by BFS
The breadth-first search algorithm is a common approach to solving mazes. We can implement it to find the shortest path from the start point to the end point in the maze. The BFS algorithm will explore the maze level by level, making sure that the shortest solution is found. We can implement it as follows:

- Node queue: The BFS algorithm uses a data structure called a "queue" to store the nodes to be explored. Each node represents a cell in the maze. Start at the node corresponding to the start point and add it to the queue.

- Node expansion: As long as the queue is not empty, remove the front node from the queue and examine its neighbors. In the maze context, the neighbors are the adjacent cells (top, bottom, left, right) that are not walls. If the endpoint is found, you have found the solution and we can stop the search.

- Mark and Queue: As you examine a node's neighbors, mark each neighbor as visited and add it to the queue if it hasn't been visited before. This will ensure that the algorithm explores the maze in width, level by level.

- Shortest path: During node expansion, we can keep a record of the parent nodes for each visited node. This allows us to reconstruct the shortest path from the start point to the end point once you find it.

- Visualization: While the BFS algorithm is in progress, the user can visualize the progress of the agent as it explores the maze. It will display the maze in a graphical interface and highlight the cells visited and the path currently explored.

Alternative 2: Search by DFS

Also the depth search algorithm is suitable for exploring mazes and finding solutions. It can be used for the agent to explore the maze. Where the algorithm for the agent to advance through the corridors and back when reaching a dead end. That is, what this algorithm does is search by depth. Also, like the BFS algorithm, it visually displays the agent's progress in the maze.

- This algorithm will go looking for the vertices, when it has reached a vertex, what it does is look for its neighbor, this process will follow it until it meets a wall, if it does, it returns the vertex that has been completely visited and will continue with another has not been visited. When you find the end point, the solution is found and the search stops.

- Mark vertex: As you examine the neighbors of a vertex , mark each neighbor as visited. As we said earlier, this will ensure that the algorithm explores the maze thoroughly, going as far along a path as possible before going back.

Alternative 3: Search by The Floyd-Warshall algorithm

Another way to solve the maze is to implement the Floyd-Warshall algorithm, where the shortest paths between all pairs of nodes in a weighted graph are found. Although its main application is in solving shortest path problems in graphs. It could be solved as follows:

- Construction of the graph: Using the representation of the maze, a weighted graph will be constructed where the vertices represent the cells of the maze and the edges have an associated weight. The edges will represent the connections between the cells of the maze and their weight will depend on whether there is a corridor or a wall between them. For example, you would assign a low weight to the corridors and a high weight to the walls.

- Application of the Floyd-Warshall algorithm: This algorithm will find the shortest paths between all pairs of vertices in the graph. After running the algorithm, a distance matrix indicating the length of the shortest path between each pair of vertices will be obtained.

- Search for the route between the start and end points: Using the distance matrix obtained from the previous step, the shortest route between the start and end points would be determined.

Alternative 4: Search by Dijkstra's algorithm
Another solution to our problem, would be to use this algorithm, where the shortest path will be found from the initial point where that will be the origin vertex to all the other vertices in a weighted graph, until reaching the end point.

- Construction of the graph: Using the representation of the maze, a weighted graph will be constructed where the vertices represent the cells of the maze and the edges have an associated weight. The edges will represent the connections between the cells of the maze and their weight will depend on whether there is a corridor or a wall between them. A low weight will be assigned to the corridors and a high weight to the walls.

- Application of Dijkstra's algorithm: Applies Dijkstra's algorithm to the constructed graph, taking the initial point of the maze as the origin vertex. Dijkstra's algorithm will calculate the shortest path from the starting point to all other vertices in the graph. During the execution of the algorithm, a record of the parent vertices will be kept for each node visited.

- Path search between start and end points: Uses the parent vertices recorded during the execution of Dijkstra's algorithm to determine the shortest path from the start point to the end point. It starts at the end point and follows the parent vertices until it reaches the start point, building the path in reverse.

The solution is similar to the previous one, only that the first one is applied with respect to a matrix while Dijkstra's algorithm is on an adjacency list.

Alternative 5: Iterative Deepening Depth-First Search (IDDFS) algorithm
Investigating a little more we found this algorithm that is also search that is a variant of the depth search algorithm (DFS) that avoids some of the problems associated with DFS, such as the possibility of falling into infinite loops in graphs with cycles.
The IDDFS uses an iterative strategy to perform a depth search with increasing maximum depth at each iteration. It starts with a depth limit of 0 and increases each iteration until the target is found or the entire search space is explored.

- IDDFS Algorithm: Implements the Depth Limited Search (IDDFS) algorithm to solve the maze. Initializes a depth limit to 0 and sets a flag to indicate if the target has been found. Make a main loop to iterate over the different limit depths.
- At each iteration: The algorithm performs a limited depth search within the maze using the depth search algorithm. During that search, it keeps track of the Vertices visited and the route followed so far. If the endpoint is found, it sets the target found flag to true and stores the route as solution. Increments the depth limit by 1 on each iteration until the target is found or the entire search space has been explored.

- Results: If the objective is found, it shows the route found to the user as a solution. If the objective is not found, it shows the possible paths or a message indicating that the mission could not be completed. This solution using the IDDFS algorithm will allow the Maze to be generated of Crete and solve it using a limited-depth search. The algorithm will gradually explore the maze for the solution, avoiding problems like infinite loops and using a reasonable amount of memory.

Phase 4: Transition from ideation to preliminary designs.

Since our main objective is only to find the ways out of the maze by comparing the algorithms to see which is the most efficient, we will not include weights on the edges. Therefore we rule out alternatives 3 and 4 ; since these only work with weights. Eventually, alternative 5 could be useful for us, but it is an algorithm that we have not yet seen in class, therefore it would be a separate investigation that could be carried out, but it would take a little time, so we will post the reasons why we should not use it, but we will pass it on. to the next phase for further evaluation, for which these are our reasons:

1. The Floyd-Warshall and Dijkstra algorithms are designed to solve problems in weighted graphs, where the edges have associated weights. However, if we are not considering maze edge weights, applying these algorithms may be unnecessary and add additional complexity to the solution.
2. Both algorithms are more complex in terms of execution time and resource usage compared to other simpler approaches such as Breadth-Finding Search (BFS) or Depth-Finding Search (DFS).
3. The IDDFS has a high time complexity, as it performs multiple depth-limited search iterations with gradual increments in the depth limit. This can lead to a long execution time, if we were to write code which would generate random mazes for especially large mazes. Because the IDDFS performs multiple search iterations in limited depth, it requires significant use of resources, such as memory and processing power. This can be problematic in resource-constrained environments or when working with very large mazes.
   Although the IDDFS guarantees to find a solution if it exists within the depth limit, it does not guarantee to find the shortest or optimal solution. You might find a solution, but there might be a shorter path that wasn't explored due to depth limit limitations.

Depending on the objective, it may be more efficient to use a simpler approach such as BFS and DFS instead of the alternatives we are ruling out.

Phase 5: Evaluation of the best solution.

According to our approach, the idea is to find the way out of the maze by comparing the efficiency of the two algorithms, which find a way only in different ways.
The reasons why we chose these two algorithms are the following:

- Finding a feasible path: For the Cretan Maze without edge weights, the main goal is to find a feasible path from the start point to the end point. BFS and DFS are designed to find a valid path in a graph without considering the edge weights. These

simple approaches can ensure that a path is found if it exists, regardless of the length of the path or whether it is the shortest path.

- Execution time: Both BFS and DFS have a linear execution time in relation to the number of cells in the maze. This means that as the maze gets larger, the time required to find a route from the start point to the end point increases in a predictable way. By comparison, algorithms such as Floyd-Warshall and Dijkstra have higher time complexity, which can result in longer execution time when applied to large mazes.

- Resource usage: BFS and DFS are algorithms that do not require as much additional storage space. While Floyd-Warshall and Dijkstra need additional data structures such as arrays and lists to store information about distances and nodes visited, BFS and DFS only require a basic data structure to store maze path information.

- Implementation and simplicity: BFS and DFS are simpler algorithms to implement and understand compared to Floyd-Warshall and Dijkstra. They do not require additional concepts such as distance matrices or shortest path weight calculations.

Therefore, to solve our problem, we will use the simpler approaches like BFS and DFS so that the Cretan Maze without edge weights can be more efficient in terms of execution time and resource usage and would meet our goal.

However, we are still evaluating a possible option, which is the Limited Depth Search algorithm (IDDFS), for the following reasons:

- Guaranteed completion: The IDDFS guarantees to find a solution if it exists within the established depth limit. This means that as the depth limit is gradually increased each iteration, a solution will eventually be found if it exists in the maze. This offers the assurance that a solution will be found, which is especially important in cases where the existence of a solution is crucial.
- Deep Scan: The IDDFS follows a deep scan strategy, which means that it goes as far into a branch of the maze as possible before backing out. This can help us in mazes where the solutions lie in deeper branches, since these branches will be explored first before going back and exploring other options.
- Simple implementation: IDDFS is relatively easy to implement compared to other more complex algorithms. It uses the basic data structures such as stacks or recursion making it easy to implement and understand.