**Engineering Method Integrative Task 2**

**Carol Mosquera (A00403934 )**

**Samuel Navia Quiceno (A00405006)**

**Simón García (A00371828)**

## Phase 1: Problem identification

**Description:**

A game must be developed that can be modeled using graphs, with a minimum of 50 vertices and 50 edges and that applies at least two (2) of the graph algorithms that will be studied during the course, In this case we are going to implement the BFS and Djistra methods, which will be implemented as the structure of the game. The game will consist of a map made up of several paths, each of which will have different events, and depending on the options the player chooses, he will advance until he reaches the end or runs out of lives.

**Causes:**

One of the objectives of the game is to implement a structure that handles graph logic, so it is necessary to develop a game that uses these structures within its functionalities, in which case it will be necessary to define systems that contain vectors and edges elements. In our case, we implement the vectors as the different events that can appear in the game, and the roads that connect each event will be the edges that will have different values that represent the difficulty depending on the type of event in its extremes.

**Symptoms:**

At the time of implementing a game, it is necessary to think of a number of qualities or requirements that should have the game, like the gameplay, the story, the environment, the possibilities the player has and their consequences, the rewards and penalties for doing something right or wrong, so these are the qualities that define a game. In our game, the gameplay will be more static because the movement will only depend on a series of buttons that represent the options that the player can make, like which path to choose or what to do in a specific event. In other aspects, the story of the game is that the player is lost in a forest and its objective it get out of there alive, so the environment of th gameplay its the map of the forest in which you can see the differents paths, so if the player choose a god path the rewards can be obtain more health or objects that can make the journey more easy, but if the player choose a bas path the consequences is loose health, if the player loose all the health before reach the eand, is game over.

**Problem definition:**

One of the problems or challenges that the game has is to define which are the easiest or most difficult routes by the minimum distance of the values of each edge that depends on the difficulty, so it is necessary to implement methods of traversing a graph that define the easiest path (the path with the smallest difficulty) and the hardest path (the path with the biggest difficulty).

**Requirement Specifications:**

| **Client:** | Jeison Mejía |
| --- | --- |
| **Users:** | * Product owner * Testers * Users |
| **Functional requirements:** | * Req#1: Generate a Map * Req#2: Move through the map * Req#3: Win the game * Req#4: Loose the game * Req#5: Assign randomly the type of stations around the map * Req#6: Mark the hardest and easiest way through the map |
| **Context of the problem:** | * Create a game with different roads and which road have an event, although each of the paths is different, their objective is to get from a starting point to an end point. |
| **Nonfunctional requirements:** | * RNF#1:connectivity * RNF#2:efficiency |

| **Identifier and Name:** | Req#1 Generate a map | | |
| --- | --- | --- | --- |
| **Summary:** | The objective of this requirement is to generate a map as soon as you click on the button called startButtom in this way the program is triggered and displays a map. | | |
| **Inputs** | **Input name** | **Data Type** | **Conditions for valid values** |
| startButtom | Event | Do a click on the section of the button |
| **Results or postcondition** | The system receives an event which it processes and subsequently generates the game map. | | |
| **Outputs** | **Output Name** | **Data type** | **Format** |
| ScreenA | Screen | Canvas |
| Error | Message | A message is printed that there was an error while generating the map. |

| **Identifier and Name:** | Req#2 Move through the map | | |
| --- | --- | --- | --- |
| **Summary:** | The objective of this req#2 allows the user the ability to move across the map as long as he moves along the paths delimited along each of its vertices. For this 5 buttons are used based on each of its columns, in the case where the user is in a position where the 5 columns (branches of the network) are not present, these buttons are disabled so that the user does not move through empty points. | | |
| **Inputs** | **Input name** | **Data Type** | **Conditions for valid values** |
| Buttom1 | Event | Do a click on the section of the button with and the player will moved to the colum1 |
| Buttom2 | Event | Do a click on the section of the button with and the player will moved to the colum2 |
| Buttom3 | Event | Do a click on the section of the button with and the player will moved to the colum3 |
| Buttom4 | Event | Do a click on the section of the button with and the player will moved to the colum4 |
| Buttom5 | Event | Do a click on the section of the button with and the player will moved to the colum5 |
| **Results or postcondition** | The system receives an event according to the selected button and moves to the corresponding column, as long as the movement to that column is enabled. | | |
| **Outputs** | **Output Name** | **Data type** | **Format** |
| MovementPlayer | Player | Canvas |
|  |  |  |

| **Identifier and Name:** | Req#3 Win the game | | |
| --- | --- | --- | --- |
| **Summary:** | The objective of this requirement is that the user is allowed to win as long as he complies with the design of the video game which is to start from a starting point and end at an end point regardless of the path taken. | | |
| **Results or postcondition** | The system verifies the execution time of the program and the vector in which the user is located, in case he/she has reached the last vector and will pass, a victory message is sent, but if he/she is located in another vector different from the start or end vector, then this condition is not fulfilled. | | |
| **Outputs** | **Output Name** | **Data type** | **Format** |
| Win | Screen | Canvas |

| **Identifier and Name:** | Req#4 Lose the game | | |
| --- | --- | --- | --- |
| **Summary:** | The purpose of this requirement is to note that just as the user has the ability to win the game he/she also has the opportunity to lose the game, in the event that he/she does not reach the last vector on the map, the user is prompted with a message that he/she has lost the game along with a retry button to resume the video game immediately. | | |
| **Results or postcondition** | The system verifies the execution time of the program and the vector in which the user is located, in case he/she has reached the last vector and will pass, a victory message is sent, but if he/she is located in another vector different from the start or end vector, then this condition is not fulfilled and the system displays a message that you have lost the game with a new button that said “try again”. | | |
| **Outputs** | **Output Name** | **Data type** | **Format** |
| Lose | Screen | Canvas |
| tryAgain | Button | Canvas |

| **Identifier and Name:** | Req#5 Assign randomly the type of stations around the map | | |
| --- | --- | --- | --- |
| **Summary:** | The purpose of this requirement is to randomly generate the type of stations along the map each time the program is initialized. | | |
| **Results or postcondition** | As soon as the system receives the action of the start button, it randomly generates the type of stations along the map.. | | |
| **Outputs** | **Output Name** | **Data type** | **Format** |
| ScreenB | Screen | Canvas |

| **Identifier and Name:** | Req#6 Mark the hardest and easiest way through the map | | |
| --- | --- | --- | --- |
| **Summary:** | The objective of this requirement is that when the program is initialized, the easiest way to get from the initial vertex to the end is marked in green. In the same way that the most difficult path is painted red, this with the purpose of giving certain recommendations to the user to have a better experience. | | |
| **Results or postcondition** | As soon as the system receives the action of the start button, mark the hardest and easiest way through the map | | |
| **Outputs** | **Output Name** | **Data type** | **Format** |
| Image1 | Image | Canvas |
| Image2 | Image | Canvas |

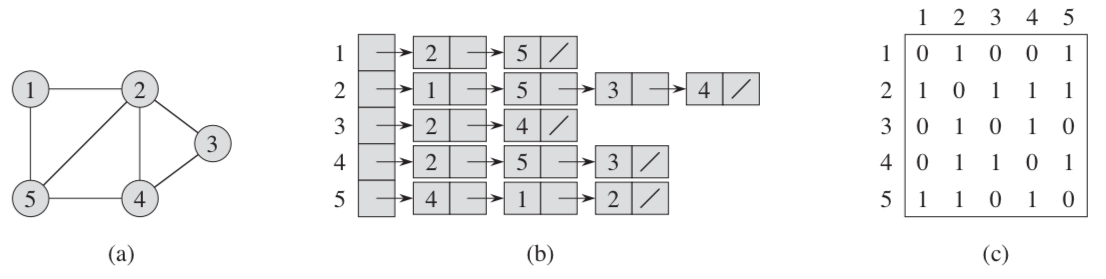
## 

## Phase 2: Gathering the necessary information

**Graphs:** A graph is a collection of vertices and edges such that G = {V, E}. Each vertex and edge can collect varied information but they are all used to make representation of general objects and the relations between them; each edge showing which objects are related and in what manner.

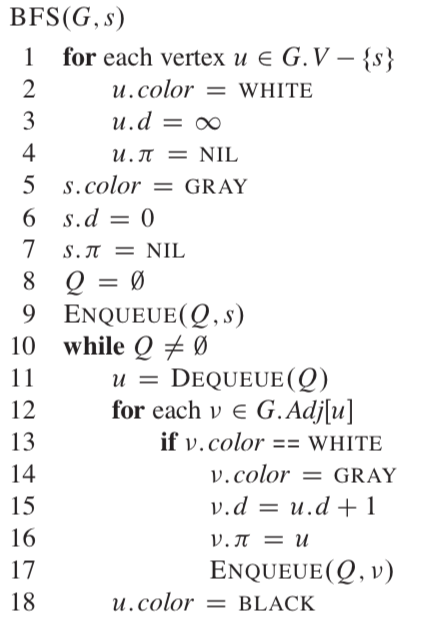
There are various types of graphs used to represent multiple situations taking into account the types of connections between the vertices (edges). Undirected graphs have edges that do not specify any direction. Directed graphs do have edges that must be orderly followed. Weighted graphs have edges with associated cost to every edge. Unweighted graphs on the other hand, don't have any costs per edge. A simple graph has only one edge between every pair of vertices. A multigraph has multiple parallel edges but no loops (edge that starts and ends in the same vertex). On the other hand, a pseudo graph contains parallel edges and does allow loops. There are many other types of graphs but they all are used to analyze complex systems, model behaviors, etc. On the other hand, large graphs tend to be computationally expensive and most algorithms take a long time to show results.

Generally speaking there are two main ways to represent a graph, using an adjacency list or an adjacency matrix. Either representation does the same, but one is usually preferred over the other in certain situations: when a graph is sparse (the number of edges is much less than the number of vertices squared) a list of adjacency tends to work better, but when the graph is dense (the number of edges is close to the number of vertices squared) a matrix tends to work better. Performance wise, an adjacency matrix works better if one needs to know constantly that an edge exists between two vertices.

 **Figure 1**. Representation of a graph with adjacency lists and adjacency matrix (Cormen et. al, 2009).

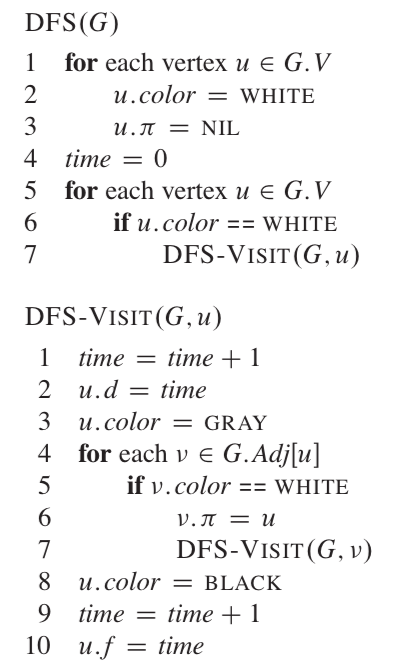
As shown in **Figure 1** an adjacency list will hold the information of what vertices are adjacent to every vertex while the adjacency matrix will number the vertices in an arbitrary way and then use a 2D matrix to represent which vertices have an edge between them. When working with a weighted graph, the information about the ending vertex, the weight and other variables can be saved within the adjacency list for every vertex. On the other hand, in matrix weight could be saved within the entry of row u and column v if there is a single edge between two vertices.

**BFS:** Breadth-First search is an algorithm used on a graph to explore from a root vertex and then discover every other reachable vertex from the root. The algorithm will generate the shortest distance to every vertex (number of edges), and a breadth-first tree which would show the shortest path from any vertex to the root. The algorithm uses colors, distances and predecessors. The used colors are white (undiscovered vertex), gray (discovered but not searched), and black (discovered and completely searched). The distances are used to calculate how many vertices are from the root to every other vertex in the graph. Finally, the predecessor is used to make some type of traceability of what vertex was used to discover any other vertex (creates the BFS tree). As seen in **Pseudocode 1** the algorithm starts by inputting a certain vertex. Then the process will reset all vertices and then use a queue to orderly and by breadth, meaning that all new discovered vertices will be searched before proceeding to a deeper level. Finally, all vertices will have their distance from the root assigned, a predecessor, and their color changed accordingly. The whole process ends up having a time complexity of theta(V + E).



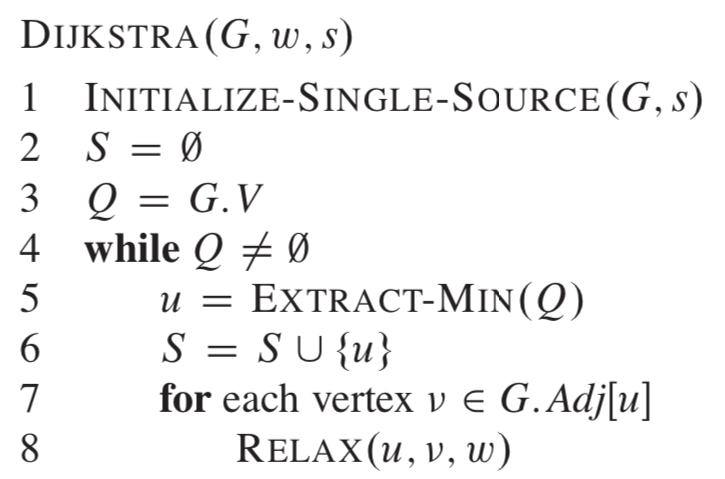
**Pseudocode 1.** BFS algorithm (Cormen et. al, 2009).

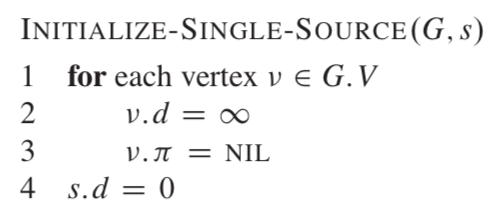
**DFS:** Depth-first search is an algorithm that is able to construct the shortest path from an arbitrary vertex and going down a path until no other vertex is found. The algorithm will go back to the original source and look for other paths as before. This algorithm will go through all vertices, even if the graph is not connected. This search will also depend on a predecessor and a color, but the distance will be measured by a time-stamp. The used colors in the algorithm have the same meaning as in the BFS algorithm, to make sure a vertex is not looked at twice. The distance used as a time-stamp will help measure the depth since a vertex is marked the moment it was discovered and how much time it took to finish looking through that path. Finally, the predecessor will be used to make a DFS forest, since each vertex will end up making an independent tree showing the relation of which vertex was used to discover that new vertex. As seen in **Pseudocode 2**, the algorithm will reset all the attributes of the vertices and then it will go visit every vertex as long as the vertex was not discovered. Inside the DFS-visit method then the time for that vertex will be set, the predecessor is defined as the vertex that the new vertex was discovered through and the color will be changed. Afterwards the method is called recursively on one of the adjacent vertices so the search is done by depth. Once there is no other vertex to find there is one last time-stamp added to each discovered vertex. The whole process ends up having a time complexity of theta(V + E).

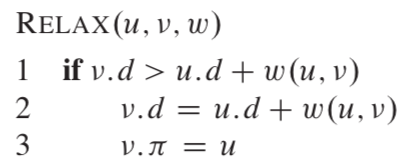
****

**Pseudocode 2.** DFS algorithm (Cormen et. al, 2009).

**Dijkstra:** It’s one of the algorithms that solves the problem of determining the shortest-path. This process is done in a weighted graph with edges that are non negative. This algorithm will find the shortest distance from a root vertex to every other edge on the graph that is connected. This algorithm will use a distance and predecessor attributes in the vertices, but in this case, the distance will refer to the actual weight of the edges up until that specified vertex. As shown in **Pseudocode 3** the algorithm will first reset all the values of weight and predecessor. Afterwards it will create a set that will contain all vertices whose weight from the root vertex has already been determined. This algorithm also uses a min priority queue to order the vertices based on their distances. Then the algorithm will go through the entire set of vertices in the min-priority queue and relax all edges that are adjacent to the recently extracted vertex from the queue. The relaxing process just compares the current distance saved in a vertex to the possible weight that could be changed to by adding the weight of the adjacent vertex and the edge that connects the two edges. The distance and predecessor will only be changed if the distance is less than the one set before. The whole process ends up having a time complexity of O(V2) but it can be reduced to O(V log V + E) if a fibonacci heap is used and the graph is sparse enough.

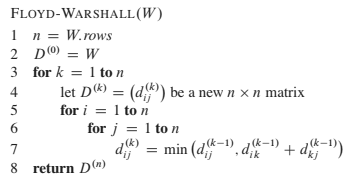
****

****

****

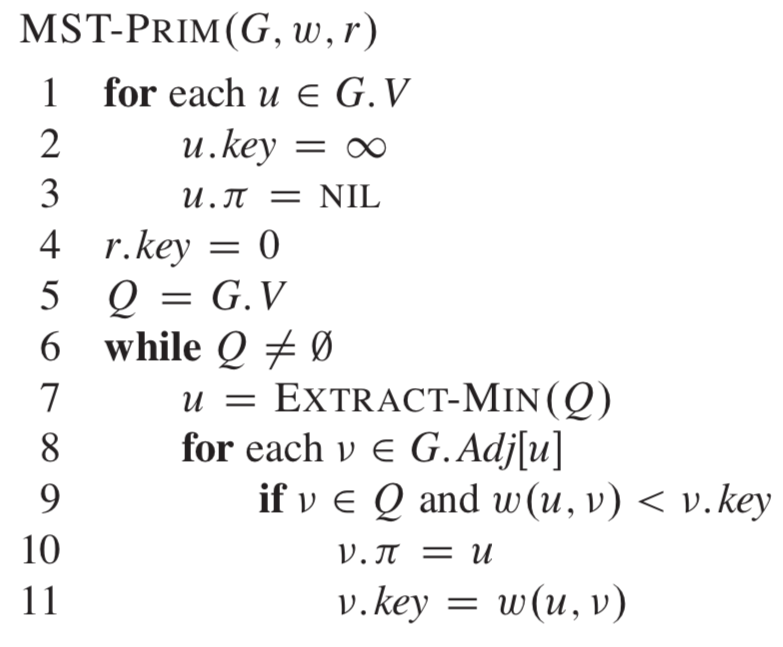
**Pseudocode 3.** Dijkstra algorithm (Cormen et. al, 2009).

**Floyed-Warshall:** This algorithm was formulated to solve a shortest distance problem in graphs the same as a dijkstra, but in this case instead of starting at a specific vertex, it will use a matrix to calculate all the shortest paths between pairs of vertices. Generally speaking, the algorithm utilizes intermediate vertices to evaluate the shortest path between two vertices. That is, for every pair of vertices, every other vertex will be considered as an intermediate step to establish the shortest path between those two vertices. The algorithm is based on the fact that if every edge between every intermediate vertex is the minimum, then the connection between every two vertices will be the minimum too. In the end, this process will end up creating a matrix that shows the shortest path between every pair of vertices. **Pseudocode 4** shows that the algorithm first creates a matrix filled with 0’s based on a matrix of vertices. Then the new matrix will start going through every element and assigning the minimum distance while also forcing to check if the current distance and the possible distance taking an intermediate vertex is less to make sure that the current path taken is the shortest between all possible intermediate vertices (if needed). The whole process ends up having a time complexity of theta(V3).

****

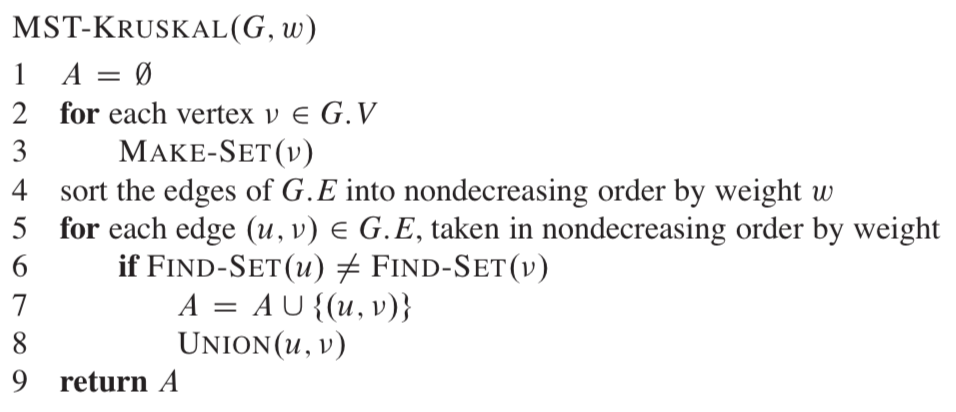
**Pseudocode 4.** Floyd-Watshall algorithm (Cormen et. al, 2009).

**Prim:** The Prim algorithm is one that helps in the creation of a minimum-spanning tree. A minimum spanning tree will create a tree-like structure where each vertex represents a node and the edges will be the ones with the least distances or weight in a weighted graph. This algorithm will create a single tree with an arbitrary root and it will cover all vertices in the graph adding a vertex one by one which holds the least weight. The attributes needed for this algorithm will be a key which represents the minimum weight to get to that vertex and a predecessor, representing the vertex by which one can get to that vertex with the minimum distance. As shown in **Pseudocode 5**, the algorithm will first reset the key and predecessor value of every vertex to the maximum possible distance and predecessors as unknowns. Then it will set the distance to 0 to the root and afterwards use a min-priority queue that will hold all vertices that haven’t been added to the tree. The vertices will be extracted from the priority queue in order of which has the least weight and marking each vertex only if the key value is less than the one put before. The whole process ends up having a time complexity of theta(E log V) or theta(E + V log V) depending if a min-heap or a fibonacci-heap were used on the min-priority queue.

****

**Pseudocode 5.** Prim algorithm (Cormen et. al, 2009).

**Kruskal:** Even though Kruskal is another algorithm to generate a minimum-spanning tree, different to Prim, this algorithm will generate a forest. This algorithm works by adding the “safe edge” which is one that connects to trees with the least possible weight. The shown implementation in **Pseudocode 6** utilizes disjoint sets to recognize if an element can be added or not to the set that is currently being checked out. After the edges are sorted by increasing order by weight, it will start testing for every edge if the set of the starting and ending vertex is different, which in that case both sets will be united (avoids creating a cycle). The whole process ends up having a time complexity of theta(E log V) (assuming that the number of edges is much less than V2).

****

**Pseudocode 6.** Kruskal algorithm (Cormen et. al, 2009).

**Bibliographic references:**

Geeksforgeeks. (2024). Types of Graphs with Examples. <https://www.geeksforgeeks.org/graph-types-and-applications/>

Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2009). *Introduction to algorithms* (3rd ed.)*.* Massachusetts Institute of Technology.

## Phase 3: Looking for creative solutions

**Method to produce original ideas**: Attribute listing. An attribute list, in this context, would be

made by gathering and ordering all the possible information we can think of about the product

that has been asked to make. The attribute is a breakdown of all the components which are

essential to this product. Afterwards there should be an idea about how to change or innovate

for each attribute.

Subject: Desktop version of a program to manage a player, graphs,minimum paths,map,campfires,bosses.

General attributes:

* Player:
  + We want to create a player with the ability to move on the edges which will be painted in the map, besides if the player is in a corner they couldn't move to another edge that is not connected in the vertex that he stays in..
* Map:
  + Use a map with 50 vertex and in each vertex will be present a challenge, or in another cases they will gain more points of health, life in the campfire but when the player try to fight with a boss, the game will take an amount of health player to put on at risk so if he lose her health will be reduce by this amount, but if he win this same amount will be added to his current life and return the amount.
* Move with Button:
  + The buttons will be implemented with the objective that the player only can move with them, and these vary according to the player's position and some buttons will be available and some will not.
* Use interface:
  + We use a interface with javafx which is related to the image of the player and the paint of the map, whom include the edges of the map, the signal of campfire and boss, besides we will create two points, one of them will call point start and the other point will call point end,these points will be painted with doors so when the player start and end,shows as if it had passed the level.
* Show green and red paths:
  + The game will use two ways to give advice to the player,the first way will call the easy path, in this case it is the path with a major probability to win, because if the player chooses this way, he doesn't lose much life. The other path will call the hard path, because if the player chooses this way, he will have lower probability of winning but if he wins, the points will be more than if he win using the easy path. But if the player doesn’t want to walk around the easy or the hard path, they can walk around his own path, the only restriction the player will have is that he can only place himself between the edges and vertices present but he does not have to follow an easy or difficult path.

**Creative solution 1: BFS and Dijkstra**

* Implement a BFS to provide a minimum path and later this path will be painted with a color green, in the same way we will use a maximum path which will be painted with a color red.
* Implemented Dijkstra to search which path will have the minimum cost of life to travel, and later when he found it, they painted the path with green and this will be the path with less probability to lose points, and in the same way a path with the maximum cost of life.

**Creative solution 2: DFS and Floyd Warshall**

* Implements DFS to search all the paths in the graphs and the DFS will choose a minimum path and later will be paint with green color
* Implements Floyd Warshall to find the shortest path in all the vertex of the graph, not only in the the minimum path of the start and end vertex, also in each path which have any vertex, and with this matrix of adjacency we choose the path with minimum cost, from the start to the end

## Phase 4: Transition from the formulated ideas to preliminary designs

**Discarded idea:** Creative solution 2:

**DFS has lower efficiency:**

* Complexity in finding minimal paths: DFS is not designed to find minimal or weight-based paths. It explores all possible paths, which can be extremely inefficient in large networks or networks with many paths.
* Lack of focus on costs: DFS cannot guarantee that the path chosen is optimal in terms of distance or cumulative cost. This could lead to the selection of suboptimal paths.
* Unnecessary recursion: DFS uses recursion or an explicit stack, which can complicate implementation and make it less efficient in resource-constrained systems.

**Floyd-Warshall is very expensive:**

* Unnecessary globality: Floyd-Warshall computes the shortest distances between all pairs of vertices. In this case, only the distance between an initial node and an end node is needed, which makes Floyd-Warshall an unnecessary computational overhead.
* High complexity: Its complexity is O(V³), which makes it unsuitable for large graphs. If the network has hundreds or thousands of nodes, the execution time would be unacceptable.

**Possible creative idea that could be implemented**

**Creative idea 1:**

The BFS algorithm will initially be implemented to find the shortest path in terms of steps (number of edges) within the graph. This path will be visually represented with the color green, highlighting it as the simplest and safest route for the player to move forward with ease.

Subsequently, a variant of BFS will be used to explore the graph and find the longest possible path (in terms of number of steps) between the initial node and the final node. This path will be highlighted in red, representing a more challenging option in terms of length.

On the other hand, the Dijkstra algorithm will be run to determine the path with the lowest cumulative cost considering the weights of the edges of the network. This path will be identified as the most feasible and will be painted green, since it minimizes the penalties or risks. Additionally, Dijkstra will be reused to search for the path with the highest cumulative cost, marking this as the most costly path and representing it in red color, for those players who wish to face a higher difficulty.

Finally, a scoring system will be implemented in which the green path will award a lower amount of points due to its low difficulty, while the red path will reward a significantly higher amount, reflecting the risks taken by the player.

## Phase 5: Evaluation and selección of the best solution

**Criteria:**

1. The efficiency of the structures that are implemented:
   1. [5] = Both solution algorithms have adequate efficiency for large graphs
   2. [3] = The algorithms work, but their efficiency is not optimal for large graphs or have a significant impact on performance.
   3. [1] = The algorithms have a complexity that could make them infeasible for large graphs.
2. Flexibility and adaptability to the problem (green and red paths):
   1. [5] = The solution perfectly covers both needs: the green path (minimum) and the red path (maximum or most costly).
   2. [3] = The solution partially solves the needs, but not all paths are identified efficiently.
   3. [1] = The solution does not adapt to the problem and does not identify minimum and maximum paths correctly.
3. Resource consumption (space and memory):
   1. [5] = Algorithms are memory efficient
   2. [3] = Memory consumption is moderate and can be manageable depending on the size of the network.
   3. [1] = Memory consumption is high and could cause problems in large graphs.
4. Scalability and Path Coverage:
   1. [5] =The solution visit all the paths of the graph to find a solution
   2. [3] =The solution works for medium-sized graphs and can identify a reasonable range of paths.
   3. [1] = The solution struggles to scale or cannot effectively cover all the necessary paths.

|  | **Criteria A** | **Criteria B** | **Criteria C** | **Criteria D** | **Total** |
| --- | --- | --- | --- | --- | --- |
| **Solution 1** | 5 | 3 | 5 | 1 | 14 |
| **Solution 2** | 3 | 1 | 1 | 5 | 10 |