Esteban Ariza, Johan Sebastián Giraldo, Mateo Valdés

Final Project

Problem Definition:

Johan, Esteban and Mateo have traveled to Japan for a job interview on Nintendo. When they arrived at Nintendo's skyscraper, mr. Sakurai said "You must prove your courage by creating the best mobile game I have ever seen. In this contest, Shigeru Miyamoto-san will choose the best videogame designer, based on their programming skills and imagination". After that, PandiYa members started creating the best game ever.

Mateo, the mastermind, had the incredible idea of creating a video game that includes spatial gaming skills, Esteban started planning the gameplay and Johan the design itself. The main idea of the game is to choose, among many paths, the shortest path between two points, but this concept is not simple at all. The player has a limited amount of movements and some levels have special properties that make the game even harder. In addition, if you achieve a level with the minimum steps possible, you could get more stars (3 if you get a perfect score) to unlock the next levels. For the level designs, they decided to import the levels from a flat file with all the information of the challenge. Finally, they need to add funny stuff to keep the attention of the player.

1. Identifying the problem

Problem: A game must be created wherein a player must choose, among many paths, the shortest path between two points to achieve all the levels.

2. Research

Functional requirements:

Name	FR1: Load a level
Summary	The system allows creating a level by reading a plain text file that includes the number of vertices of the graph and their coordinates on the screen, the adjacency matrix, the vertex where start and the vertex where ends, the level number, number of stars required for unlocking the level and finally, the color of the character.

Input	The **path** of the plain text file with the level information
Output	The level was created successfully

Name	FR2: Unlock a level
Summary	The system allows unlocking a level. A level only can be unlocked if the user has the number of stars required for the level.
Input	-
Output	The level was unlocked successfully

Name	RF3: Calculate minimum movements required
Summary	The system allows calculating the minimum movements required to go from the start vertex to the ends vertex,
Input	-
Output	The minimum movements to go from the start vertex to the ends vertex

Name	RF4: Calculate maximum movements allowed
Summary	The system allows calculating the maximum movements allowed to go from the start vertex to the ends vertex. The way to calculate the number of movements allowed is adding up two to the number of movements required for the minimum path
Input	-
Output	The maximum movements

Name	FR5: Play a level
Summary	The system allows playing a level, the level is

	passed if the path lets go from the start vertex to the ends vertex with the movements allowed.
Input	A path from the start vertex to ends vertex
Output	The level is played successfully

Name	FR6: Calculate the number of stars earned in the level.
Summary	The system allows calculating the number of stars earned depending on how many movements were left over, if there are 2 movements left, earns 3 stars, if there are 1 movement left, earns 2 stars, if there are no one movements left, earns 1 star.
Input	-
Output	The number of stars earned

Matrix
Matrix is a simple data structure that stores data in an organized form in the form of rows and columns. It could be useful to manipulate data in an easier and more organized way. Some problems of the matrix, is the spatial complexity ($O(n^2)$) and is not efficient in some searching algorithms.
The matrix is very useful to represent 2D problems and is also used in graphs as Adjacency Matrix to represent the edges between two vertices.

Vector
Java, C++ and other programming languages use vectors or arrays to store data. It’s also used in some game engines such as Unity and Unreal to make easier and fluid transitions, translations and scales.

Graph
A graph is a data structure that has a finite set of vertices and a finite set of edges, where a vertex or also called node, is an element of the graph and an edge is a vertex connection by pairs in the form (u,v).

Graphs can be used to model many real life situations such as the roads of a city, relationships of influence,

There are two commonly used representation of graphs:
1. Adjacency List
2. Adjacency Matrix

Tree
Trees, a particular type of graph that is organized hierarchically. The first vertex is called root, and it is connected with nodes that are also connected with other nodes, creating a structure similar to a tree. A tree is a very particular way of representing a graph, because it is easier to visualize the connections between the nodes.

References:

Wankhede, Rahul. Different Operations on Matrices. GeeksForGeeks.
https://www.geeksforgeeks.org/different-operation-matrices/

Msakibhr, Estenger. Vector in C++ STL. GeeksForGeeks.
https://www.geeksforgeeks.org/vector-in-cpp-stl/

Malhotra, Kanav. Graph and its representations. GeeksForGeeks.
https://www.geeksforgeeks.org/graph-and-its-representations/

Barnwal, Aashish. Graph and its representations. GeeksForGeeks.
https://www.geeksforgeeks.org/graph-and-its-representations/

Singh, Shikha. Binary Tree Data Structure. GeeksForGeeks.
https://www.geeksforgeeks.org/binary-tree-data-structure/

3. Creative solutions.

Alternative #1:
Have the user draw the points and paths in a piece of paper, and then ask him to find the shortest path and keep track of the score himself manually.

Alternative #2:
Have the user draw the points and paths in a piece of paper, and then ask him to find the shortest path and have a friend or acquaintance keep track of the score manually.

Alternative #3:
Use a tree to represent the points and the paths, with the tree nodes representing the points and the connections between the parent nodes and the children nodes representing the paths.

Alternative #4:

Use a vector or arraylist of size two to store every pair of points that are connected by a path.

Alternative #5:
Use a matrix as a game board of nodes, in which the user could travel between nodes as a labyrinth.

Alternative #6:
Use a graph to represent the points and the paths, with the nodes in the graph representing the points and the edges representing the paths between each two points.

Alternative #7:
Use a matrix of points as a game board, and if two points are in the same column or row then they are connected by a path.

Alternative #8:
Use a Doubly LinkedList of points, and if a point is linked to another point in the LinkedList then there is a path between those two points.

4. Transformation to preliminary designs

Alternative #1:
This alternative might be tedious or boring to the user, and if he is the one responsible for keeping his own score, then he might be dishonest or cheat by increasing his score. Therefore this alternative is discarded.

Alternative #2:
This alternative might also be tedious or boring to the user, and the user might be dishonest and try to persuade the friend or acquaintance to increase his score. Therefore this alternative is discarded.

Alternative #3:
Trees are a very popular and useful data structure, and they might even be useful in this case because their nodes can represent the points in this specific problem. Thus, we must consider this alternative.

Alternative #4:
This implementation seems easy and straightforward to implement in order to successfully solve our problem, so we must consider this alternative.

Alternative #5:
Alternative seems too complicated to implement effectively, so we must unfortunately discard this alternative.

Alternative #6:
This alternative seems to model this problem perfectly, since the points can be represented by nodes and the paths can be represented by edges, so we must consider this alternative.

Alternative #7:
Even though the program might take up a lot of space if the matrix gets too big, this alternative still gives us a valid solution and we must therefore consider it.

Alternative #8:
This alternative lacks versatility because if we use a Doubly LinkedList then a point can have a maximum of two paths, and our program might need more than two paths. Therefore, we must discard this alternative.

5. Evaluation and selection of the best solution
Criterion A: The problem model.

- [3] Represent the problem the best way possible.
- [2] The problem could be represented at least with this model.
- [1] The model is not appropriate for this problem.

Criterion B: Versatility in this problem.

- [3] This solution is very versatile and works in most situations.
- [2] This solution is moderately versatile and works in certain situations.
- [1] This data structure is not versatile at all and can only work in limited situations.

Criterion C: The difficulty of implementation.

- [3] The implementation is simple, even trivial
- [2] Average people can implement this solution.
- [1] Very difficult to actually implement

Criterion D: The speed of the solution.

- [3] The solution is the fastest discovered in history
- [2] The solution is relatively fast
- [1] The solution is extremely slow

Solution	**Criterion A**	**Criterion B**	**Criterion C**	**Criterion D**	**Total**
Tree	3	1	2	3	9
Vector	1	1	2	2	6
Graph	3	3	3	3	12

Matrix	2	2	2	2	8

Justification:

- Tree is a good way to represent the problem, however it lacks the versatility of a graph.
- Is very difficult to represent this problem as a vector, because of the limitations of this data structure.
- A graph is a good solution for the problem, because it is very intuitive and easy to represent in this particular context.
- The problem could be represented as a matrix, but there are a lot of better options for the implementation.

6. Preparing the report and specifications

ADT - Graph

Graph G = <N, E>

- N is a collection of nodes $\{n_1, n_2, n_3, ..., n_n\}$
- E is a collection of edges { $e_1, e_2, e_3, ..., e_n$ } that connect two nodes

$|N| \geq 0, \; |E| \geq 0$

Primitive operations:

CreateGraph:		→	Graph : Creator
AddNode:	Graph x Node	→	Graph : Mutator
RemoveNode:	Graph x Node	→	Graph : Mutator
AddEdge:	Graph x Node x Node	→	Graph : Mutator
RemoveEdge:	Graph x Node x Node	→	Graph : Mutator
NodesSize:	Graph	→	Integer : Observer
EdgesSize:	Graph	→	Integer : Observer
GetNodes:	Graph	→	Nodes : Observer
GetEdges:	Graph	→	Edges : Observer
GetNode:	Graph x Node	→	Node : Observer
AreConnected:	Graph x Node x Node	→	Boolean : Observer

CreateGraph()

“Creates a new graph”

{ pre: $true$ }
{ post: $\exists G \wedge N = \{\} \wedge E = \{\}$}

AddNode(Graph, Node**)**

“Adds a node to N, the graph’s collection of nodes”

{ pre: $\exists G$ }
{ post: $n \in N$ }

RemoveNode(Graph, Node**)**

“Removes a node from N, the graph’s collection of nodes”

{ pre: $\exists G \wedge N \neq \{\}$ }
{ post: $n \notin N$ }

AddEdge(Graph**,** Node, Node**)**

“Adds an edge that connects the two nodes”

{ pre: $\exists G \wedge n_1, n_2 \in N$ }
{ post: $e_{n_1,n_2} \in E$ }

RemoveEdge(Graph**,** Node, Node**)**

“Removes the edge that connects the two nodes”

{ pre: $\exists G \wedge n_1, n_2 \in N \wedge e_{n_1,n_2} \in E$ }
{ post: $e_{n_1,n_2} \notin E$ }

NodesSize(Graph**)**

“Returns the size of N, the graph’s collection of nodes”

{ pre: $\exists G$ }
{ post: |N| }

EdgesSize(Graph**)**

“Returns the size of E, the graph’s collection of edges”

{ pre: $\exists G$ }
{ post: |E| }

GetNodes(Graph**)**

“Returns N, the graph’s collection of nodes”

{ pre: $\exists G$ }
{ post: N }

GetEdges(Graph**)**

“Returns E, the graph’s collection of edges”

{ pre: $\exists G$ }
{ post: E }

GetNode(Graph, Node**)**

“Returns n, which is a node inside N, which is the graph’s collection of nodes”

{ pre: $\exists G \wedge n \in N$ }
{ post: n }

AreConnected(Graph, Node, Node**)**

“Evaluates if there is an edge in the graph that connects the two nodes”

{ pre: $\exists G \wedge n_1, n_2 \in N$ }
{ post: $V \in \{true, false\}$ }