Engineering Method

Daniel Alejandro Castro Escobar

Katerine Valens Orejuela

Icesi University

Computation and Discrete Structures

November 5, 2023

***First Step: Identification of the problem***

**SOFTWARE ENGINEERING PROBLEM SPECIFICATION TABLE, identifying the following elements**

|  |  |
| --- | --- |
| **CLIENT** | Computation and discrete structures |
| **USER** | Players |
| **FUNCTIONAL REQUIREMENTS** | R1 - Create and show an 8x8 functional board with a randomly placed source and drain.  R2 - Location of the different type of pipes on the board.  R3 - Simulation of the water flow through the pipes in the board.  R4 – Show the best solution for the waterflow |
| **CONTEXT OF THE PROBLEM** | Computation and discrete structures require students to develop a pipe simulation game, which starts with the creation of an 8x8 board with a randomly placed source of water and drain. The user should be able to select the type of pipe they wish to use and place it on the given coordinates on the board. Also, it should allow the user to, whenever they’re ready, simulate the waterflow through the pipes to confirm if their solution is working. Finally, the program is also required to calculate a score for the game. |
| **NON-FUNCTIONAL REQUIREMENTS** | RN1 – The board must be modelled using graphs.  RN2 – The game must work properly with both types of graphs, being adjacency matrices and adjacency lists.  RN3 – The graph must implement a visiting method and one of minimum paths |

**Functional Requirements Analysis Table**

| Name or identifier | R1 - Create and show an 8x8 functional board with a randomly placed source and drain. | | |
| --- | --- | --- | --- |
| Summary | When a game starts, an 8x8 functional board will be created. In this board, the water source and the drain are randomly located. | | |
| Inputs | Input name | Data type | Selection or repetition condition |
| graphType | String | Button 1 - Adjacency matrix  Button 2 – Adjacency list |
| Result or postcondition | A board designed with the selected type of graph is created. The vertices of the graph contain objects of type Box. These boxes contain the different types of pipes. The representation of the board is an 8x8 square matrix with empty boxes, one source, and one drain’. | | |
| Outputs | Output name | Data type | Selection or repetition condition |
| table | String |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Name or identifier | R2 - Location of the pipes on the board. | | |
| Summary | The game must allow to place different pipes on the board depending on the indications of position that the user input (Rows and columns). Then the user chooses the option of pipe to put in that location. The player can repeat this process as much as he needs, until the fountain is linked with the drain. The menu must be presented when the user inserts a new pipe.  The program must not allow the option to change the position of the water source and the drain. | | |
| Inputs | Input name | Data type | Selection or repetition condition |
| Location | String | Row, column |
| pipeType | String | Button 1: Vertical pipe  Button 2: Horizontal pipe  Button 3: Cicular pipe |
| Result or postcondition | The program selects the box in the graph board according to the coordinate provided by the user. Inside the box contained by the node, a pipe of the selected type of pipe is created and placed. The program keeps showing the board and the menu until the user decides to go back to menu or simulate the waterflow. | | |
| Outputs | Output name | Data type | Selection or repetition condition |
|  | board | String | Representation of the updated board in a matrix form. |
|  | menu | String | 1. Place pipe  2. Simulate  3. Go back to menu |

|  |  |  |  |
| --- | --- | --- | --- |
| Name or identifier | R3 - Simulation of the water in the pipes | | |
| Summary | The program must allow, during a game session, to simulate the waterflow through the pipes from the source to the drain. | | |
| Inputs | Input name | Data type | Selection or repetition condition |
|  |  |  |
| Result or postcondition | A method starts at the source and looks for the connected vertices to validate the contents of the boxes and verify if the pipes allow the proper flow of water. Returns true if the pipes located allow water to go from the source to the drain, and false if this is not possible. | | |
| Outputs | Output name | Data type | Selection or repetition condition |
| allowsFlow | boolean | True = The water flow worked  False = The water flow did not work |

|  |  |  |  |
| --- | --- | --- | --- |
| Name or identifier | R4 - Show the best solution for the waterflow | | |
| Summary | The software must show the best solution for the waterflow, marking the best solution on the screen for the player to know where they should place all the pipes in order to use the least amount of pipes, and therefore the greatest score. | | |
| Inputs | Input name | Data type | Selection or repetition condition |
|  |  |  |
| Result or postcondition | When this function is called, the program implements the Dijkstra Algorithm to find the minimum path from the water source to the drain. When this path is found, the nodes in the board are shown in a different color for the player to know where they should place the pipes to get the highest score. | | |
| Outputs | Output name | Data type | Selection or repetition condition |
| bestSolution | Canva | The boxes on the game are painted in a different color |

***Second Step: Information Gathering***

**Definitions:**

* **Structure:** A structure, in computer science, is an organized way of storing data. They are very important for making different operations on some stored collections efficiently. For example, we can find some structures in computing such as : arrays, linked lists, trees, stacks, queues and many others. Each of these structures has its own characteristics, it depends on the context of the problem to use their functionalities and advantages to develop a solution.
* **Linked List**: Is a data structure used for organizing collection of nodes. Linked list store elements in the nodes that are connected to each other. Its functionality consists in a value stored in the node and pointers that allow it to create a link between two nodes making a chain starting with a head node. There are two types of list: singly linked list and doubly linked list.
* **Graph**: A graph is a data structure composed of a set of vertices and edges, which form connections between the vertices. There exist many different types of graphs. This structure can be represented using:
* Edge Lists
* Adjacency Lists
* Adjacency Matrix
* Incidence Matrix

The representation that is going to be used to the development of this project are: Adjacency List and Adjacency Matrix

* **DFS**: (Depth first search) It is a search algorithm with the function of making a tour through all the vertices of a graph and searching for information. The technique implemented by the algorithm is to seek the greatest depth and when it reaches it begins to go back.
* **BFS:** (Breadth first search) It is a search algorithm with the function of searching and traversing a graph. It starts at the root and visits neighboring nodes. It is usually most useful when the shortest path between nodes in the network is needed.
* **Dijkstra:** The algorithm of Dijkstra, in graph theory, is intended to find the shortest possible path between two nodes in a graph. It maintains a list of visited nodes and assigns an initial distance to each one. During each iteration, it selects the node with the shortest distance, marks it as visited, and updates the tentative distances to its neighboring nodes.

***Third Step: Search Creative Solutions***

In this step we are going to consider as many util possibilities that we can find to give a solution for the problem to solve. Starting from the objective to develop an application that has all the functionalities, we need to find the best alternative to the software.

* **1- Implementation of a graph:** We can represent the board and the connections implementing a graph. The vertex can be the slots of the board and the edges are the possible ways. Also, based on this structure we can implement some of the searching algorithms for verifying the ways from the source to the drain. This makes this solution very efficient to verify the connectivity and allow us to represent complex relations between the board and the pipes.
* **2- Implement a doubly linked list:** We can represent each node as a slot of the board and the connections as the possible ways. This one is the simplest option because we only need to create methods that allow the insertion of pipes, verification and simulation based on recursivity.
* **3- Implement a bidimensional matrix:** This option makes it possible to create an array of two dimensions (columns and rows) and store the pipes in the positions allowed in the board (slots). We need to develop methods to insert, verify and simulate based on recursivity and double paths in the matrix.

***Fourth Step: Preliminary Designs***

Since we have already discussed and analyzed in detail the 3 previous solutions and described their possible functionalities, we decided to discard the third alternative ***"Implement a two-dimensional matrix"*** since it generates a high time complexity due to the multiple paths that have to be used in the insertion and search methods. In addition, it has a high memory consumption, since it is a predefined 8x8 board.

In this way, we have 2 solutions left:

**1- Implement a graph:**

* We can use efficient algorithms for searching and verify
* The solution allow us to represent complex connections between the nodes
* It is harder to develop this structure in comparison with other solutions
* Allow the application of advanced algorithms that makes the software more complete
* Sometimes can consume more computational resources

**2- Implement a doubly linked list:**

* Its flexible to create the methods to add and delete pipes
* Offers a mostly efficient use of memory
* It is easy to execute validations and methods for flow control.
* The access to the information in the list is slow
* Allows navigation in both directions
* It is adaptable to changes
* Most temporal complexity

***Fifth Step: Selection of the Best Solution***

Based on the previous analysis, we have two possible solutions to the problem. For choosing the best option we need to set different criteria to evaluate the advantages and disadvantages. The criteria are:

* **Criteria A: Program completeness**

[1] Not complete

[2] Partially completed

[3] Very complete

* **Criteria B: Efficiency**

[1] Inefficient

[2] Low efficient

[3] Normal

[4] Efficient

[5] Very efficient

* **Criteria C: Ease of implementation**

[1] Hard to implement.

[2] Medium difficulty to implement.

[3] Easy to implement.

* **Criteria D: Adaptability for the complexity**

[1] Not adaptable

[2] Low adaptability

[2] Medium adaptability

[4] High adaptability

***EVALUATION:*** *Now we need to rate the solutions depending on the previous criteria*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ***Solution*** | ***Criteria A*** | ***Criteria B*** | ***Criteria C*** | ***Criteria D*** | ***Total*** |
| ***Implement a graph*** | ***3*** | ***5*** | ***2*** | ***4*** | ***14*** |
| ***Implement a doubly linked list*** | ***2*** | ***3*** | ***3*** | ***2*** | ***10*** |

Since we have evaluated both solutions and implemented the criteria by way of evaluation, we can conclude from the results that the most optimal solution is the first one, implementing a graph, since this structure allows us to create efficient algorithms for both the search and the insertion of information into the graph. It also gives us flexibility in the long run if the problem increases in difficulty. It also offers greater ease of handling connections.

***Sixth Step: Preparation of Reports and Specifications***

**Specifications:**

As it was seen before, it’s required to develop a pipe mania game, which starts with the creation of an 8x8 board with a randomly placed source of water and drain. The user should be able to select the type of pipe they wish to use and place it on the given coordinates on the board. Also, it should allow the user to, whenever they’re ready, simulate the waterflow through the pipes to confirm if their solution is working. Finally, the program is also required to calculate a score for the game. We can summarize this information, adding the best solution that was already decided, in this way:

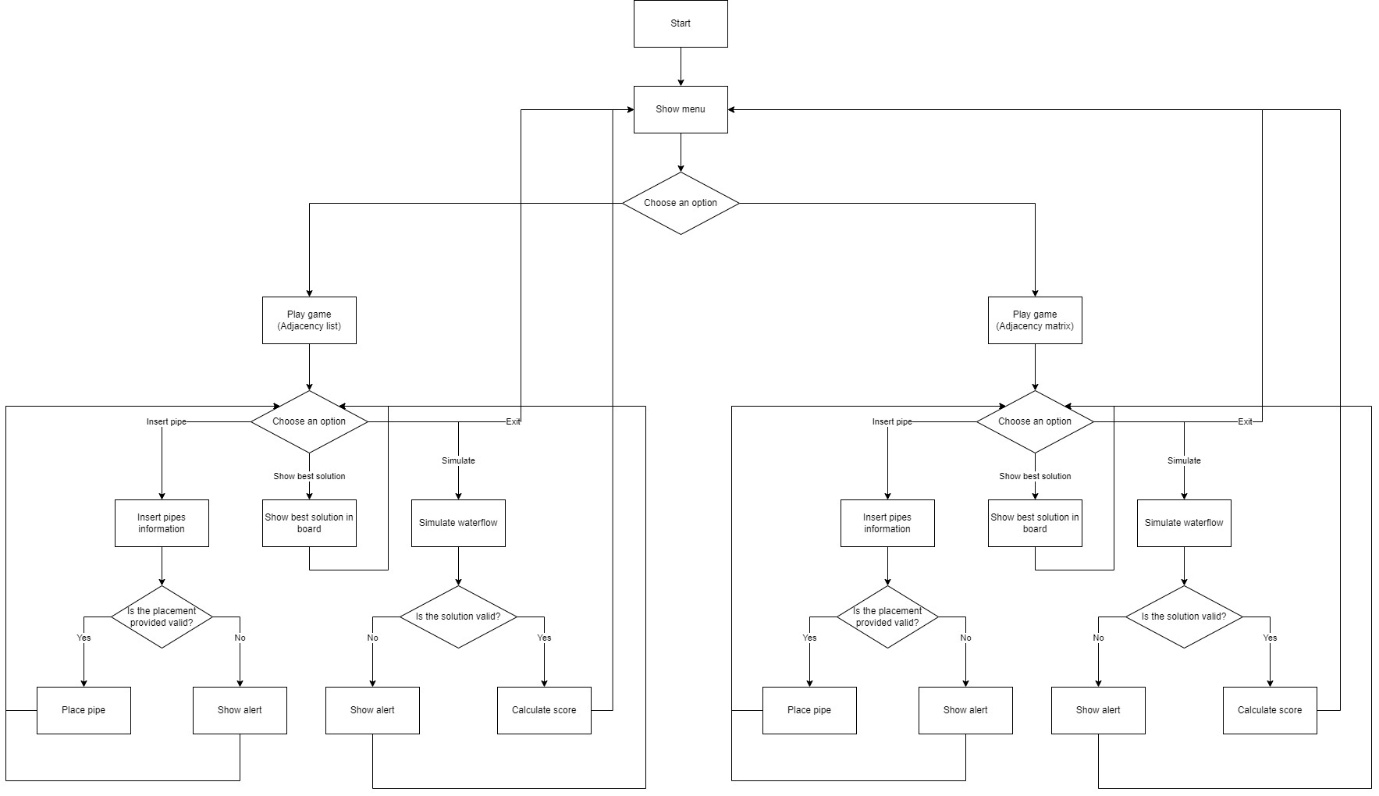
Problem: Implementing a Pipe Mania game using graphs in both of its forms (Adjacency list and adjacency matrix)

Inputs: The different inputs depend on the functionality that the user requires to perform and could normally be to press a button or to provide the coordinates to the pipes.

Outputs: The outputs depend on the functionality that the user did and could normally be an alert or a change in the game board or scenery.

Considerations:

* The coordinate provided for the pipe must be a valid coordinate, within the range of the board.
* The type of pipe selected must be a valid type of pipe.
* There must be only one path from the water source to the drain.
* The user must not place pipes next to their path from the water source to the drain, as it could cause their solution to count as not valid



***Pseudocode:***

To prepare a software solution, is necessary to first elaborate a pseudocode with the main classes that are going to be implemented in the program, as the pseudocode allows to make de coding process easier. The classes that are going to be first implemented into pseudocode, are both the generic graphs, in adjacency list and adjacency matrix, as well as the controllers for each game screen and some of the model classes.

In the model package, there will be:

Class: AdjacencyListGraph<T>

- adjacencyList: ArrayList<Vertex<T>>

Constructor AdjacencyListGraph():

adjacencyList = new ArrayList<>()

Method: getAdjacencyList() -> ArrayList<Vertex<T>>

Return adjacencyList

Method: setAdjacencyList(ArrayList<Vertex<T>> adjacencyList)

Set this.adjacencyList to adjacencyList

Method: addVertex(contentN: T) -> bool

If contentN is null:

Return false

result = false

newVertex = Create a new Vertex with contentN and an empty ArrayList

If adjacencyList.add(newVertex):

result = true

Return result

Method: addEdge(v1: Vertex<T>, v2: Vertex<T>) -> bool

If searchValue(v1) or searchValue(v2):

Add v2 to v1's adjacencyListVertex

Add v1 to v2's adjacencyListVertex

Return true

Return false

Method: searchValue(node: Vertex<T>) -> bool

cont = node.getContent()

For each ver in adjacencyList:

If ver.getContent() equals cont:

Return true

Return false

Method: searchVertex(content: T) -> Vertex<T>

found = null

For each aux in adjacencyList:

If aux.getContent() equals content:

found = aux

Return found

Return found

Method: deleteVertex(contentN: T) -> bool

vertexSearching = searchVertex(contentN)

result = false

If vertexSearching is not null:

For each vertex in adjacencyList:

If vertex.adjacencyListVertex contains vertexSearching:

Remove vertexSearching from vertex.adjacencyListVertex

Remove vertexSearching from adjacencyList

result = true

Return result

Method: toString() -> string

sb = Create a new StringBuilder

Append "Graph Information: \n" to sb

Append "Vertices: \n" to sb

For each vertex in adjacencyList:

Append vertex.getContent() + "\n" to sb

Append "Edges: \n" to sb

For each vertex in adjacencyList:

For each adjacent in vertex.adjacencyListVertex:

Append vertex.getContent() + " - " + adjacent.getContent() + "\n" to sb

Return sb.toString()

Method: easierSolutionDijkstra(source: Vertex<T>, drain: Vertex<T>) -> List<Vertex<T>>

distance = Create a new HashMap<Vertex<T>, Integer>

predecessor = Create a new HashMap<Vertex<T>, Vertex<T>>

minHeap = Create a new PriorityQueue<Vertex<T>>(Comparator.comparingInt(distance::get))

distance.put(source, 0)

minHeap.add(source)

While minHeap is not empty:

currentVertex = minHeap.poll()

For each neighbor in currentVertex.getAdjacencyListVertex():

newDistance = distance.get(currentVertex) + 1

If neighbor is not in distance or newDistance < distance.get(neighbor):

distance.put(neighbor, newDistance)

predecessor.put(neighbor, currentVertex)

minHeap.add(neighbor)

Return reconstructPath(source, drain, predecessor)

Method: reconstructPath(source: Vertex<T>, drain: Vertex<T>, predecessor: HashMap<Vertex<T>, Vertex<T>>) -> List<Vertex<T>>

path = Create a new ArrayList<Vertex<T>>

current = drain

While current is not null and current is not source:

Add current to path

current = predecessor.get(current)

If current is source:

Add source to path

Reverse path

Else:

Clear path

Return path

Method: searchValueIndex(index: int) -> Vertex<T>

Return adjacencyList.get(index)

Class: MatrixGraph<T>

- maxVertex: int

- vertexList: ArrayList<Vertex<T>>

- matrixAdj: int[][]

Constructor MatrixGraph(maxVertex: int):

Set this.maxVertex to maxVertex

vertexList = Create a new ArrayList<Vertex<T>>

matrixAdj = Create a new int[maxVertex][maxVertex]

Call initializeMatrix()

Method: initializeMatrix()

For i from 0 to maxVertex - 1:

For j from 0 to maxVertex - 1:

Set matrixAdj[i][j] to 0

Method: addVertex(content: T) -> bool

If content is null:

Return false

newVertex = Create a new Vertex<T> with content

If vertexList.add(newVertex):

Return true

Return false

Method: addEdge(i: int, j: int) -> bool

If vertexList is not empty:

Set matrixAdj[i][j] to 1

Return true

Return false

Method: returnIndex(v1: Vertex<T>) -> int

For i from 0 to vertexList.size() - 1:

If vertexList.get(i).getContent() equals v1.getContent():

Return i

Return -1

Method: indexVertex(i: int) -> Vertex<T>

If vertexList is not empty and vertexList.get(i) is not null:

Return vertexList.get(i)

Return null

Method: searchVertex(content: T) -> Vertex<T>

found = null

For each aux in vertexList:

If aux.getContent() equals content:

found = aux

Return found

Return found

Method: deleteVertex(content: T) -> bool

vertex = searchVertex(content)

If vertex is not null:

indexToRemove = returnIndex(vertex)

For i from 0 to maxVertex - 1:

Set matrixAdj[indexToRemove][i] to 0

Set matrixAdj[i][indexToRemove] to 0

If vertexList.remove(vertex):

Return true

Return false

Method: getMaxVertex() -> int

Return maxVertex

Method: setMaxVertex(maxVertex: int)

Set this.maxVertex to maxVertex

Method: getVertexList() -> ArrayList<Vertex<T>>

Return vertexList

Method: setVertexList(vertexList: ArrayList<Vertex<T>>)

Set this.vertexList to vertexList

Method: getMatrixAdj() -> int[][]

Return matrixAdj

Method: setMatrixAdj(matrixAdj: int[][])

Set this.matrixAdj to matrixAdj

Method: printMatrix() -> string

msg = ""

For each vertex in vertexList:

msg += vertex.getContent() + " "

msg += "\n"

For i from 0 to maxVertex - 1:

msg += vertexList.get(i).getContent() + " "

For j from 0 to maxVertex - 1:

msg += matrixAdj[i][j] + " "

msg += "\n"

Return msg

Method: dijkstra(source: Vertex<T>, drain: Vertex<T>) -> List<Vertex<T>>

distance = Create a new HashMap<Vertex<T>, Integer>

predecessor = Create a new HashMap<Vertex<T>, Vertex<T>>

settled = Create a new HashSet<Vertex<T>>

distance.put(source, 0)

While true:

current = getClosestVertex(distance, settled)

If current is null:

Break

settled.add(current)

For each neighbor in getNeighbors(current):

If neighbor is not in settled:

newDistance = distance.get(current) + 1

If not distance.containsKey(neighbor) or newDistance < distance.get(neighbor):

distance.put(neighbor, newDistance)

predecessor.put(neighbor, current)

Return reconstructPath(source, drain, predecessor)

Method: getClosestVertex(distance: Map<Vertex<T>, Integer>, settled: Set<Vertex<T>>) -> Vertex<T>

closestVertex = null

minDistance = Integer.MAX\_VALUE

For each entry in distance:

vertex = entry.getKey()

dist = entry.getValue()

If vertex is not in settled and dist < minDistance:

minDistance = dist

closestVertex = vertex

Return closestVertex

Method: getNeighbors(vertex: Vertex<T>) -> List<Vertex<T>>

neighbors = Create a new ArrayList<Vertex<T>>

vertexIndex = getVertexList().indexOf(vertex)

For i from 0 to getMaxVertex() - 1:

If matrixAdj[vertexIndex][i] equals 1:

neighbors.add(indexVertex(i))

Return neighbors

Method: reconstructPath(source: Vertex<T>, drain: Vertex<T>, predecessor: Map<Vertex<T>, Vertex<T>>) -> List<Vertex<T>>

path = Create a new ArrayList<Vertex<T>>

current = drain

While current is not null and current is not source:

Add current to path

current = predecessor.get(current)

If current is source:

Add source to path

Reverse path

Else:

Clear path

Return path

Class: Vertex<T>

- content: T

- adjacencyListVertex: ArrayList<Vertex<T>>

Constructor Vertex(cont: T, adjVertex: ArrayList<Vertex<T>>):

Set this.content to cont

Set this.adjacencyListVertex to adjVertex

Constructor Vertex(cont: T):

Set this.content to cont

Method: getAdjacencyListVertex() -> ArrayList<Vertex<T>>

Return adjacencyListVertex

Method: setAdjacencyListVertex(adjacencyListVertex: ArrayList<Vertex<T>>)

Set this.adjacencyListVertex to adjacencyListVertex

Method: getContent() -> T

Return content

Method: setContent(content: T)

Set this.content to content

Class: Pipe

- pipeType: PipeType

Constructor Pipe(pipeType: PipeType):

Set this.pipeType to pipeType

Method: getPipeType() -> PipeType

Return pipeType

Method: setPipeType(pipeType: PipeType)

Set this.pipeType to pipeType

Method: toString() -> string

msg = ""

If pipeType equals PipeType.CIRCULAR:

msg = "°"

Else If pipeType equals PipeType.DRAIN:

msg = "D"

Else If pipeType equals PipeType.HORIZONTAL:

msg = "="

Else If pipeType equals PipeType.VERTICAL:

msg = "||"

Else If pipeType equals PipeType.WATERSOURCE:

msg = "F"

Return msg

Class: Box

- canva: Canvas

- graphicsContext: GraphicsContext

- content: Pipe

- shortActivate: boolean

- waterSourceImage: Image

- drainImage: Image

- horizontalImage: Image

- verticalImage: Image

- circularImage: Image

- defaultImage: Image

- shortActivateImage: Image

Constructor Box(content: Pipe, canva: Canvas):

Set this.canva to canva

Set this.graphicsContext to this.canva.getGraphicsContext2D()

Set this.content to content

Set this.shortActivate to false

Method: isShortActivate() -> boolean

Return shortActivate

Method: setShortActivate(shortActivate: boolean)

Set this.shortActivate to shortActivate

Method: getContent() -> Pipe

Return content

Method: setContent(content: Pipe)

Set this.content to content

Method: loadImage(imagePath: String) -> Image

Try (with resources):

Create an InputStream using Box.class.getResourceAsStream(imagePath)

Return a new Image created from the InputStream

Catch (Exception e):

Print the stack trace

Return null

Method: paint(x: int, y: int, size: int, box: Box)

If box is not null:

Get the pipeType from box's content, or null if content is null

If pipeType is PipeType.WATERSOURCE:

Call drawImage(x, y, waterSourceImage)

Else If pipeType is PipeType.DRAIN:

Call drawImage(x, y, drainImage)

Else If pipeType is PipeType.HORIZONTAL:

Call drawImage(x, y, horizontalImage)

Else If pipeType is PipeType.VERTICAL:

Call drawImage(x, y, verticalImage)

Else If pipeType is PipeType.CIRCULAR:

Call drawImage(x, y, circularImage)

Else If shortActivate is true:

Call drawImage(x, y, shortActivateImage)

Else:

Call drawImage(x, y, defaultImage)

Method: drawImage(x: int, y: int, image: Image)

If image is not null:

Call graphicsContext.drawImage(image, x, y)

Method: toString() -> string

Return content.toString()

On the other hand, the control package will have:  
  
Class: BaseControl

- canvas: Canvas

- graphicsContext: GraphicsContext

- adjacencyGraph: AdjacencyListGraph<Box>

- pipesUsed: int

- score: long

- random: Random

- startTime: LocalDateTime

- endTime: LocalDateTime

- watersourceIndex: int

- drainIndex: int

Constructor BaseControl(canvas: Canvas):

Set this.canvas to canvas

Set this.graphicsContext to this.canvas.getGraphicsContext2D()

Set this.startTime to LocalDateTime.now()

Set this.endTime to null

Create a new AdjacencyListGraph<Box> and set it to this.adjacencyGraph

Set pipesUsed to 0

Set score to 0

Create a new Random and set it to this.random

Call addFullBoxes()

Method: finalizarPartida()

Set this.endTime to LocalDateTime.now()

Method: getGameLenghtInSeconds() -> long

If startTime is not null and endTime is not null:

Calculate the duration between startTime and endTime

Return the duration in seconds

Else:

Return -1

Method: calculateScore() -> double

Calculate the score based on the formula (100 - pipesUsed) \* 10 - getGameLenghtInSeconds()

Set this.score to the calculated score

Return the calculated score

Method: paint()

Call drawBoard()

Method: setRandomNodeDrain() -> int

Return a random integer between 0 and 63

Method: setRandomNodeSource() -> int

Return a random integer between 0 and 63

Method: addFullBoxes()

Initialize sourceRow, sourceCol, drainRow, drainCol to 0

Do:

Generate a random source and drain index

Calculate sourceRow, sourceCol, drainRow, drainCol from the indices

While the absolute difference between sourceRow and drainRow is less than 3 or the absolute difference between sourceCol and drainCol is less than 3

Set source to coordinadeToIdex(sourceRow, sourceCol)

Set drain to coordinadeToIdex(drainRow, drainCol)

Loop over indices from 0 to 63:

If the index is equal to source:

Add a new Box with a Pipe of type WATERSOURCE to adjacencyGraph and set watersourceIndex to the index

Else If the index is equal to drain:

Add a new Box with a Pipe of type DRAIN to adjacencyGraph and set drainIndex to the index

Else:

Add a new Box with null content to adjacencyGraph

Loop over indices from 0 to size of adjacencyGraph's adjacencyList:

Get the current vertex

Calculate row and col from the index

If col is less than 7:

Get the vertex to the right and add it to the adjacencyListVertex of the current vertex

If col is greater than 0:

Get the vertex to the left and add it to the adjacencyListVertex of the current vertex

If row is less than 7:

Get the vertex below and add it to the adjacencyListVertex of the current vertex

If row is greater than 0:

Get the vertex above and add it to the adjacencyListVertex of the current vertex

Method: drawBoard()

Clear the graphicsContext

Set cellSize to 64

Set numRows to 8

Set numCols to 8

Loop over rows from 0 to numRows:

Loop over columns from 0 to numCols:

Calculate x and y from the row and column

Get the vertex from adjacencyGraph based on the index (row \* numCols + col)

Get the content from the vertex and cast it to Box

If box is not null:

Call box.paint(x, y, cellSize, box)

Method: insertPipe(x: int, y: int, pipe: int)

Get the coordinate index from x and y using coordinadeToIdex

Loop over indices from 0 to size of adjacencyGraph's adjacencyList:

If the index is equal to the coordinate index:

Get the content from the current vertex

If the content is not null and is a water source or drain, show an alert

Else, set the content to a new Pipe with the given pipe type and increment pipesUsed

Method: simulate() -> boolean

Get the water source and drain vertices from adjacencyGraph

If simulateBFS(water source, drain) returns true, return true, else return false

Method: simulateBFS(fountVertex: Vertex<Box>, drainVertex: Vertex<Box>) -> boolean

Create a queue and a set of visited vertices

Add fountVertex to the queue and to visitedVertex

While the queue is not empty:

Get the current vertex from the queue

Get the content from the current vertex and cast it to Box

If the content is a Pipe:

Get the pipe type from the content

If validateDrain() and the pipe type is DRAIN, return true

Loop over adjacent vertices to the current vertex:

If the adjacent vertex content is not null:

Get the pipe type from the adjacent vertex content

If the pipe type is VERTICAL and validateVertical is true, add the adjacent vertex to the queue and visitedVertex

If the pipe type is HORIZONTAL and validateHorizontal is true, add the adjacent vertex to the queue and visitedVertex

If the pipe type is not VERTICAL or HORIZONTAL, add the adjacent vertex to the queue and visitedVertex

Return false

Method: validateDrain() -> boolean

Get the drain vertex from adjacencyGraph

Loop over adjacent vertices to the drain vertex:

If the adjacent vertex content is a Pipe and its type is CIRCULAR, return true

Return true

Method: validateVertical(value: Vertex<Box>) -> boolean

Loop over adjacent vertices to the given vertex:

If the adjacent vertex content is a Pipe and its type is HORIZONTAL, return false

Return true

Method: validateHorizontal(value: Vertex<Box>) -> boolean

Loop over adjacent vertices to the given vertex:

If the adjacent vertex content is a Pipe and its type is VERTICAL, return false

Return true

Method: earierSolutionActiveButton() -> boolean

Get the water source and drain vertices from adjacencyGraph

Call showShorterSolution(waterSource, drain)

Return false

Method: showShorterSolution(source: Vertex<Box>, drain: Vertex<Box>)

Get the shortest path using easierSolutionDijkstra from adjacencyGraph

Loop over vertices in the shortest path:

Loop over vertices in adjacencyGraph's adjacencyList:

If the path vertex is equal to the graph vertex:

Get the content from the graph vertex and cast it to Box

Set shortActivate to true for the box

Method: coordinadeToIdex(x: int, y: int) -> int

Return x \* 8 + y

Method: getWatersourceIndex() -> int

Return watersourceIndex

Method: setWatersourceIndex(watersourceIndex: int)

Set this.watersourceIndex to watersourceIndex

Method: getDrainIndex() -> int

Return drainIndex

Method: setDrainIndex(drainIndex: int)

Set this.drainIndex to drainIndex

Class: MatrixGameController

- canvas: Canvas

- graphicsContext: GraphicsContext

- matrixGraph: MatrixGraph<Box>

- pipesUsed: int

- score: long

- random: Random

- startTime: LocalDateTime

- endTime: LocalDateTime

- watersourceIndex: int

- drainIndex: int

Constructor MatrixGameController(ca: Canvas):

Set this.canvas to ca

Set this.graphicsContext to this.canvas.getGraphicsContext2D()

Set this.startTime to LocalDateTime.now()

Set this.endTime to null

Create a new MatrixGraph<Box> with a capacity of 64 and set it to this.matrixGraph

Set pipesUsed to 0

Set score to 0

Create a new Random and set it to this.random

Call addFullBoxes()

Method: finalizarPartida()

Set this.endTime to LocalDateTime.now()

Method: getGameLenghtInSeconds() -> long

If startTime is not null and endTime is not null:

Calculate the duration between startTime and endTime

Return the duration in seconds

Else:

Return -1

Method: calculateScore() -> double

Calculate the score based on the formula (100 - pipesUsed) \* 10 - getGameLenghtInSeconds()

Set this.score to the calculated score

Return the calculated score

Method: paint()

Call drawBoard()

Method: drawBoard()

Clear the graphicsContext

Set cellSize to 64

Set numRows to 8

Set numCols to 8

Loop over rows from 0 to numRows:

Loop over columns from 0 to numCols:

Calculate x and y from the row and column

Get the vertex from matrixGraph's vertexList based on the index (row \* numCols + col)

Get the content from the vertex and cast it to Box

If box is not null:

Call box.paint(x, y, cellSize, box)

Method: setRandomNodeDrain() -> int

Return a random integer between 0 and 63

Method: setRandomNodeSource() -> int

Return a random integer between 0 and 63

Method: coordinadeToIdex(x: int, y: int) -> int

Return x \* 8 + y

Method: addFullBoxes()

Try:

int counter = 0

int sourceRow, sourceCol, drainRow, drainCol

do:

Generate random source and drain indices

Calculate sourceRow, sourceCol, drainRow, drainCol from the indices

While the absolute difference between sourceRow and drainRow is less than 2 or the absolute difference between sourceCol and drainCol is less than 2

int source = coordinadeToIdex(sourceRow, sourceCol)

int drain = coordinadeToIdex(drainRow, drainCol)

Loop over indices from 0 to 63:

If the index is equal to source:

Add a new Box with a Pipe of type WATERSOURCE to matrixGraph and set watersourceIndex to the index

Else If the index is equal to drain:

Add a new Box with a Pipe of type DRAIN to matrixGraph and set drainIndex to the index

Else:

Add a new Box with null content to matrixGraph

Loop over indices from 0 to 63:

int row = index / 8

int col = index % 8

If col < 7:

Add an edge from index to index + 1 in matrixGraph

Increment counter

If col > 0:

Add an edge from index to index - 1 in matrixGraph

Increment counter

If row < 7:

Add an edge from index to index + 8 in matrixGraph

Increment counter

If row > 0:

Add an edge from index to index - 8 in matrixGraph

Increment counter

Print counter

Catch Exception:

Print stack trace

Method: insertPipe(x: int, y: int, pipe: int)

Get the coordinate index from x and y using coordinadeToIdex

Loop over indices from 0 to size of matrixGraph's vertexList:

If the index is equal to the coordinate index:

Get the content from the current vertex

If the content is not null and is a water source or drain, show an alert

Else, set the content to a new Pipe with the given pipe type and increment pipesUsed

Method: simulate() -> boolean

Get the water source and drain vertices from matrixGraph's vertexList

If simulateBFS(water source, drain) returns true, return true, else return false

Method: simulateBFS(fountVertex: Vertex<Box>, drainVertex: Vertex<Box>) -> boolean

Create a queue and a set of visited vertices

Add fountVertex to the queue and to visitedVertex

While the queue is not empty:

Get the current vertex from the queue

Get the content from the current vertex and cast it to Box

If the content is not null and is a Pipe:

Get the pipe type from the content

If validateDrain() and the pipe type is DRAIN, return true

Loop over adjacent vertices to the current vertex:

If the adjacent vertex content is not null:

Get the pipe type from the adjacent vertex content

If the pipe type is VERTICAL and validateVertical is true, add the adjacent vertex to the queue and visitedVertex

If the pipe type is HORIZONTAL and validateHorizontal is true, add the adjacent vertex to the queue and visitedVertex

If the pipe type is not VERTICAL or HORIZONTAL, add the adjacent vertex to the queue and visitedVertex

Return false

Method: validateDrain() -> boolean

Get the drain index from getDrainIndex

***Seven Step: Implementation***

Finally, the implementation of the game can be found completed in this respository :

https://github.com/tobivalens/Second-Integrative-Task-DIS.git