Daniel Alejandro Fernández Robles
A00354694
Camilo Enriquez Delgado
A00354532

Step 1. Identification of the problem.

The specific needs of the problematic situation, symptoms and conditions for resolution are listed below.

Identification of needs and symptoms:

Pac-Man needs to eat Pac-Dots, fruits, energizers and blue ghosts to achieve the greatest possible score.

The map should be modeled using a graph.

A shortest path algorithm must define the behavior of the ghosts.

The game must not be delayed so anyone who plays can focus in the gameplay.

Definition of the problem:

Design and program a Pac-Man game that emulates a wide variety of the functions and behavior of the original game by Namco.

Requirements:

Functional requirements:

Name	FR.#1. Find the shortest path between two points on the map
Summarize	The program will allow to find the shortest path between two tiles of the map using Floy-Warshall algorithm
Input	● The source Vertex of the path. ● The target Vertex of the path.
Output	A list of the vertices that conform the shortest path from source Vertex to target Vertex.

Name	FR.#2. Move pacman through the maze
Summarize	Move Pacman according to the direction preselected by the user if there is no wall interposing.
Input	None

Output	Pacman has moved a pixel forward if no wall was in his way.

Name	FR.#3. Move a ghost through the maze
Summarize	Ghosts move according to the tile that has been determined as their target through the shortest path. It is also taken into account if they are in chase, scatter or frightened mode, on which the speeds of all the characters depend.
Input	● An object of type Ghost representing the ghost to be moved.
Output	The ghost has moved a pixel forward its target.

Name	FR.#4. Register the player score when a high score is achieved
Summarize	When Pacman has lost all his lives along the game and the actual score is higher than one of the first ten positions, a little window will handle of register the name of this player, which cannot exceed the four characters.
Input	● The name of the player (cannot exceed the four characters).
Output	The corresponding place of the actual player in the leaderboard.

Name	FR.#5. Show the leaderboard
Summarize	When the player presses the "Higher Scores" button a TableView with all the previous higher scores will be displayed as long as there is a file with these previous matches inside it. By default the TableView is displayed empty.
Input	● A file with all the previous matches if these exist.
Output	A TableView with all the current scores.

Name	FR.#6. Change direction of travel
Summarize	Allows to request a change in the direction of travel of pacman. It is only set as the direction of pacman if it is possible, that is, if there is not a wall in the direction the user wants to go.

Input	• An object of type Direction representing the requested direction of travel.
Output	The direction has been set as requested and will be updated when pacman reaches a tile.

Name	**FR.#7. Earn a life**
Summarize	Increase in one the total lives of Pacman if he reaches a score divisible by 5000. It is only allowed when pacman has less than 5 lives left, as this is the maximum amount of lives.
Input	None
Output	Pacman earned a life.

Name	**FR.#8. Show information and instructions about the game**
Summarize	Allows to show relevant information about the game and instructions about the controls or the mechanics of the game.
Input	None
Output	Information displayed.

| Name	**FR.#9. Level up when Pacman eats all the dots in the maze.**
Summarize	The stage must lead to next when pacman has eaten all the dots in it.
Input	None
Output	The staged has been updated.

Non-Functional requirements:

Name	**NFR.#1. Design the GUI using JavaFX**
Summarize	The graphical user interface will be made in JavaFX and styled using CSS.

| Name	NFR.#2. Make the game works either using an adjacency matrix or an adjacency list representation of the graph that models the maze.
Summarize	The graph that represents the maze can be switch between two types of representations, an Adjacency matrix and list.

Step 2. Gathering needed information.

Node or Vertex:

These are the most important components in any graph. Nodes are entities whose relationships are expressed using edges.
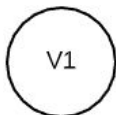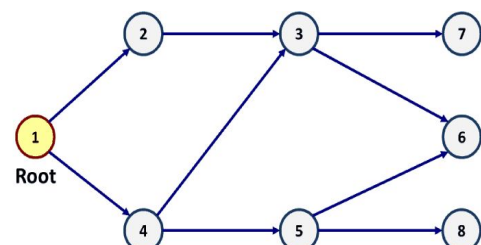

Edges:

Edges(also called an "arcs") are the components that are used to represent the relationships between various pairs of nodes in a graph. An edge between two nodes expresses a one-way or two-way relationship between the nodes. They're ordered pairs of the form (u, v). The pair is ordered because (u, v) is not the same as (v, u) in the case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

| Edges in an undirected graph		Edges in a directed graph(di-graph)	
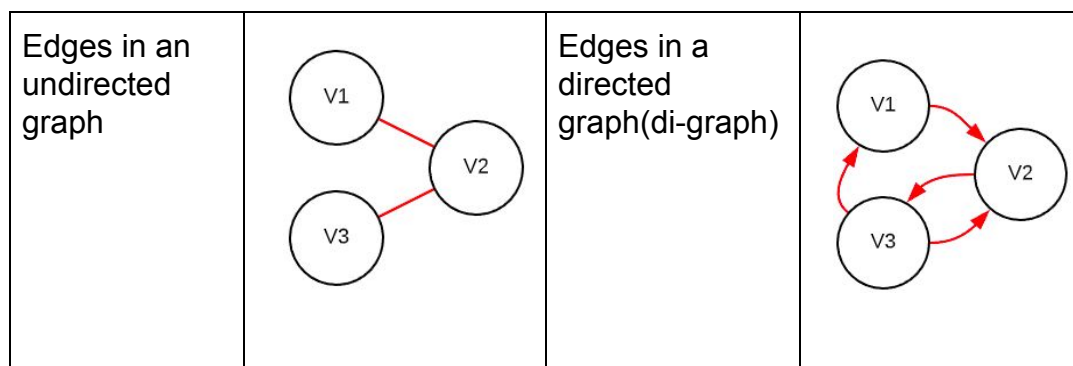
Types of nodes:

Root node: The root node is the ancestor of all other nodes in a graph. It does not have any ancestors. Each graph consists of exactly one

root node. Generally, you must start traversing a graph from the root node.

Leaf nodes: In a graph, leaf nodes represent the nodes that do not have any successors. These nodes only have ancestor nodes. They can have any number of incoming edges but they will not have any outgoing edges.
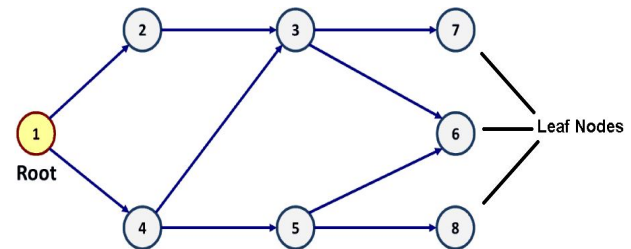
Graph:

They are discrete structures that consist of vertices and edges that connect these vertices together. With those definitions in hand we can formally define a graph. A graph can be represented by G where G=(V,E). For the graph G, V is a set of vertices and E is a set of edges. Each edge is a tuple (v,w) where w,v∈V. We can add a third component to the edge tuple to represent a weight.
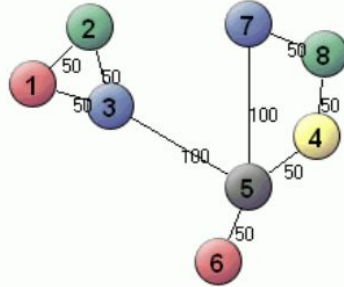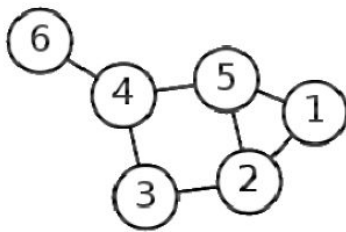
About the game:

Pac-Man is a maze arcade game developed and released by Namco in 1980. The player controls Pac-Man, who must eat all the dots inside an enclosed maze while avoiding four colored ghosts. Eating large flashing dots called power pellets causes the ghosts to turn blue, allowing Pac-Man to eat them for bonus points.

The player navigates Pac-Man through a maze with no dead ends. The maze is filled with Pac-Dots, and includes four roving multi-colored ghosts: Blinky, Pinky, Inky, and Clyde. There is a passageway from the left side of the screen to the right side, one energizer in each of the four quadrants, and bonus fruits that appear in each level.

The objective of the game is to accumulate as many points as possible by eating dots, fruits, and blue ghosts. When all of the dots in a stage are eaten, that stage is completed, and the player will advance to the next. The four ghosts roam the maze and chase Pac-Man. If any of the ghosts touches Pac-Man, a life is lost. When all lives have been lost, the game is over. The player begins with three lives, but that amount

can be changed to one, two, or five. The player will receive one extra life bonus after obtaining 10,000 points. The number of points needed for a bonus life can be changed to 15,000 or 20,000 or disabled altogether.

Near the corners of the maze are four flashing energizers that allow Pac-Man to eat the ghosts and earn bonus points. The enemies turn deep blue, reverse direction and move away from Pac-Man, and usually move more slowly. When an enemy is eaten, its eyes return to the center ghost box where the ghost is regenerated in its normal color. The bonus score earned for eating a blue ghost increases exponentially for each consecutive ghost eaten while a single energizer is active: a score of 200 points is scored for eating one ghost, 400 for eating a second ghost, 800 for a third, and 1600 for the fourth. This cycle restarts from 200 points when Pac-Man eats the next energizer. Blue enemies flash white to signal that they are about to return to their normal color and become dangerous again; the length of time the enemies remain vulnerable varies from one stage to the next, generally becoming shorter as the game progresses. In later stages, the enemies begin flashing immediately after an energizer is consumed, without a solid-blue phase; starting at stage nineteen, the ghosts do not become edible at all, but still reverse direction. There are fruits that appear twice per level, directly below the center ghost box; eating one gives 100 to 5,000 points.

Enemy behavior:

| Color | Puckman (original)[16] | | | | | | Pac-Man (English version) | |
	Character (personality)	Translation	Nickname	Translation	Alternate character	Alternate nickname	Character (personality)	Name
Red	Oikake (追いかけ)	Chaser	Akabei (赤ベイ)	Red guy	Urchin	Macky	Shadow	Blinky
Pink	Machibuse (待ち伏せ)	Ambusher	Pinky (ピンキー)	Pink guy	Romp	Micky	Speedy	Pinky
Cyan	Kimagure (気まぐれ)	Fickle	Aosuke (青助)	Blue guy	Stylist	Mucky	Bashful	Inky
Orange	Otoboke (お惚け)	Feigned Ignorance	Guzuta (愚図た)	Slow guy	Crybaby	Mocky	Pokey	Clyde

Ghosts have three mutually-exclusive modes of behavior they can be in during play: chase, scatter, and frightened. Each mode has a different objective/goal to be carried out:

Chase: A ghost's objective in chase mode is to find and capture Pac-Man by hunting him down through the maze. Each ghost exhibits unique behavior when chasing Pac-Man, giving them their different personalities: Blinky (red) is very aggressive and hard to shake once he gets behind you, Pinky (pink) tends to get in front of you and cut you off, Inky (light blue) is the least predictable of the bunch, and Clyde (orange) seems to do his own thing and stay out of the way.

Scatter: In scatter mode, the ghosts give up the chase for a few seconds and head for their respective home corners. It is a welcome but brief rest-soon enough, they will revert to chase mode and be after Pac-Man again.

Frightened: Ghosts enter frightened mode whenever Pac-Man eats one of the four energizers located in the far corners of the maze. During the early levels, the ghosts will all turn dark blue (meaning they are vulnerable) and aimlessly wander the maze for

a few seconds. They will flash moments before returning to their previous mode of behavior.

-- When time runs out, they return to the mode they were in before being frightened and the scatter/chase timer resumes. This information is summarized in the following table (all values are in seconds):

Mode	Level 1	Levels 2-4	Levels 5+
Scatter	7	7	5
Chase	20	20	20
Scatter	7	7	5
Chase	20	20	20
Scatter	5	5	5
Chase	20	1033	1037
Scatter	5	1/60	1/60
Chase	indefinite	indefinite	indefinite

Parameters of each level:

Table A.1 — Level Specifications

Level	Bonus Symbol	Bonus Points	Pac-Man Speed	Pac-Man Dots Speed	Ghost Speed	Ghost Tunnel Speed	Elroy 1 Dots Left	Elroy 1 Speed	Elroy 2 Dots Left	Elroy 2 Speed	Fright. Pac-Man Speed	Fright Pac-Man Dots Speed	Fright Ghost Speed	Fright. Time (in sec.)	# of Flashes
1	Cherries	100	80%	71%	75%	40%	20	80%	10	85%	90%	79%	50%	6	5
2	Strawberry	300	90%	79%	85%	45%	30	90%	15	95%	95%	83%	55%	5	5
3	Peach	500	90%	79%	85%	45%	40	90%	20	95%	95%	83%	55%	4	5
4	Peach	500	90%	79%	85%	45%	40	90%	20	95%	95%	83%	55%	3	5
5	Apple	700	100%	87%	95%	50%	40	100%	20	105%	100%	87%	60%	2	5
6	Apple	700	100%	87%	95%	50%	50	100%	25	105%	100%	87%	60%	5	5
7	Grapes	1000	100%	87%	95%	50%	50	100%	25	105%	100%	87%	60%	2	5
8	Grapes	1000	100%	87%	95%	50%	50	100%	25	105%	100%	87%	60%	2	5
9	Galaxian	2000	100%	87%	95%	50%	60	100%	30	105%	100%	87%	60%	1	3
10	Galaxian	2000	100%	87%	95%	50%	60	100%	30	105%	100%	87%	60%	5	5
11	Bell	3000	100%	87%	95%	50%	60	100%	30	105%	100%	87%	60%	2	5
12	Bell	3000	100%	87%	95%	50%	80	100%	40	105%	100%	87%	60%	1	3
13	Key	5000	100%	87%	95%	50%	80	100%	40	105%	100%	87%	60%	1	3
14	Key	5000	100%	87%	95%	50%	80	100%	40	105%	100%	87%	60%	3	5
15	Key	5000	100%	87%	95%	50%	100	100%	50	105%	100%	87%	60%	1	3
16	Key	5000	100%	87%	95%	50%	100	100%	50	105%	100%	87%	60%	1	3
17	Key	5000	100%	87%	95%	50%	100	100%	50	105%	–	–	–	–	–
18	Key	5000	100%	87%	95%	50%	100	100%	50	105%	100%	87%	60%	1	3
19	Key	5000	100%	87%	95%	50%	120	100%	60	105%	–	–	–	–	–
20	Key	5000	100%	87%	95%	50%	120	100%	60	105%	–	–	–	–	–
21+	Key	5000	90%	79%	95%	50%	120	100%	60	105%	–	–	–	–	–

Click here for a more detailed description about the behavior of ghosts and how they look for their targets.
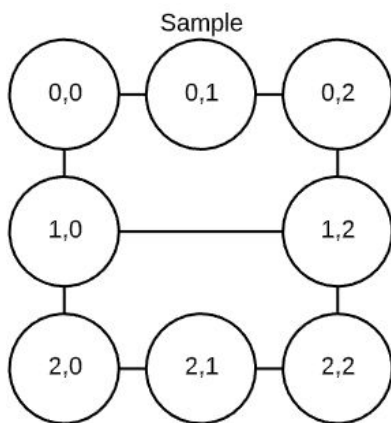
<u>Scatter targets:</u>

Each ghost has a fixed target tile it is trying to reach in scatter mode.
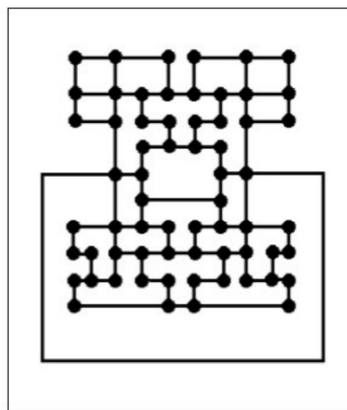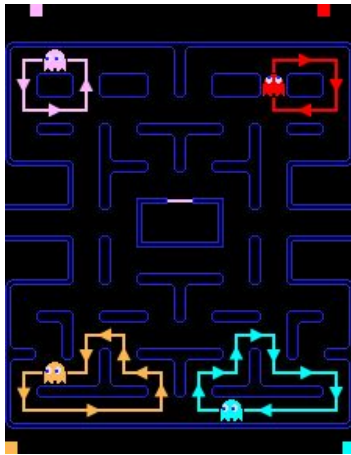
Step 3. Searching for creative solutions.

First option: Use an adjacency matrix to model the map

	0,0	0,1	0,2	1,0	1,2	2,0	2,1	2,2
0,0	0	1	0	1	0	0	0	0
0,1	1	0	1	0	0	0	0	0
0,2	0	1	0	0	1	0	0	0
1,0	1	0	0	0	1	1	0	0
1,2	0	1	1	0	0	0	0	1
2,0	0	0	0	1	0	0	1	0
2,1	0	0	0	0	0	1	0	1
2,2	0	0	0	0	1	0	1	0

This option requires to associate a two-dimensional matrix where each row and column represents a vertex in the graph and the value of a cell(i,j) is the cost of going from vertex i to vertex j, just like the showed previously.

A value of 0 (Although it can also be *infinity*) indicates that there is no edge from vertex i to vertex j, for instance, from (0,1) to (1,0) and integer values greater than 0 indicating the cost of arriving from vertex i to vertex j, for example, from (1,0) to (2,0). The idea here is doing that Pacman ghosts be capable to move through the corridors which the Pac-dots, fruits and energizes are going to be allocated as well.

When any of the characters attempt to travel through one of the boxes marked with "0" (or *infinity*), these should not be able to do it because the value of this box represents a wall or no edge between two referenced vertices. In our case we will use a graph where all the edges will have unit weight, except in the tunnel where the ghosts slow down and therefore there should be a higher cost. Other edges with a high cost would be the ones just up the ghosts' house as ghosts are not allowed to travel them from bottom to top (although they can do it from top to bottom).
Food and the bonus objects are going to be allocated in their respective vertex.

Second option: Use an adjacency list to model the map

|				
0,0 ------>	0,1 ------>	1,0		
0,1 ------>	0,0 ------>	0,2		
0,2 ------>	0,1 ------>	1,2		
1,0 ------>	0,0 ------>	1,2 ------>	2,0	
1,2 ------>	0,2 ------>	1,0 ------>	2,2	
2,0 ------>	1,0 ------>	2,1		
2,1 ------>	2,0 ------>	2,2		
2,2 ------>	1,2 ------>	2,1		

For this option is necessary the use of an array of lists for each vertex where the outcoming list of each vertex contains all the edges to their neighbors. For doing this efficiently we need a list of type dictionary where the keys are the vertices and their values are their adjacent vertices.

If one of the vertices of the list is not associated with another one means that there is no edge between those two vertices (although there can be a path going through more vertices) or there is a "wall" between them. Each edge in the list is associated with a weight that represents the cost of moving in a determinate part of the maze, where the

cost will be the highest for the ghosts when they try to pass through the tunnel or from bottom to top when they are in the exit of their house.

Food and the bonus objects are going to be displayed in their location as the original game being assigned in each respective vertex along all the maze.

Third option: Use a matrix whose components are the map coordinates
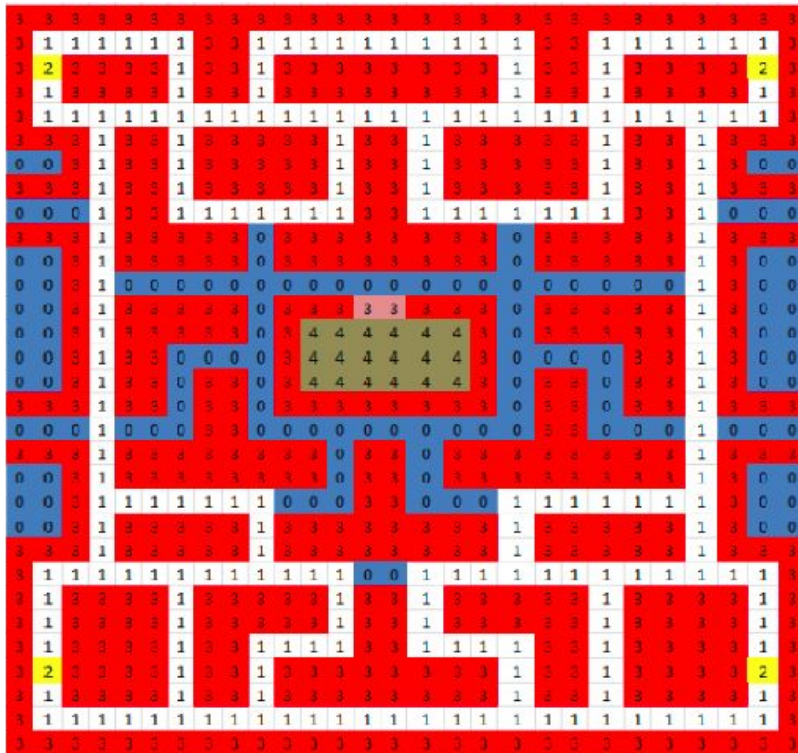
The idea is to have a matrix whose boxes have identifiers of the element that is in that part of the map, for example, in the previous matrix, elements of the map are represented by a number: (0) Corridor without food, (1) Corridor with food, (2) Power-pellets, (3) Wall, (4) House of ghosts.

Characters can move through the boxes marked 0 or 1. When Pacman goes through a box marked 1, the box will change to 0. Characters can be represented by ASCII characters not present on the map, one character for each ghost and another for Pacman.

Step 4. Stepping from ideation to preliminary designs.

Adjacency matrix:

-- Get vertices adjacent to a vertex cost $O(V)$ time since it implies scanning a whole row of the matrix.

-- It uses $O(V^2)$ memory.

-- Tell whether a vertex is connected to another one costs $O(1)$ time since the operation

just implies to look at the value of a particular cell of the matrix.
-- It allows natively the implementation of the Floyd-Warshall algorithm.
-- Kruskal's algorithm is hard to imagine due to the nature of the edges; they only
    contain numbers and no more information

Adjacency list:
-- Get vertices adjacent to a vertex cost O(1) time in the generic implementation but
    if the elements of the vertices are desired to be returned instead of the vertices
    containing them, then the time complexity increases to O(V), just as in the adjacency
    matrix.
-- It uses O(V+E) memory.
-- Tell whether a vertex is connected to another one costs O(V) time with a proper
    implementation such as using hash tables for storing edges with the keys being the
    vertices and the values being a list of edges with the vertex key as the tail.
-- It does not allow natively the implementation of the Floyd-Warshall algorithm.
-- It is easy to think about Kruskal's algorithm since the edges are placed in the same
    list to be able to sort them, then the list has just the necessary information to
    implement the algorithm.

Regular matrix:
-- It is very difficult to design algorithms in this type of structure so one would end up
    choosing to represent it through one of the other two options
-- It is easy to understand graphically.

Step 5. Evaluation and selection of preferred solution.

Criterion A: Memory usage.
    [2] Linear.
    [1] Quadratic.
Criterion B: All graph algorithms supported natively.
    [4] All of them.
    [3] The majority of them.
    [2] Some of them.
    [1] None of them.

Criterion C:Information available.
    [2] Wide variety of resources available to learn how to use the alternative.
    [1] No information or unreliable information.
Criterion D: Easiness to codificate.
    [3] Friendly and simplistic way to code in any programming language.
    [2] Although not that easy to code is still manageable.
    [1] The syntaxis and the code structure are difficult to understand and model.
Criterion E: Average algorithms speed.
    [3] Fast.
    [2] Fast but not so much.
    [1] Unknown.

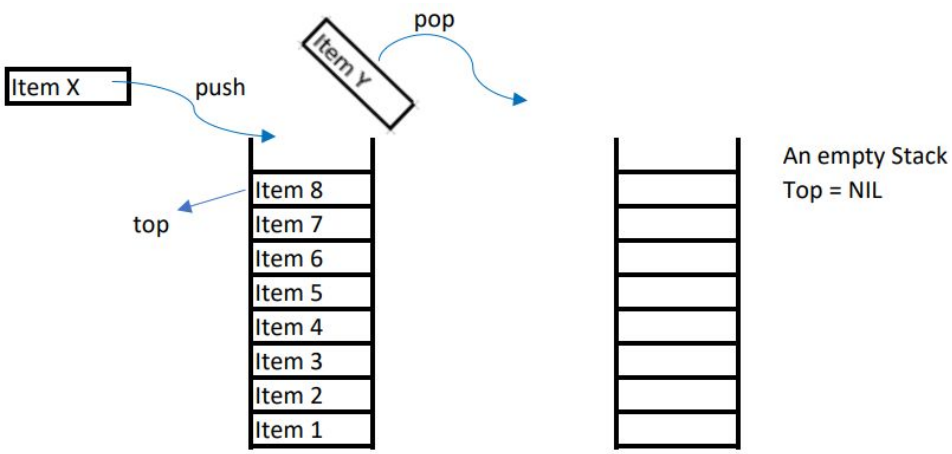|	Criterion A	Criterion B	Criterion C	Criterion D	Criterion E	Total
Adjacency matrix	1	4	2	2	3	12
Adjacency list	2	3	2	3	2	12
Regular matrix	2	1	1	2	1	7

Selection:

The alternative that we will be using is the **adjacency matrix**, since, although it got the same score as the adjacency list, we would end up using more space with the list. This is because we need to use Floyd-Warshall algorithm and we could not figure out how to implement it natively in the list, so we translated the list into a matrix and performed Floyd-Warshall in there.

Step 6. Preparation of reports, plans, and specifications.

Statement of Abstract Data Types

ADT Stack	
Definition	The ADT Stack is a linear sequence of an arbitrary number of items, along with access procedures. Note that sequence does not imply order. The access procedures permit insertion, deletion and retrieval of elements only at one end of the sequence (the "top"). It's because of this that is called a last-in-first-out (or LIFO) list. A stack is either empty, or it consists of a sequence of items.

Object Representation	Item X — push · pop · Item Y top → Item 8 Item 7 Item 6 Item 5 Item 4 Item 3 Item 2 Item 1 An empty Stack Top = NIL
Invariant	(s.isEmpty() == true $\wedge$ top == NIL) $\oplus$ (s = $\{a_n \rightarrow a_{n-1} \rightarrow ... \rightarrow a_1\}$ $\wedge$ top == a_n)

Procedures	**Constructor operations** *createStack()*
	//Creates an empty stack
	//**post:** A stack with no elements inside has been created **Analyzer operations** *top()*
	//Returns the element at the top of the stack but does not
	//modify it.
	//**post:** Returns the top element of the stack. If the stack is
	//empty then its top is NIL and this is the retrieval. *isEmpty()*
	//Determines whether a stack is empty.
	//**post:** Returns TRUE if the top of the stack is NIL and FALSE
	//otherwise. *size()*
	//Returns the size of the stack.
	//**post:** Returns the total number of elements in the stack**.** **Modifier operations** *push(newItem)*
	//Adds newItem to the stack.
	//**post:** newItem is at the top of the stack and the previous top
	//is under it in the same order as before. *pop()*
	//Removes and returns the top element of the stack.
	//**pre:** The stack is not empty.
	//**post:** The top is taken off the stack and the new top is the
	//element that was under the previously returned element. If
	//the stack is empty, it results in an error.

Axioms of the ADT Stack

1. (aStack.createStack()).size() = 0
2. (aStack.push(item)).size() = aStack().size() +1
3. (aStack.pop()).size() = aStack.size() - 1
4. (aStack.createStack()).isEmpty() = TRUE
5. (aStack.push(item)).isEmpty() = FALSE
6. (aStack.createStack()).pop() = ERROR
7. (aStack.push(item)).top() = item
8. (aStack.push(item)).pop() = item
9. (((aStack.push(item1)).push(item2)).pop()).top() = item1

ADT Queue	
Definition	The ADT Queue is a linear sequence of an arbitrary number of items, along with access procedures. Note that sequence does not imply order. The access procedures permit insertion of elements only at the back of the sequence. Deletion and retrieval of elements are only performed at the front of the sequence. It's because of this that is called a first-in-first-out (or FIFO) list. A queue is either empty, or it consists of a sequence of items.
Object Representation	 front = a_1 $\wedge$ back = a_n
Invariant	(q.isEmpty() == true $\wedge$ front == back == NIL) $\oplus$ (q = {$a_1 \rightarrow a_2 \rightarrow ... \rightarrow a_n$} $\wedge$ front == a_1 $\wedge$ back == a_n)

Procedures	**● Constructor operations**
	  ○ *createQueue()*
	    //Creates an empty queue.
	    //**post:** A queue with no elements inside has been created.
	● Analyzer operations
	  ○ *front()*
	    //Returns the element at the front of the queue but does not modify it.
	    //**post:** Returns the front element of the queue. If the queue is //empty then its front is NIL and this is the retrieval.
	  ○ *isEmpty()*
	    //Determines whether a queue is empty.
	    //**post:** Returns TRUE if the front of the queue is NIL and //FALSE otherwise.
	  ○ *size()*
	    //Returns the size of the queue.
	    //**post:** Returns the total number of elements in the queue.
	● Modifier operations

	○ *enqueue(newItem)*
	//Adds newItem to the queue.
	//**post:** newItem is at the back of the queue.
	○ *dequeue()*
	//Removes and returns the front element of the queue.
	//**pre:** The queue is not empty.
	//**post:** The front is taken off the queue and the new front is the
	//element that was behind the previously returned element. If
	//the queue is empty, it results in an error.

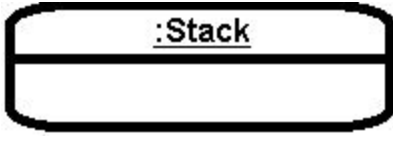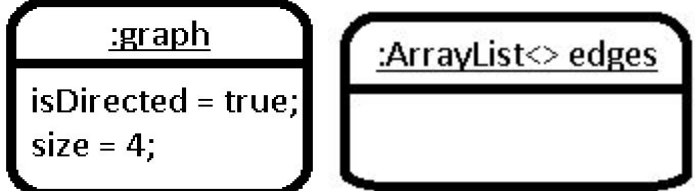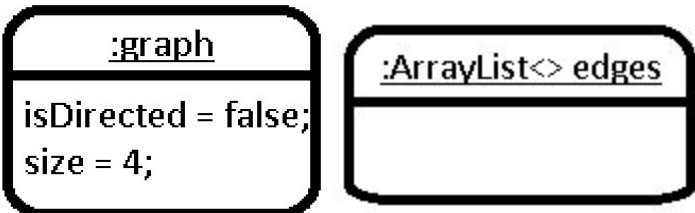| Axioms of the ADT Queue |
| --- |
| 1. (aQueue.createQueue()).size() = 0
2. (aQueue.enqueue(item)).size() = aQueue().size() +1
3. (aQueue.dequeue()).size() = aQueue.size() - 1
4. (aQueue.createQueue()).isEmpty() = TRUE
5. (aQueue.enqueue(item)).isEmpty() = FALSE
6. (aQueue.createQueue()).dequeue() = ERROR |
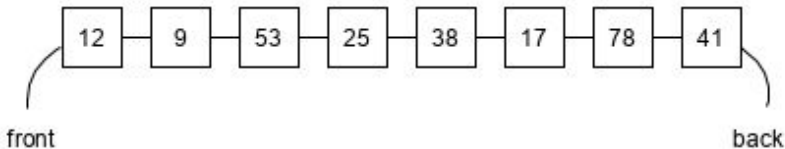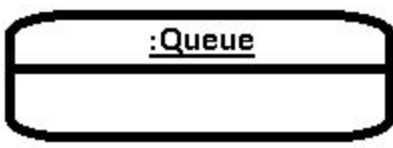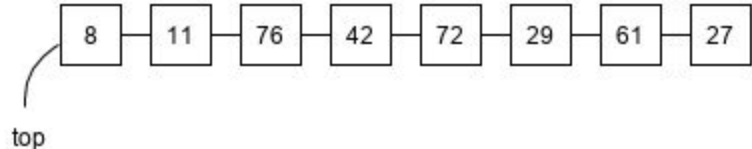
ADT Graph	
Definition	A Graph is a collection of finite sets of vertices or nodes (V) and edges (E). Edges are represented as ordered pairs (u, v) where (u, v) indicates that there is an edge from vertex u to vertex v. Edges may contain cost, weight or length. The degree or valency of a vertex is the number of edges that connect to it.
Object Representation	$G(V, E) \land V = \{v_1, v_2 \ldots v_{n-1}, v_n\} \land E \subseteq \{\{x, y\} / (x, y) \in V^2 \land x \neq y\}$
Invariant	$n = |V| \rightarrow (\forall x \in V / |x| <= n\text{-}1) \land |E| <= n(n - 1)$

Procedures	● **Constructor operations**
	  ○ *createGraph()*
	    //Creates an empty graph
	    //**post:** A graph with no vertices has been created
	● **Analyzer operations**
	  ○ *traverse()*
	    //Goes through all the nodes performing any kind of operation
	    //or retrieving any sort of information
	  ○ *isEmpty()*
	    //Determines whether a graph is empty.
	    //**post:** Returns TRUE if the graph is empty and FALSE
	    //otherwise.

	○ *searchVertex(key)*
	//Traverses the graph looking for an element with the specified //key
	//**post:** Returns the element with specified key if found and //NIL otherwise.
	○ *order()*
	//Returns the order of the graph, that is, the number of nodes //in it.
	//**post:** Returns the total number of vertices in the graph**.**
	● **Modifier operations**
	  ○ *delete(key)*
	//Deletes the verter that matches the key
	//**post:** Returns TRUE if the vertex was found and deleted or //FALSE otherwise
	○ *insert(key)*
	//Inserts a new isolated vertex into the graph with the specified //key.
	//**pre:** key does not match any existing node key.
	//**post:** The isolated node has been added to the graph.
	○ *link(u, v)*
	//Adds an edge from node *u* and *v*
	//**pre:** Both *u* and *v* are nodes currently in the graph
	//**post:** An edge has been created to link node *u* to node *v*

Axioms of the ADT Graph

1. (aGraph.createGrapsh()).order() = 0
2. (aGraph.insert(k)).order() = aGraph().order() +1
3. (aGraph.createGraph()).delete(k) = FALSE
4. (aGraph.createGraph().insert(k)).delete(k) = TRUE
5. (aGraph.insert(k)).isEmpty() = FALSE
6. (aGraph.createGraph()).isEmpty() = TRUE
7. (aGraph.insert(k)).getVertex(k) = V(k)
8. (aGraph.createGraph()).getVertex(k) = NIL
9. (aGraph.link(u, v)) → E(u, v) != NIL

Name	Class under test	Stage
setupStage1	Queue	(A)
setupStage2	Queue	:Queue
setupStage1	Stack	(B)
setupStage2	Stack	:Stack
setupStageDirected	AdjacencyMatrixGraph	:graph — isDirected = true; size = 4; :ArrayList<> edges
setupStageUndirected	AdjacencyMatrixGraph	:graph — isDirected = false; size = 4; :ArrayList<> edges

setupStageGraphWithIsolatedVertices	AdjacencyMatrixGraph	
setupStageDirected	AdjacencyListGraph	
setupStageUndirected	AdjacencyListGraph	
setupStageGraphWithIsolatedVertices	AdjacencyListGraph	

TEST CASES

Test Objective: Verificate that whenever you create a queue this must be empty and its front null.				
Class	Method	Stage	Input	Output
Queue	Queue<>()	N/A	N/A	The queue's front is null when you try to get the front because the queue is empty.

Test Objective: Verificate that the method enqueue adds elements inside the queue and the queue size is increasing by one per each element is added.				
Class	Method	Stage	Input	Output
Queue	enqueue	setupStage 2	50 random elements	The queue size has increased by one per each calling as we expected.

Test Objective: Verificate that the method dequeue eliminates elements inside the queue and the queue size is decreasing by one per each element is eliminated.				
Class	Method	Stage	Input	Output
Queue	dequeue	setupStage 1	N/A	The queue size has decreased by one per each calling as we expected.

Test Objective: Verificate that whenever you create a stack this must be empty and its top null.				
Class	Method	Stage	Input	Output
Stack	Stack<>()	N/A	N/A	The stack's top is null when you try to get the top because the stack is empty.

Test Objective: Verificate that the method top shows and does not delete the right element previously added inside the stack. Moreover, the method must throw null when the stack is empty.				
Class	Method	Stage	Input	Output
Stack	top	setupStage1	80 random elements	The element showed corresponds with the previously added. The stack's top is null when the stack is empty.

Test Objective: Verify that the method pop shows and deletes the right element previously added inside the stack. Moreover, the method must decrease the stack size by one whenever top is called.

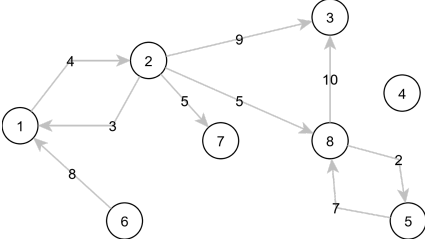| Class	Method	Stage	Input	Output
Stack	pop	setupStage 1	N/A	The stack size is decreasing by one each time pop is called.

Test Objective: Verify that the method push adds an element inside the stack and besides this increases the stack size by one.

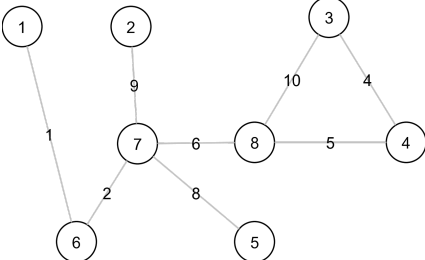| Class	Method	Stage	Input	Output
Stack	push	setupStage 2	50 random elements	The stack size has increased by one every when push was called.

Test Objective: Verify that whenever we create an AdjacencyListGraph/AdjacencyMatrixGraph regardless the constructor input values this must be empty. Whenever a vertex is added to the graph it should be found when searchVertex is called. Additionally, the order of the graph must increase in one every time a vertex is successfully added to it.

Class	Method	Stage	Input	Output
AdjacencyListGraph

AdjacencyMatrixGraph | insertAndSearchVertexTest | setupStage Directed | A graph with 50 random inserted vertices. | Every stage call yield successive empty-filled states in the graph just it was expected. The order of the graph behaved as expected with respect to the insertion. Each added vertex was found by the searchVertex method. |
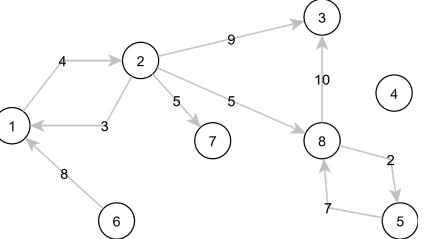
Test Objective: Verificate that the method link links a source vertex with another one and assigns correctly the weight of their relationship when the graph is directed.

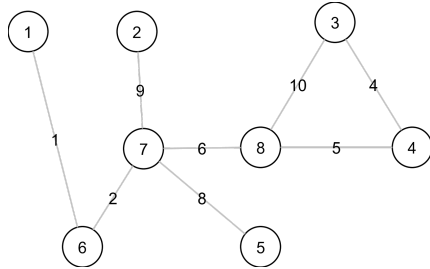| Class	Method	Stage	Input	Output
Adjacency ListGraph Adjacency MatrixGraph	linkVertices InDirected GraphTest	setupStage Directed		The directed graph contains all of the edges that were inserted as we expected.

Test Objective: Verificate that the method link links a source vertex with another one and assigns correctly the weight of their relationship when the graph is undirected.

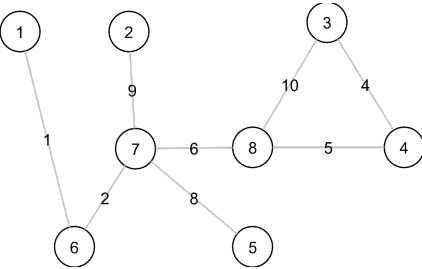| Class	Method	Stage	Input	Output
Adjacency ListGraph Adjacency MatrixGraph	linkVertice sInUndirect edGraphTe st	setupStag eUndirecte d		The undirected graph contains all of the edges that were inserted as we expected.

Test Objective: Verificate that the method unlink unlinks a source vertex with another one which already exist inside the graph when the graph is directed.

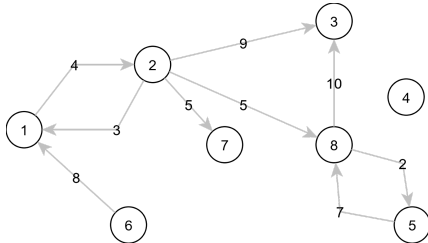| Class	Method	Stage	Input	Output
Adjacency ListGraph Adjacency MatrixGraph	unlinkVe rticesInD irectedG raphTest	setupStage Directed		The selected edges were removed successfully.

Test Objective: Verificate that the method unlink unlinks a source vertex with another one which already exist inside the graph when the graph is undirected.

Class	Method	Stage	Input	Output
Adjacency ListGraph Adjacency MatrixGraph	unlinkVe rticesTes tInUndire ctedGra ph	setupStage Undirected		The selected edges were removed successfully.

Test Objective: Verificate whenever BFS is applied in an undirected graph as from a source vertex visits all the vertices and besides delivers the correct least stop paths.

Class	Method	Stage	Input	Output
Adjacency ListGraph Adjacency MatrixGraph	BFSInUn directed Graph	setupStage Undirected		BFS discovered and finished all the vertices as from the specified source vertex and gave us the expected least stop paths when the graph was undirected.

Test Objective: Verificate whenever BFS is applied in a directed graph as from a source vertex visits all the vertices and besides delivers the correct least stop paths.

Class	Method	Stage	Input	Output
Adjacency ListGraph Adjacency MatrixGraph	BFSInDir ectedGr aph	setupStage Directed		BFS discovered and finished all the vertices as from the specified source vertex and gave us the expected least stop paths when the graph was directed.

Test Objective: Verificate whenever DFS is applied in an undirected this visits all the vertices regardless of the starting point.
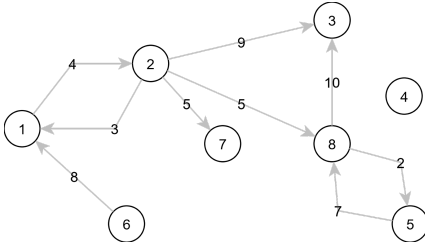
Class	Method	Stage	Input	Output
Adjacency ListGraph Adjacency MatrixGraph	DFS	N/A		All the vertices have their state in black, therefore, they were visited.

Test Objective: Verificate whenever DFS is applied in a directed this visits all the vertices specifying a source vertex.

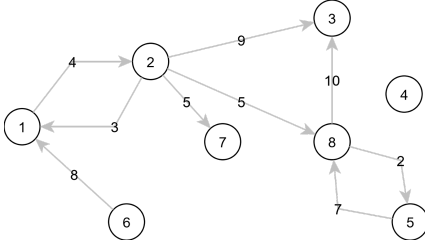| Class	Method	Stage	Input	Output
Adjacency ListGraph Adjacency MatrixGraph	DFSWith GivenSource	N/A		All the vertices were as from the specified source were visited.

Test Objective: Verificate that the method deleteVertex deletes a specified vertex that exist inside the graph with its edge when is necessary.

Class	Method	Stage	Input	Output
Adjacency ListGraph Adjacency MatrixGraph	deleteVertex	N/A		All the selected vertices were deleted successfully.

Test Objective: Verificate whenever Dijkstra is applied in a source vertex this gives the correct shortest path of others selected vertices that already exist inside the graph.

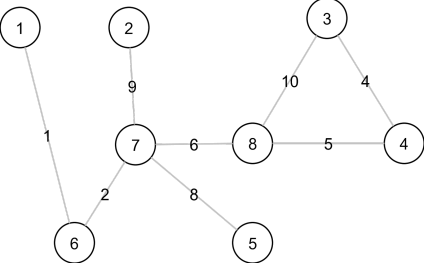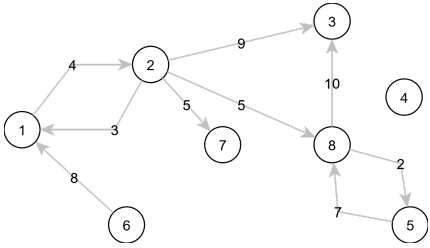| Class	Method	Stage	Input	Output
Adjacency ListGraph Adjacency MatrixGraph	Dijkstra	setupStage Directed		Dijkstra gave all the correct shortest paths as from a specified source vertex.

Test Objective: Verificate whenever FloydWarshall is applied in the actual graph this gives the correct shortest path between two specified vertices.				
Class	Method	Stage	Input	Output
Adjacency ListGraph Adjacency MatrixGraph	FloydWa rshall	setupStage Directed		FloydWarshall has given the correct shortest path for specified pair of vertices.

Test Objective: Verificate whenever Prim is applied in the actual graph this gives the correct minimum spanning tree with any source vertex when the graph is connected and weighted.				
Class	Method	Stage	Input	Output
Adjacency ListGraph Adjacency MatrixGraph	Prim	setupStage Directed		Prim has given the correct minimum spanning tree for specified source vertex.

<table>
<tr>
<td colspan="6">Test Objective: Verificate whenever Kruskal is applied in the actual graph this gives the correct minimum spanning tree no matter if it is connected or not. Besides the graph has to be weighted as well.</td>
</tr>
<tr>
<th>Class</th>
<th>Method</th>
<th>Stage</th>
<th>Input</th>
<th colspan="2">Output</th>
</tr>
<tr>
<td>Adjacency ListGraph

Adjacency MatrixGraph</td>
<td>Kruskal</td>
<td>setupStage Directed</td>
<td></td>
<td colspan="2">Kruskal has given the correct minimum spanning tree for the specified graph.</td>
</tr>
</table>

Step 7. Implementation of the design.

Class diagram
https://github.com/7yrionLannister/pacman-game/blob/master/classdiagram.png

Sources.

https://en.wikipedia.org/wiki/Pac-Man

https://www.gamasutra.com/view/feature/132330/the_pacman_dossier.php?page=1

https://www.hackerearth.com/practice/algorithms/graphs/graph-representation/tutorial/

https://www.geeksforgeeks.org/graph-and-its-representations/

https://runestone.academy/runestone/books/published/pythonds/Graphs/toctree.html

https://www.amazon.com/Introduction-Algorithms-3rd-MIT-Press/dp/0262033844