Proyecto

Algoritmos y estructuras de datos

Autores:

Sergio Andrés Lozada Sánchez.

Iván Camilo Goez Palacio.

Cristhian Eduardo Castillo Meneses.

Tabla de contenido

Contexto Problemático:

Enunciado:

Triadam Cola es una popular compañía que produce y vende bebidas de Cola en diferentes países, entre ellos Colombia. El departamento de distribución ha tenido problemas debido a los constantes retrasos en la entrega de los productos.

La empresa realizó un estudio sobre cómo estaban haciendo la distribución del producto, dándose cuenta de que los recorridos que realizaban tenían una distancia muy larga, puesto que entregaban los productos según el orden en el que se realizaba el pedido, es decir, el pedido #1 era entregado de primero, el pedido #2 de segundo y así con todos los pedidos.

Le piden a la empresa TriadamSoft (dónde usted trabaja) que forma parte del conglomerado Triadam que le ayude con su situación por medio de una solución tecnológica. Usted como es conocedor de los grafos, sabe que puede representar los puntos de entrega como nodos dentro de un grafo, y así hallar el camino más rápido para hacer las entregas, propone esta solución a su superior y es aceptada.

Triadam Cola desea probar la solución en la ciudad de Cali antes de llevarla a otros lados, los camiones en esta ciudad pueden hacer un máximo de 15 paradas, debido a la cantidad de productos que pueden llevar.

El equipo de TriadamSoft se pone manos a la obra para presentar una versión funcional de la solución que se pueda poner en práctica en la ciudad de Cali, esta versión debe permitir mostrar el recorrido que debe hacer cada uno de los camiones para entregar todos los pedidos.

1. Identificación del problema

1.1. Necesidades y síntomas:

- TriadamCola requiere hacer los pedidos con una mayor velocidad, debido a que ha tenido múltiples problemas con sus clientes, lo cual no es bueno para su marca.
- La capacidad de los camiones es limitada, y para mejorar el tiempo de entrega las rutas deben distribuirse de una manera óptima.
- La solución del problema debe dar la ruta más corta que debe realizar cada uno de los camiones para realizar la entrega de todos los pedidos.
- La solución debe contar con un sistema que permita administrar los pedidos y camiones que tiene la empresa.

1.2. Definición del problema

Triadam Cola requiere el desarrollo de un programa que le permita ser más eficiente a la hora de realizar la entrega de sus pedidos, debido a los constantes atrasos que han tenido a la hora de hacer esta tarea, lo que ha llevado a que tengan problemas con sus clientes.

1.3.Requerimientos funcionales (RF) y no funcionales (RNF).

NOMBRE	RF1. Agregar un pedido
RESUMEN	Permite agregar un pedido que debe realizarse
ENTRADAS	• Cantidad del producto • Ubicación
RESULTADOS	El pedido se agrega satisfactoriamente

NOMBRE	RF2. Agregar un camión
RESUMEN	Permite agregar un nuevo camión
ENTRADAS	• Matricula del camión
RESULTADOS	El camión se agrega satisfactoriamente

NOMBRE	RF3. Eliminar un pedido
RESUMEN	Permite eliminar un pedido realizado.
ENTRADAS	• Id del pedido
RESULTADOS	El pedido se elimina de la lista

NOMBRE	RF4. Eliminar un camión
RESUMEN	Se elimina el camión indicado de la lista de camiones disponibles
ENTRADAS	• Matricula del camión
RESULTADOS	El camión se elimina de los camiones disponibles

NOMBRE	RF4. Encontrar ruta
RESUMEN	Encuentra la ruta que debe hacer cada camión
ENTRADAS	
RESULTADOS	Se indica la ruta de cada camión debe hacer para realizar las entregas de los pedidos

NOMBRE	RNF1. Ruta Mínima
RESUMEN	La ruta realizada por cada camión debe ser la mínima posible.
ENTRADAS	
RESULTADOS	

2. Recopilación de la información:

https://es.wikipedia.org/wiki/Bebida_de_cola

https://es.wikipedia.org/wiki/Problema_del_camino_m%C3%A1s_corto

https://brilliant.org/wiki/dijkstras-short-path-finder

https://es.wikipedia.org/wiki/Grafo

https://es.wikipedia.org/wiki/Teor%C3%ADa_de_grafos#Grafo_simple

https://es.wikipedia.org/wiki/Teor%C3%ADa_de_grafos#Grafos_conexos

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

https://es.wikipedia.org/wiki/Lista_de_adyacencia

https://www.ecured.cu/Matriz_de_adyacencia

https://es.wikipedia.org/wiki/Multigrafo

https://www.ecured.cu/Algoritmo_de_Dijkstra

https://www.ecured.cu/Algoritmo_de_Kruskal

https://sites.google.com/site/complejidadalgoritmicaes/prim

https://es.wikipedia.org/wiki/B%C3%BAsqueda_en_anchura

https://es.wikipedia.org/wiki/B%C3%BAsqueda_en_profundidad

https://www.ecured.cu/Floyd-Warshall

Bebidas de cola: Una bebida de cola es un refresco usualmente saborizado con caramelo colorado, y que frecuentemente posee cafeína.

Camino más corto: es el problema que consiste en encontrar un camino entre dos vértices (o nodos) de tal manera que la suma de los pesos de las aristas que lo constituyen es mínima. Un ejemplo de esto es encontrar el camino más rápido para ir de una ciudad a otra en un mapa. En este caso, los vértices representarían las ciudades y las aristas las carreteras que las unen, cuya ponderación viene dada por el tiempo que se emplea en atravesarlas.

Grafo: Es un conjunto de objetos llamados vértices o nodos unidos por enlaces llamados aristas o arcos, que permiten representar relaciones binarias entre elementos de un conjunto.

Grafo simple: Un grafo es simple si a lo sumo existe una arista uniendo dos vértices cualesquiera. Esto es equivalente a decir que una arista cualquiera es la única que une dos vértices específicos.

Grafos conexos: Un grafo es conexo si cada par de vértices está conectado por un camino; es decir, si para cualquier par de vértices (a, b), existe al menos un camino posible desde a hacia b.

Grafo no dirigido: Un grafo no dirigido es un grafo en el que todas las aristas son bidireccionales, es decir, las aristas no apuntan en una dirección específica.

Grafo dirigido: Un grafo dirigido es un grafo en el que todas las aristas son unidireccionales, es decir, las aristas apuntan en una sola dirección.

Grafo ponderado: En un grafo ponderado, a cada arista se le asigna un peso o un costo.

Lista de adyacencia: Una lista de adyacencia es una representación de todas las aristas o arcos de un grafo mediante una lista. Si el grafo es no dirigido, cada entrada es un conjunto o multiconjunto

de dos vértices conteniendo los dos extremos de la arista correspondiente. Si el grafo es dirigido, cada entrada es una tupla (lista ordenada de elementos) de dos nodos, uno denotando el nodo fuente y el otro denotando el nodo destino del arco correspondiente.

Matriz de adyacencia: Es una matriz cuadrada de orden NxN asociada a un grafo de orden N, donde sus filas y columnas se identifican con los vértices del grafo y en las celdas se indican la cantidad de aristas a los nodos asignado a la fila y columnas en cuestión.

Multigrafo: Es un grafo que está facultado para tener aristas múltiples; es decir, aristas que relacionan los mismos nodos. De esta forma, dos nodos pueden estar conectados por más de una arista.

Los multígrafos podrían usarse, por ejemplo, para modelar las posibles conexiones de vuelo ofrecidas por una aerolínea. Para este caso tendríamos un grafo dirigido, donde cada nodo es una localidad y donde pares de aristas paralelas conectan estas localidades, según un vuelo es hacia o desde una localidad a la otra.

Algoritmo de dijkstra: También llamado algoritmo de caminos mínimos es un algoritmo para la determinación del camino más corto dado un vértice origen al resto de vértices en un grafo con pesos en cada arista. Su nombre se refiere a Edsger Dijkstra, quien lo describió por primera vez en 1959.

En múltiples aplicaciones donde se aplican los grafos, es necesario conocer el camino de menor costo entre dos vértices dados:

- Distribución de productos a una red de establecimientos comerciales.
- Distribución de correos postales.
- Sea G = (V, A) un grafo dirigido ponderado.

El problema del camino más corto de un vértice a otro consiste en determinar el camino de menor costo, desde un vértice u a otro vértice v. El costo de un camino es la suma de los costos (pesos) de los arcos que lo conforman.

Algoritmo de kruskal: Es un algoritmo de la teoría de grafos para encontrar un árbol recubridor mínimo en un grafo conexo y ponderado. Es decir, busca un subconjunto de aristas que, formando un árbol, incluyen todos los vértices y donde el valor total de todas las aristas del árbol es el mínimo. Si el grafo no es conexo, entonces busca un bosque expandido mínimo (un árbol expandido mínimo para cada componente conexa).

- Funcionamiento del algoritmo de Kruskal:

1. Se selecciona, de entre todas las aristas restantes, la de menor peso siempre que no cree ningún ciclo.
2. Se repite el paso 1 hasta que se hayan seleccionado |V| - 1 aristas. Siendo V el número de vértices.

Algoritmo de prim: El algoritmo de Prim, dado un grafo conexo, no dirigido y ponderado, encuentra un árbol de expansión mínima. Es decir, es capaz de encontrar un subconjunto de las aristas que formen un árbol que incluya todos los vértices del grafo inicial, donde el peso total de las aristas del árbol es el mínimo posible.

- Funcionamiento del algoritmo de Prim:

1. Se marca un vértice cualquiera. Será el vértice de partida.
2. Se selecciona la arista de menor peso incidente en el vértice seleccionado anteriormente y se selecciona el otro vértice en el que incide dicha arista.
3. Repetir el paso 2 siempre que la arista elegida enlace un vértice seleccionado y otro que no lo esté. Es decir, siempre que la arista elegida no cree ningún ciclo.
4. El árbol de expansión mínima será encontrado cuando hayan sido seleccionados todos los vértices del grafo.

Depth first search (dfs): Es un algoritmo de búsqueda no informada utilizado para recorrer todos los nodos de un grafo o árbol (teoría de grafos) de manera ordenada, pero no uniforme. Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando, de forma recurrente, en un camino concreto. Cuando ya no quedan más nodos que visitar en dicho camino, regresa (Backtracking), de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado.

Breadth first search (bfs): Es un algoritmo de búsqueda no informada utilizado para recorrer o buscar elementos en un grafo (usado frecuentemente sobre árboles). Intuitivamente, se comienza en la raíz (eligiendo algún nodo como elemento raíz en el caso de un grafo) y se exploran todos los vecinos de este nodo. A continuación, para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el árbol.

Floyd-warshall: Es un algoritmo de análisis sobre grafos que permite encontrar el camino mínimo en grafos dirigidos ponderados. El algoritmo encuentra el camino entre todos los pares de vértices en una única ejecución, constituyendo un ejemplo de programación dinámica.

- Características:

1. Obtiene la mejor ruta entre todo par de nodos.
2. Trabaja con la matriz D inicializada con las distancias directas entre todo par de nodos.
3. La iteración se produce sobre nodos intermedios, o sea para todo elemento de la matriz se prueba si lo mejor para ir de i a j es a través de un nodo intermedio elegido o como estaba anteriormente, y esto se prueba con todos los nodos de la red. Una vez probados todos los nodos de la red como nodos intermedios, la matriz resultante da la mejor distancia entre todo par de nodos.
4. El algoritmo da sólo la menor distancia; se debe manejar información adicional para encontrar tablas de encaminamiento.
5. Hasta no hallar la última matriz no se encuentran las distancias mínimas.
6. Su complejidad es del orden de n^3.

3. **Búsqueda de soluciones creativas:**

Para la búsqueda de soluciones creativas se realiza una lluvia de ideas donde cada uno de los integrantes planteamos ideas sobre cómo podría ser el funcionamiento del software, la interfaz, los algoritmos y que estructuras de datos se podrían usar durante el desarrollo.

Alternativa 1: Cola de prioridad

A través de las rutas generadas por el grafo, se agregan a una cola de prioridad donde su factor de orden es el peso menor de la primera arista, es decir, el lugar más cercano a la fábrica.

Alternativa 2: Lista enlazada

Al generarse las rutas para los pedidos de la compañía, estas se agregan a una lista enlazada donde el peso menor de la primera arista tendrá relación con la segunda con peso menor, la segunda con la tercera y así sucesivamente.

Alternativa 3: Montículo o heap

Las rutas generadas se guardarán en un montículo o heap, con la idea de que se obtengan las rutas más cercanas a la fábrica en O(1).

Alternativa 4: Dijsktra

La idea consiste en que se va a generar la ruta más corta o con menor peso desde la fábrica hasta el último pedido. Es decir, se va a encontrar la mejor ruta entre todas las posibles.

Alternativa 5: Prim

Se encuentra un árbol de expansión mínima. Es decir, se encuentra un subconjunto de las aristas que formen un árbol que incluya todas las rutas de la fábrica a los pedidos, donde el peso total de las aristas del árbol es el mínimo posible.

Alternativa 6: Kruskal

Se busca un subconjunto de aristas que, formando un árbol, incluyen todos los pedidos, incluyendo la fábrica, donde el valor total de todas las aristas del árbol es el mínimo posible.

Alternativa 7: Floyd Warshall

La idea se basa en que se conocerá todas las rutas desde la fábrica hasta los pedidos, y de los pedidos a otros pedidos y así saber el camino mínimo entre todos los vértices del grafo.

4. Transición de las Ideas a los Diseños Preliminares:

Alternativa 4: Dijsktra

- Este algoritmo nos sirve para determinar cuál es el camino más corto dado un vértice origen al resto de vértices en un grafo con pesos en cada arista, la idea principal de este algoritmo es de ir explorando todos los caminos más cortos que parten del vértice origen y que llevan a cada uno de los vértices.

Alternativa 5: Prim

- El algoritmo empieza por un nodo que se le ha asignado, esto sirve perfectamente para representar el camino más corto para el problema de la compañía Triadam porque el nodo inicial será la fábrica y está, con el algoritmo Prim, estará conectada con todos los pedidos.

4.1.Ideas no factibles

Alternativa 6: Kruskal

- El algoritmo empieza por la arista de menor peso, pero esto representa un problema porque lo que se quiere es que se empiece por el vértice fábrica porque de ahí es de donde salen los camiones a realizar la entrega de pedidos.

Alternativa 7: Floyd Warshall

- Si hay muchos pedidos, es decir, más de 1000 pedidos el algoritmo no sería muy eficiente.
- Al ser un algoritmo de complejidad n^3 con datos de más de 1000 seria ineficiente.
- Permite conocer el camino más corto entre todos los nodos, pero para solucionar el problema solo se quiere conocer la ruta más corta entre la fábrica y sus pedidos.

Alternativa 1: Cola de prioridad

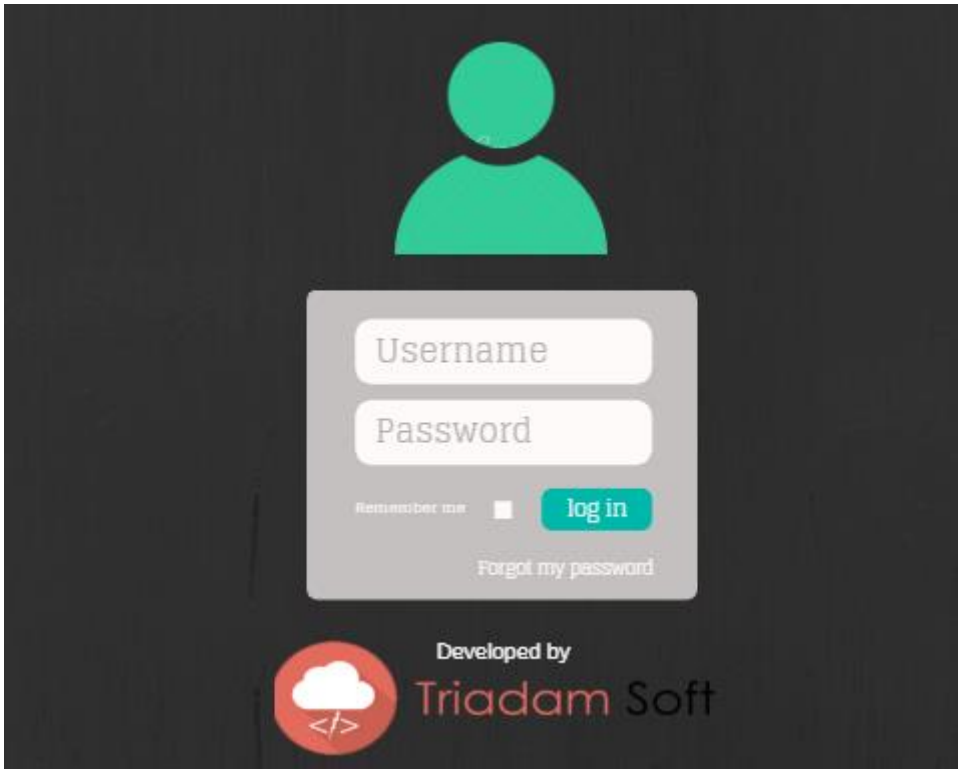
- Es posible que los caminos estén organizados de una forma que los caminos no estén de la óptima.

Alternativa 2: Lista Enlazada:

- Aunque las aristas estén organizadas por su peso de menor a mayor, esto no garantiza que el camino sea el más corto para realizar la entrega de los productos.

4.2. Diseños preliminares

Login:



Interfaz:

5. Evaluación y selección de la mejor solución:

Criterios

A. La solución permite acceder a un nodo en una complejidad temporal:

[2] <O (1)

[1] O (V)

B. El modelo de base de datos en la solución es:

[2] Intuitivo y fácil de entender.

[1] Complejo y difícil de codificar para programadores.

C. El nivel de aprendizaje gracias a la implementación de la solución es:

[3] Alto

[2] Mediol

[1] Bajo

D. La solución utiliza la complejidad espacial:

[2] <O (n ^ 2)

[1] O (n ^ 2)

Alternativas	Criterio A	Criterio B	Criterio C	Criterio D	Total
Alternativa 4	1	2	3	1	7
Alternativa 5	1	2	3	1	7

6. Preparación de informes y especificaciones:

6.1. Especificación del problema con base en la entrada y la salida

Problema: Ruta más corta para la entrega de pedidos

Entrada: Sea P= $\{P_1, P_2, P_3, \dots, P_n\}$ donde P_x es una localización donde se entrega un pedido.

Salida: $R \ni P_1, P_2, P_3, \ldots, P_{15}$, donde R representa el camino más corto que puede recorrer un camión para realizar las entregas de los pedidos.

6.2.TAD

TAD Definición

GraphMatrix
Representation: **Matrix n^2 where the rows are the start vertices and the columns represent the arrival points**
Invariants: **n is the number of vertices.** **Every Object in the matrix belong to the same class**
Operations: **CreateGraph boolean ----> Graph** **addEdge E, V, V ----> boolean** **addVertex V ----> void** **getVertices True ----> List<V>** **getEdges True ----> List<E,V,V>** **getLabel V, V ----> E** **getNeighbors V ----> List<V>** **isThereEdge V, V -----> boolean** **isUndirected True -----> boolean** **getNumberOfVertices True -----> int** **getValue int -----> V** **getInteger V -----> int** **getEdgesArray True -----> List<E>[][]**

CreateGraph(boolean d)
Pre: true
Post: A new graph has been created, if d is true, is undirected, directed otherwise

addEdge(E e, V v1, V v2)
Pre: e, v1 y v2 are != null
Post: The matrix is filled in the positions in which the vertices are joined, if is directed, v1 point v2

addVertex(V v)
Pre: v != null
Post: New vertex added in the graph increasing the size of the matrix by rows and columns

getVertices()
Pre: True
Post: list of vertices returned

getEdges()
Pre: True
Post: list of edges returned

getLabel(V v1,V v2)
Pre: v1 and v2 are in the graph
Post: edge connecting the two vertices is returned, null otherwise

getNeighbors(V v)
Pre: v is in the graph
Post: List of neighbors of v is returned

isThereEdge(V v1,V v2)
Pre: True
Post: There are vertices v1 and v2 and there exist an edge connecting them

IsUndirected()
Pre: True
Post: returns true if the graph is undirected, false otherwise

getNumberOfVertices()
Pre: True
Post: return number of vertices of the graph

getValue(int i)
Pre: True
Post: return the value associated to the given integer, null if there is no such integer

getInteger(V v)
Pre: v != null
Post: return the integer associated to the given value, null if there is no such integer

getEdgesArray()
Pre: True
Post: return the matrix containing the edges

TAD GraphList

Representation:
List of V elements in the graph, which point to its neighbors by edges E.

Invariants:
Every Object in the matrix belong to the same class.
V is the number of vertices.

Operations:

CreateGraph	**boolean**	**----> Graph**
addEdge	**E, V, V**	**----> boolean**
addVertex	**V**	**----> void**
getVertices	**True**	**----> List<V>**
getEdges	**True**	**----> List<E,V,V>**
getLabel	**V, V**	**----> E**
getNeighbors	**V**	**----> List<V>**
isThereEdge	**V, V**	**-----> boolean**
isDirected	**True**	**-----> boolean**
getNumberOfVertices	**True**	**-----> int**
getVertex	**V**	**-----> Vertex**

CreateGraph(boolean d)

Pre: true

Post: A new graph has been created, if d is true, is undirected, directed otherwise

addEdge(E e, V v1, V v2)

Pre: e, v1 y v2 son != null

Post: The matrix is filled in the positions in which the vertices are joined, if is directed, v1 point v2

addVertex(V v)

Pre: v != null
Post: New vertex added in the graph increasing the size of the matrix by rows and columns

getVertices()
Pre: True
Post: list of vertices returned

getEdges()
Pre: True
Post: list of edges returned

getLabel(V v1,V v2)
Pre: v1 and v2 are in the graph
Post: edge connecting the two vertices is returned, null otherwise

getNeighbors(V v)
Pre: v is in the graph
Post: List of neighbors of v is returned

isThereEdge(V v1,V v2)
Pre: True
Post: There are vertices v1 and v2 and there exist an edge connecting them

IsUndirected()
Pre: True

Post: returns true if the graph is undirected, false otherwise

getNumberOfVertices()
Pre: True
Post: return number of vertices of the graph

getVertex(V v)
Pre: True
Post: return vertex associated to the given value, null if there is no such vertex

6.3.Diseño de pruebas unitarias

Prueba No. 1	**Objetivo de la prueba:** Comprobar que el método addEdge une correctamente dos nodos por medio de una arista. **Firma del método:** public boolean addEdge(Edge e, Vertex v1, Vertex v2)			
Clase para probar	**Método**	**Escenario**	**Valores de entrada**	**Resultado**
GraphMatrix	AddEdge(Edge e, Vertex v1, Vertex v2)	GraphMatrix g = new GraphMatrix(boolean d) g.addVertex(Vertex v1) g.addVertex(Vertex v2)	AddEdge(Edge e, Vertex v1, Vertex v2) e: es la arista con que se unirán a los dos nodos que pertenecen al grafo. v1: es un nodo del grafo. v2: es un nodo del grafo	True, la arista e conectó a los nodos v1 y v2.

Prueba No. 2	**Objetivo de la prueba:** Comprobar que el método addVertex agrega correctamente un nodo al grafo. **Firma del método:** public void addVertex(Vertex v1)			
Clase para probar	**Método**	**Escenario**	**Valores de entrada**	**Resultado**
GraphMatrix	AddVertex(Vertex v)	GraphMatrix g = new GraphMatrix(boolean d) Vertex v1 = new Vertex(Integer value, Integer k)	g.addVertex(Vertex v1) g.addVertex(Vertex v2) V1: es el nodo que se va a agregar al grafo. V2: es el nodo que se va a agregar al grafo.	Se espera que el método halla agregado satisfactoriamente los dos nodos al grafo.

Prueba No. 3	**Objetivo de la prueba:** Comprobar que el método getLabel retorna la arista que une a dos nodos dentro del grafo. **Firma del método:** public Edge getLabel(Vertex v1, Vertex v2)			
Clase para probar	**Método**	**Escenario**	**Valores de entrada**	**Resultado**
GraphMatrix	getLabel(Vertex v1, Vertex v2)	GraphMatrix g = new GraphMatrix(boolean d) g.addVertex(Vertex v1) g.addVertex(Vertex v2) g. AddEdge(Edge e, Vertex v1, Vertex v2)	g.getLabel(Vertex v1, Vertex v2) V1 y V2: son los nodos de los cuales se quiere saber si están o no unidos. .	Se retorna la arista e que une a los nodos v1 y v2. Retorna null si no están unidos.

Prueba No. 4	**Objetivo de la prueba:** Comprobar que el método getNeighbors retorna una lista de todos los nodos vecinos a un nodo x en particular. **Firma del método:** public List getNeighbors (Vertex v1)			
Clase para probar	**Método**	**Escenario**	**Valores de entrada**	**Resultado**
GraphMatrix	getNeighbors (Vertex v1)	GraphMatrix g = new GraphMatrix(boolean d) g.addVertex(Vertex v1) g.addVertex(Vertex v2) g. AddEdge(Edge e, Vertex v1, Vertex v2)	g. getNeighbors(Vertex v1) V1: es el nodo al que se quiere saber cuáles son sus vecinos. .	Se retorna la lista de los vecinos del nodo v1. Su único vecino es el nodo v2.

Prueba No. 5	**Objetivo de la prueba:** Comprobar que el método getNumberOfVertices retorna el número de nodos que existen en el grafo. **Firma del método:** public int getNumberOfVertices()			
Clase para probar	**Método**	**Escenario**	**Valores de entrada**	**Resultado**
GraphMatrix	getNumberOfVertices()	GraphMatrix g = new GraphMatrix(boolean d) g.addVertex(Vertex v1) g.addVertex(Vertex v2)	g. getNeighbors(). El método no tiene entrada alguna.	Se retorna el número de nodos que hay en el grafo, en este caso se retorna 2.

Prueba No. 6	**Objetivo de la prueba:** Comprobar que el método getEdgesArray retorna el número de aristas que existen en el grafo. **Firma del método:** public int getEdgesArray()			
Clase para probar	**Método**	**Escenario**	**Valores de entrada**	**Resultado**
GraphMatrix	getEdgesArray( )	GraphMatrix g = new GraphMatrix(b oolean d) g.addVertex(Ve rtex v1) g.addVertex(Ve rtex v2) g.addEgde(Edg e e, Vertex v1, Vertex v2)	g.getEdgesArra y(). El método no tiene entrada alguna.	Se retorna el número de aristas que hay en el grafo, en este caso se retorna 2.

Prueba No. 7	**Objetivo de la prueba:** Comprobar que el método addEdge une correctamente dos nodos por medio de una arista. **Firma del método:** public boolean addEdge(Edge e, Vertex v1, Vertex v2)			
Clase para probar	**Método**	**Escenario**	**Valores de entrada**	**Resultado**
GraphList	AddEdge(Edge e, Vertex v1, Vertex v2)	GraphList g = new GraphList(bool ean d) g.addVertex(Ve rtex v1) g.addVertex(Ve rtex v2)	AddEdge(Edge e, Vertex v1, Vertex v2) e: es la arista con que se unirán a los dos nodos que pertenecen al grafo. v1: es un nodo del grafo. v2: es un nodo del grafo	True, la arista e conectó a los nodos v1 y v2.

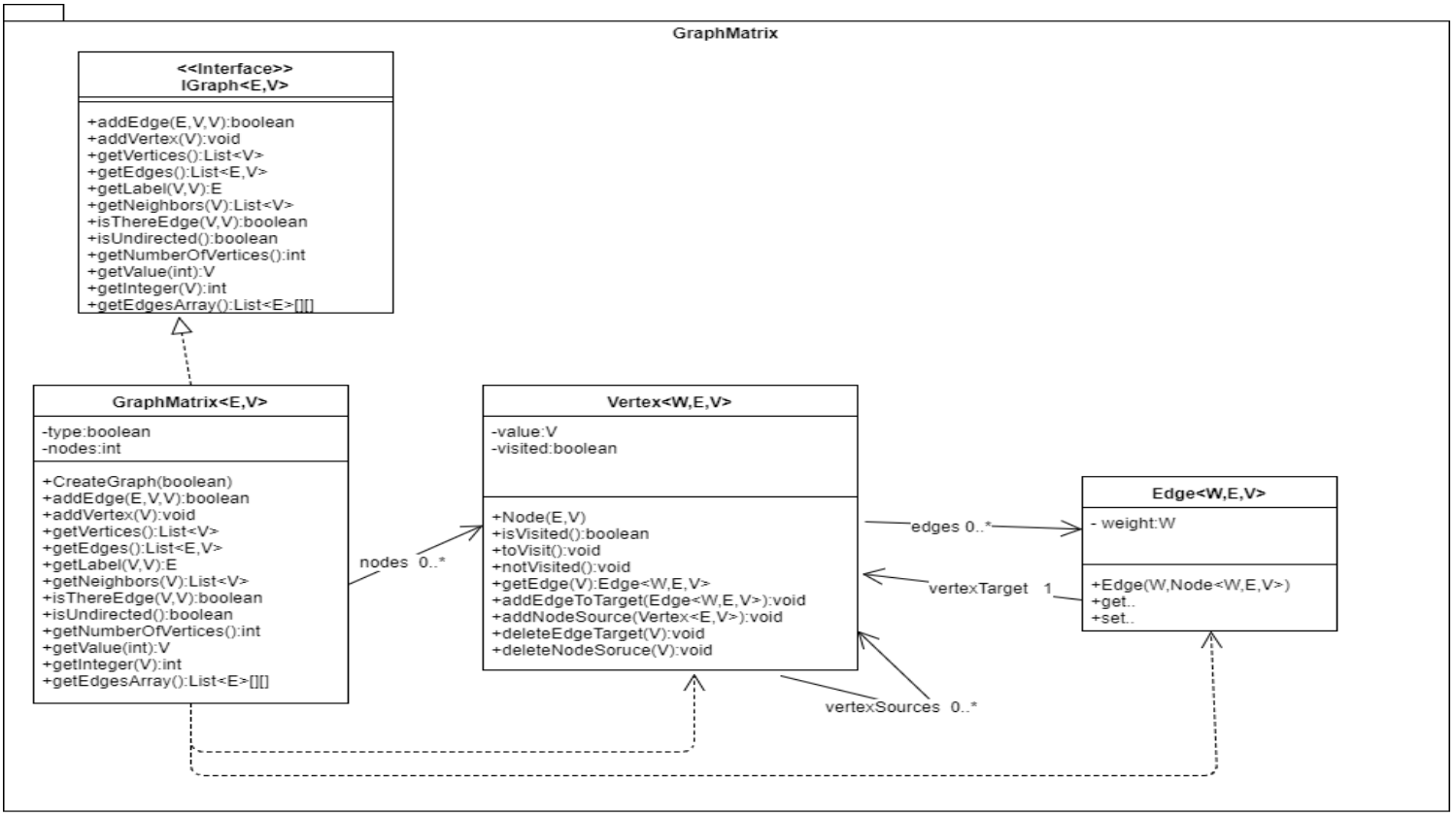
Prueba No. 8	**Objetivo de la prueba:** Comprobar que el método addVertex agrega correctamente un nodo al grafo. **Firma del método:** public void addVertex(Vertex v1)			
Clase para probar	**Método**	**Escenario**	**Valores de entrada**	**Resultado**
GraphList	AddVertex(Vertex v)	GraphList g = new GraphList(boolean d) Vertex v1 = new Vertex(Integer value, Integer k)	g.addVertex(Vertex v1) g.addVertex(Vertex v2) V1: es el nodo que se va a agregar al grafo. V2: es el nodo que se va a agregar al grafo.	Se espera que el método halla agregado satisfactoriamente los dos nodos al grafo.

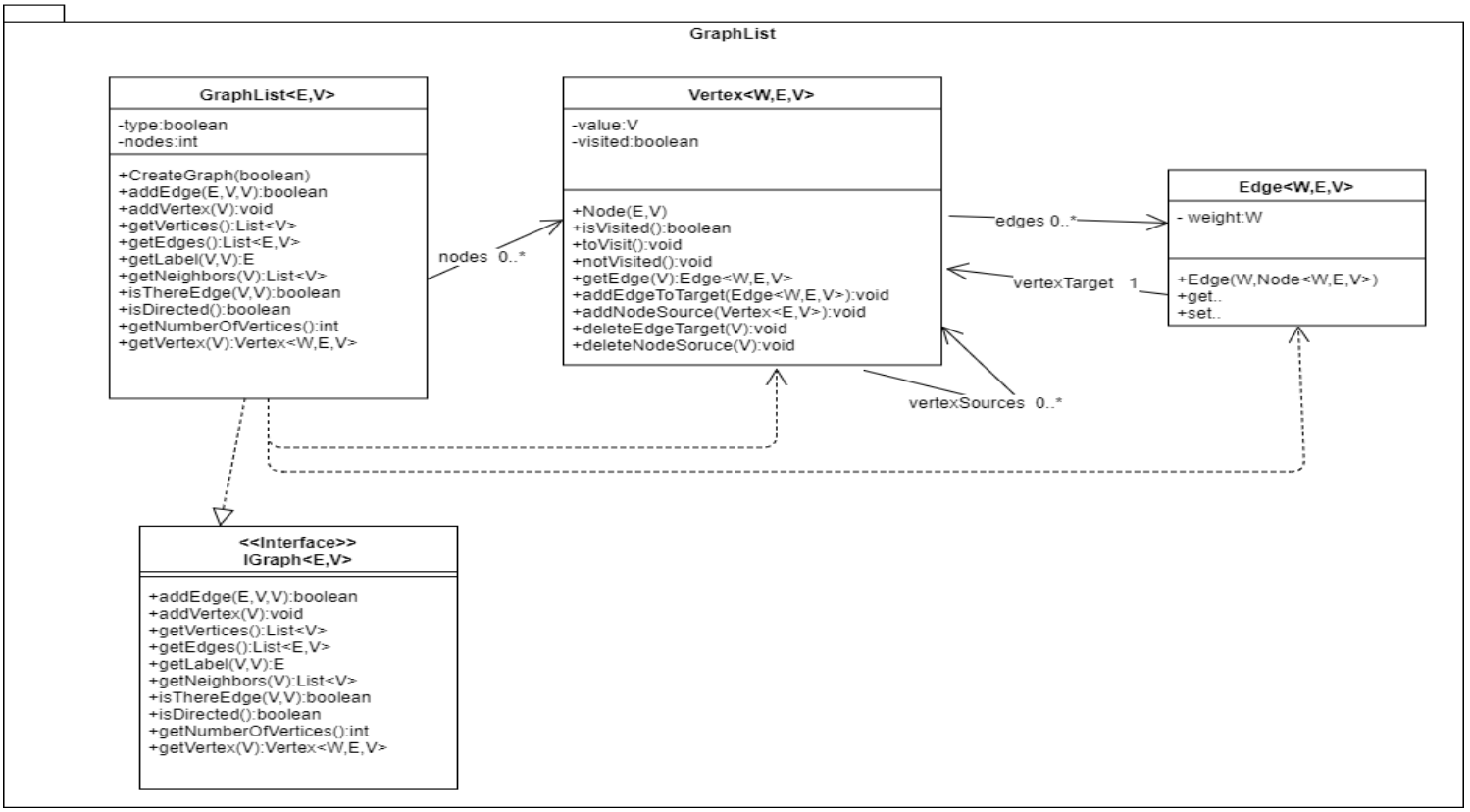
Prueba No. 9	**Objetivo de la prueba:** Comprobar que el método getLabel retorna la arista que une a dos nodos dentro del grafo. **Firma del método:** public Edge getLabel(Vertex v1, Vertex v2)			
Clase para probar	**Método**	**Escenario**	**Valores de entrada**	**Resultado**
GraphList	getLabel(Vertex v1, Vertex v2)	GraphList g = new GraphList(boolean d) g.addVertex(Vertex v1) g.addVertex(Vertex v2) g. AddEdge(Edge e, Vertex v1, Vertex v2)	g.getLabel(Vertex v1, Vertex v2) V1 y V2: son los nodos de los cuales se quiere saber si están o no unidos. .	Se retorna la arista e que une a los nodos v1 y v2. Retorna null si no están unidos.

Prueba No. 10	**Objetivo de la prueba:** Comprobar que el método getNeighbors retorna una lista de todos los nodos vecinos a un nodo x en particular. **Firma del método:** public List getNeighbors (Vertex v1)			
Clase para probar	**Método**	**Escenario**	**Valores de entrada**	**Resultado**
GraphList	getNeighbors (Vertex v1)	GraphList g = new GraphList(boolean d) g.addVertex(Vertex v1) g.addVertex(Vertex v2) g. AddEdge(Edge e, Vertex v1, Vertex v2)	g. getNeighbors(Vertex v1) V1: es el nodo al que se quiere saber cuáles son sus vecinos. .	Se retorna la lista de los vecinos del nodo v1. Su único vecino es el nodo v2.

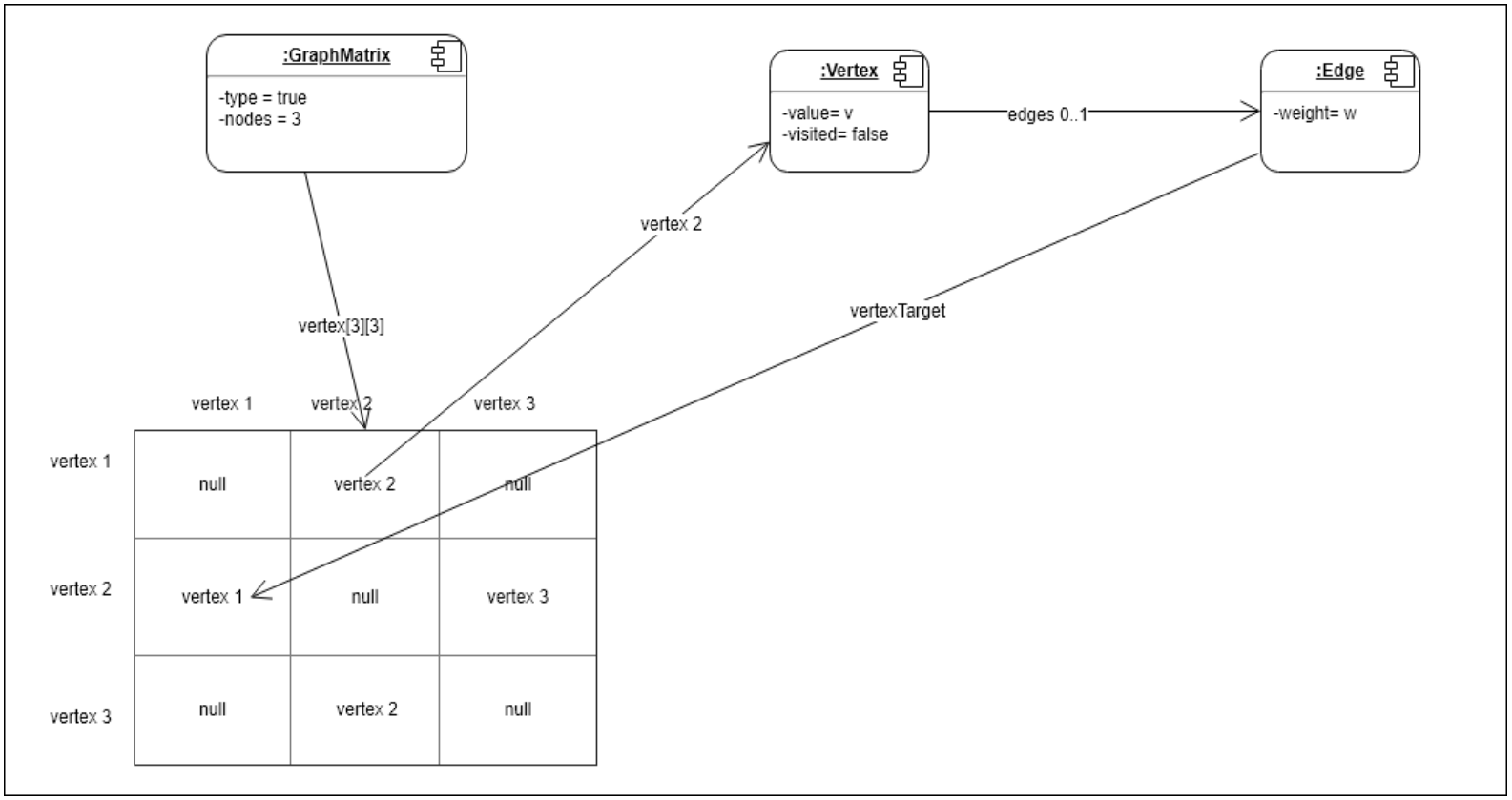
Prueba No. 11	**Objetivo de la prueba:** Comprobar que el método getNumberOfVertices retorna el número de nodos que existen en el grafo. **Firma del método:** public int getNumberOfVertices()			
Clase para probar	**Método**	**Escenario**	**Valores de entrada**	**Resultado**
GraphList	getNumberOfVertices()	GraphList g = new GraphList(boolean d) g.addVertex(Vertex v1) g.addVertex(Vertex v2)	g. getNeighbors(). El método no tiene entrada alguna.	Se retorna el número de nodos que hay en el grafo, en este caso se retorna 2.

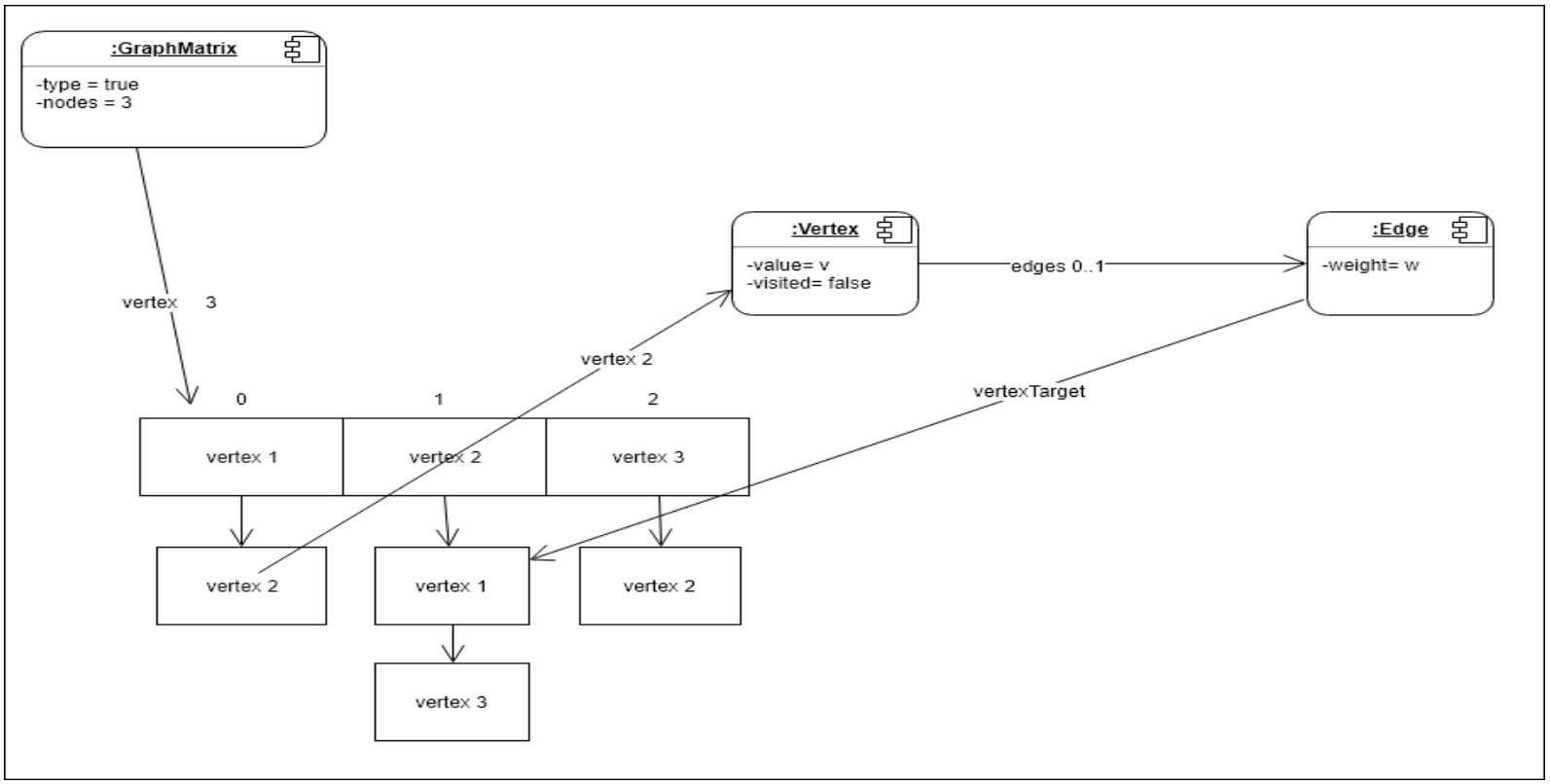
6.4.Diagrama UML





6.5.Diagrama de objetos





7. Implementación del diseño:

Enlace al repositorio en Git: