Algorithms and data structures

Final Proyect - 2019-2

Daniel Villota - A00356255, Natalia Gonzalez - A00354849

Engineering Method

(problem 1 – 1112 Mice and Maze)

Problem definition:

Judge: Online Judge
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=3553

link: https://drive.google.com/file/d/12zpE-cB_zDxe5ajTpYpyCBEaEufSy3M9/view?usp=sharing

Phase 1: Problem identification

The question that defines this problem is: How many mice manage to leave the maze in time?

Phase 2: Collection of the necessary information

All the information necessary to understand the problem is provided by the statement, the entries are specific and it is the only thing that will be obtained.

Phase 3: Search for creative solutions

To solve this problem, an algorithm is needed to find the path of the lowest weight of the labyrinth, from each of the cells. For this you can use algorithms known as:

- Floyd-Warshall
- Dijkstra

Another option may be to look for all possible paths until one meets the time limit and if it is not possible to discard it

Phase 4: Transition of the formulation of ideas to preliminary designs

The idea of looking for all possible paths is likely to function, but the complexity is the highest compared to the other options, therefore it is ruled out.

On the other hand, the Floyd Warshall option has a complexity of V^3 and the minimum weight of all roads is obtained

Also, the Dijkstra algorithm has a temporal complexity of V to 2 with a single path, so this algorithm must be repeated V times, therefore, the serious complexity of V^3

Phase 5: Evaluation and selection of the best solution

To solve this problem, Floyd Warshall was selected, since it is easier to implement and at the end of the algorithm the answer is already obtained, while for the Dijkstra algorithm, it is necessary to repeat the algorithm to verify if each mouse manages to exit

Phase 6: Preparation of reports and specifications

Functional requirements

Name	**R.# 1. Find the number of mice that manage to get out of the maze**
Summary	Find the number of mice that leave the maze in the given time
Input	
• Number of cages • Departure location • Maximum time determined to exit • Connection between cages • Time it takes to go from one cage to another Results	
Output	
A number that indicates the number of mice that manage to get out of the maze	

Name	**R.# 2. Visualize the maze**
Summary	Summary Show the shape of the maze
Inputs	
• Number of cages • Departure location • Maximum time determined to exit • Connection between cages • Time it takes to go from one cage to another	
Output	
A screen maze display	

Nombre	**R.# 3. Visualize the cages of the mice that manage to leave the labyrinth**
Resumen	It allows to visualize the mouse cages that manage to leave the labyrinth
Input	
None	
Output	
A visualization of the maze on the screen with the cages of the mice that managed to leave a different color	

UNIVERSIDAD ICESI

Engineering Method

(problem 2 – 11631 Dark roads)

Problem definition:

Judge: Online
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2678

link: https://drive.google.com/file/d/1gN7QIJUoG5PKY4rR4p72jPhRXBAQ62Ch/view?usp=sharing

Phase 1: Problem identification

The main problem is knowing the maximum amount of money that the government can save without sacrificing the security of citizens

Phase 2: Collection of the necessary information

All the information necessary to understand the problem is provided by the statement, the entries are specific and it is the only thing that will be obtained.

Phase 3: Search for creative solutions

To solve this problem, we must implement an algorithm that allows us to simulate the map of the city and find the roads that have the least cost and it is possible to reach all places. For this you can use algorithms that already exist as Prim or Kruskal.

Phase 4: Transition of the formulation of ideas to preliminary designs

The Prim algorithm has a temporal complexity of O (n2) and the Kruskal algorithm has a similar complexity of O (n * log (n)).

Phase 5: Evaluation and selection of the best solution

For ease of implementation it was decided to use the Prim algorithm.

Phase 6: Preparation of reports and specifications

Functional requirements

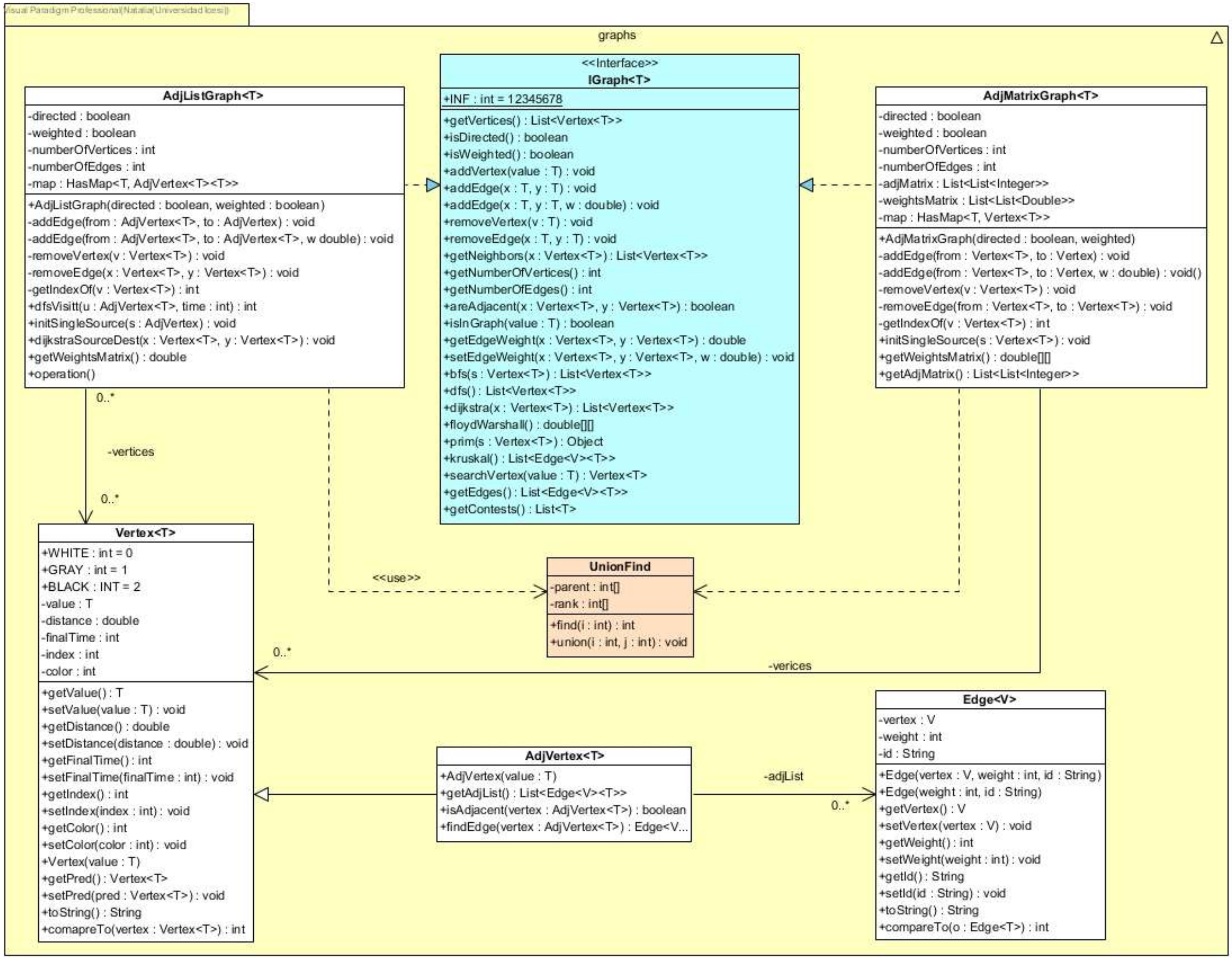
Name	**R.# 1. Find the maximum amount of money that can be saved**
Summary	Find the maximum amount of money that can be saved without sacrificing the safety of citizens
Input	
• Number of places • Number of roads • Connection between places • The distance of the connections	
Output	
A number that indicates the money that can be saved with this method	

Name	**R. # 2. Display the shape of the city**
Summary	Display the shape of the city
Input	
• Number of places • Number of roads • Connection between places • The distance of the connections	
Output	
A display of the city by screen	

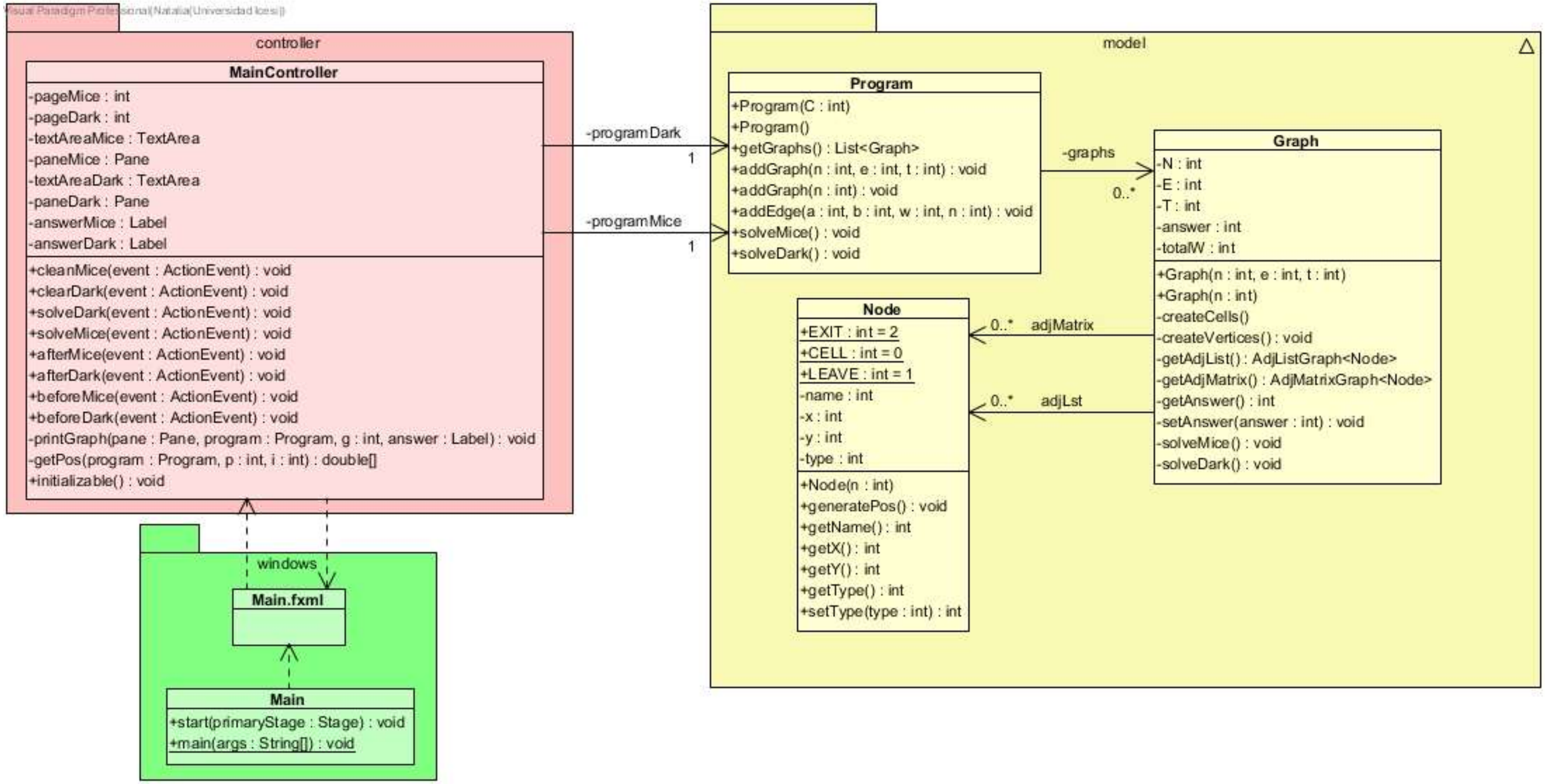
Name	**R.# 3. View the route that is on and connect all places in the city**
Summary	View the route that remains on
Input	
None	
Output	
A visualization of the shortest streets that connect all places in the city.	

UNIVERSIDAD ICESI

Class Diagram Graph Implementation


Visual Paradigm Professional(Natalia(Universidad Icesi))
graphs
<<Interface>>
IGraph<T>
+INF : int = 12345678
+getVertices() : List<Vertex<T>>
+isDirected() : boolean
+isWeighted() : boolean
+addVertex(value : T) : void
+addEdge(x : T, y : T) : void
+addEdge(x : T, y : T, w : double) : void
+removeVertex(v : T) : void
+removeEdge(x : T, y : T) : void
+getNeighbors(x : Vertex<T>) : List<Vertex<T>>
+getNumberOfVertices() : int
+getNumberOfEdges() : int
+areAdjacent(x : Vertex<T>, y : Vertex<T>) : boolean
+isInGraph(value : T) : boolean
+getEdgeWeight(x : Vertex<T>, y : Vertex<T>) : double
+setEdgeWeight(x : Vertex<T>, y : Vertex<T>, w : double) : void
+bfs(s : Vertex<T>) : List<Vertex<T>>
+dfs() : List<Vertex<T>>
+dijkstra(x : Vertex<T>) : List<Vertex<T>>
+floydWarshall() : double[][]
+prim(s : Vertex<T>) : Object
+kruskal() : List<Edge<V><T>>
+searchVertex(value : T) : Vertex<T>
+getEdges() : List<Edge<V><T>>
+getContests() : List<T>
AdjListGraph<T>
-directed : boolean
-weighted : boolean
-numberOfVertices : int
-numberOfEdges : int
-map : HasMap<T, AdjVertex<T><T>>
+AdjListGraph(directed : boolean, weighted : boolean)
-addEdge(from : AdjVertex<T>, to : AdjVertex) : void
-addEdge(from : AdjVertex<T>, to : AdjVertex<T>, w double) : void
-removeVertex(v : Vertex<T>) : void
-removeEdge(x : Vertex<T>, y : Vertex<T>) : void
-getIndexOf(v : Vertex<T>) : int
+dfsVisitt(u : AdjVertex<T>, time : int) : int
+initSingleSource(s : AdjVertex) : void
+dijkstraSourceDest(x : Vertex<T>, y : Vertex<T>) : void
+getWeightsMatrix() : double
+operation()
AdjMatrixGraph<T>
-directed : boolean
-weighted : boolean
-numberOfVertices : int
-numberOfEdges : int
-adjMatrix : List<List<Integer>>
-weightsMatrix : List<List<Double>>
-map : HasMap<T, Vertex<T>>
+AdjMatrixGraph(directed : boolean, weighted)
-addEdge(from : Vertex<T>, to : Vertex) : void
-addEdge(from : Vertex<T>, to : Vertex, w : double) : void()
-removeVertex(v : Vertex<T>) : void
-removeEdge(from : Vertex<T>, to : Vertex<T>) : void
-getIndexOf(v : Vertex<T>) : int
+initSingleSource(s : Vertex<T>) : void
+getWeightsMatrix() : double[][]
+getAdjMatrix() : List<List<Integer>>
0..*
-vertices
0..*
Vertex<T>
+WHITE : int = 0
+GRAY : int = 1
+BLACK : INT = 2
-value : T
-distance : double
-finalTime : int
-index : int
-color : int
+getValue() : T
+setValue(value : T) : void
+getDistance() : double
+setDistance(distance : double) : void
+getFinalTime() : int
+setFinalTime(finalTime : int) : void
+getIndex() : int
+setIndex(index : int) : void
+getColor() : int
+setColor(color : int) : void
+Vertex(value : T)
+getPred() : Vertex<T>
+setPred(pred : Vertex<T>) : void
+toString() : String
+comapreTo(vertex : Vertex<T>) : int
<<use>>
UnionFind
-parent : int[]
-rank : int[]
+find(i : int) : int
+union(i : int, j : int) : void
0..*
-verices
AdjVertex<T>
+AdjVertex(value : T)
+getAdjList() : List<Edge<V><T>>
+isAdjacent(vertex : AdjVertex<T>) : boolean
+findEdge(vertex : AdjVertex<T>) : Edge<V...
-adjList
0..*
Edge<V>
-vertex : V
-weight : int
-id : String
+Edge(vertex : V, weight : int, id : String)
+Edge(weight : int, id : String)
+getVertex() : V
+setVertex(vertex : V) : void
+getWeight() : int
+setWeight(weight : int) : void
+getId() : String
+setId(id : String) : void
+toString() : String
+compareTo(o : Edge<T>) : int

Class Diagram for Problems


controller
MainController
-pageMice : int
-pageDark : int
-textAreaMice : TextArea
-paneMice : Pane
-textAreaDark : TextArea
-paneDark : Pane
-answerMice : Label
-answerDark : Label
+cleanMice(event : ActionEvent) : void
+clearDark(event : ActionEvent) : void
+solveDark(event : ActionEvent) : void
+solveMice(event : ActionEvent) : void
+afterMice(event : ActionEvent) : void
+afterDark(event : ActionEvent) : void
+beforeMice(event : ActionEvent) : void
+beforeDark(event : ActionEvent) : void
-printGraph(pane : Pane, program : Program, g : int, answer : Label) : void
-getPos(program : Program, p : int, i : int) : double[]
+initializable() : void
-programDark
1
-programMice
1
model
Program
+Program(C : int)
+Program()
+getGraphs() : List<Graph>
+addGraph(n : int, e : int, t : int) : void
+addGraph(n : int) : void
+addEdge(a : int, b : int, w : int, n : int) : void
+solveMice() : void
+solveDark() : void
-graphs
0..*
Graph
-N : int
-E : int
-T : int
-answer : int
-totalW : int
+Graph(n : int, e : int, t : int)
+Graph(n : int)
-createCells()
-createVertices() : void
-getAdjList() : AdjListGraph<Node>
-getAdjMatrix() : AdjMatrixGraph<Node>
-getAnswer() : int
-setAnswer(answer : int) : void
-solveMice() : void
-solveDark() : void
0..*
adjMatrix
0..*
adjLst
Node
+EXIT : int = 2
+CELL : int = 0
+LEAVE : int = 1
-name : int
-x : int
-y : int
-type : int
+Node(n : int)
+generatePos() : void
+getName() : int
+getX() : int
+getY() : int
+getType() : int
+setType(type : int) : int
windows
Main.fxml
Main
+start(primaryStage : Stage) : void
+main(args : String[]) : void