Proyecto

Algoritmos y estructuras de datos

En el siguiente documento se encuentra documentación relevante que pone en evidencia el planteamiento, desarrollo e implementación de la solución al problema planteado por la empresa de bebidas Triadam Cola, la estructura del texto está basada en la aplicación del método de la ingeniería

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
http://www.maestrosdelweb.com/que-son-las-bases-de-datos/
https://es.wikipedia.org/wiki/Google_Maps
http://tratandodeentenderlo.blogspot.com/2009/09/que-es-maven.html

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.

Base de Datos: Una base de datos es un "almacén" que nos permite guardar grandes cantidades de información de forma organizada para que luego podamos encontrar y utilizar fácilmente.

Google Maps: Es un servidor de aplicaciones de mapas en la web que pertenece a Alphabet Inc. Ofrece imágenes de mapas desplazables, así como fotografías por satélite del mundo e incluso la ruta entre diferentes ubicaciones o imágenes a pie de calle con Google Street View, condiciones de tráfico en tiempo real (Google Traffic) y un calculador de rutas a pie, en coche, bicicleta (beta) y transporte público y un navegador GPS.

Maven: es una herramienta de gestión de proyectos que se basa en la "convención sobre configuración". Es decir, asume un comportamiento por defecto que permite empezar a trabajar sin necesidad de configuración. Por ejemplo, Maven asume la estructura de los ficheros del proyecto, con lo cual a la hora de compilar no se pierde tiempo indicando donde se encuentra el código fuente o las librerías como en Ant.

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.

Alternativa 8: BFS

Al generarse las rutas de los pedidos de la fábrica, se conocerán todos los "vecinos" que van desde la fábrica hasta el último pedido.

4. Transición de las Ideas a los Diseños Preliminares:

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 con el menor peso posible.

Alternativa 8: BFS

- Cuando el algoritmo de Prim genere el nuevo grafo con los caminos más cortos, el BFS utilizará dicho grafo para retornar una cola de los nodos visitados y así saber en qué vértice empezar la entrega de los pedidos.

4.1. Ideas no factibles

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. Sin embargo, no es una buena opción para la solución porque se quiere saber el camino más corto en el que se conecten todos los nodos del grafo.

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:

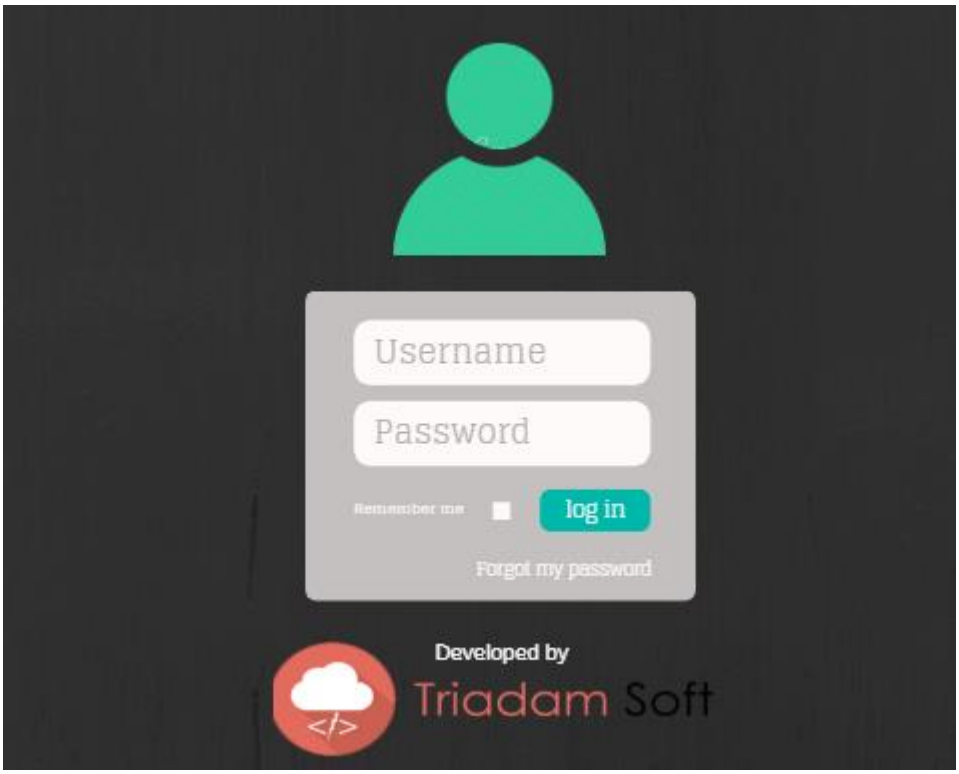
- Aunque las aristas estén organizadas por su peso de menor a mayor, esto no garantiza que el camino sea el más corto para realizar la entrega de los productos.

Alternativa 3: Montículo o heap

- Aunque se obtengas las rutas más cercanas a la fábrica en O(1) esto presentará un problema para los pedidos que no estén cerca de ella porque la idea es estar conectado con todos los pedidos.

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] Medio

[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 5	1	2	3	1	7
Alternativa 8	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

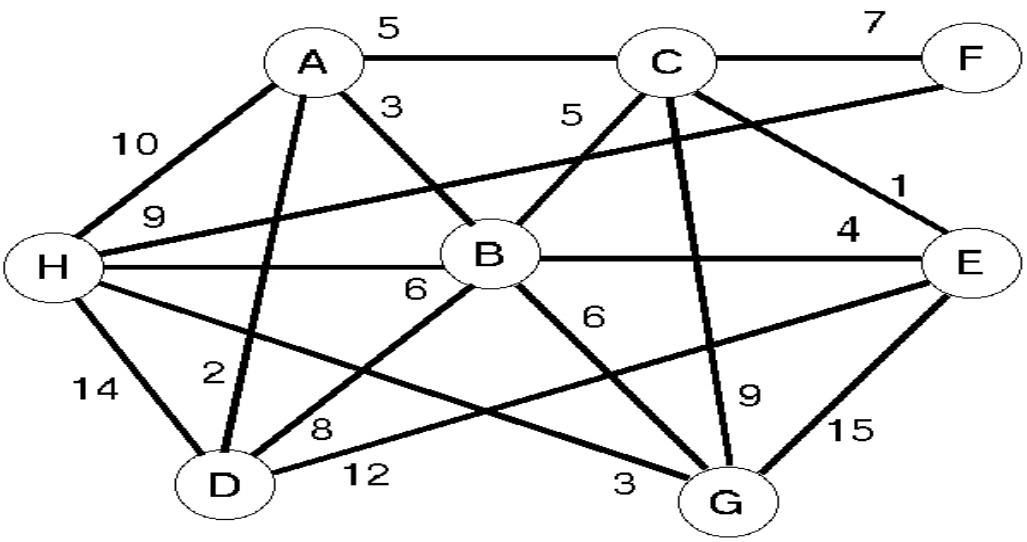
Entrada: Sea P= $\{P_1, P_2, P_3, \dots, P_n\}$ donde P_x es una localización donde se entrega un pedido.

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

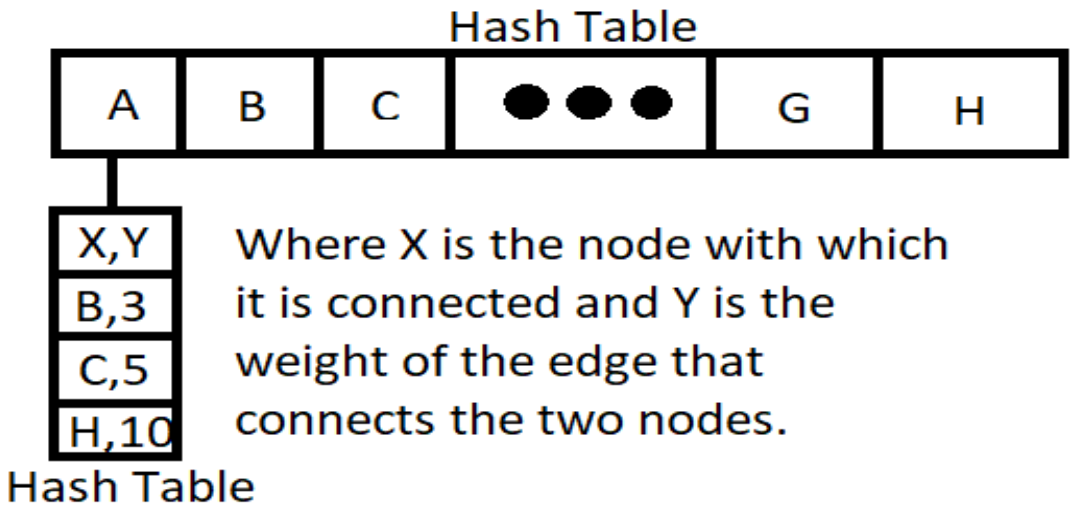
6.2. TAD

TAD Definición

Graph
Graphic representation:



Representation in the code:



Invariants:
n is the number of vertices.
Every Object in the graph belong to the same class

Operations:

Graph	**----> Graph**
getVertices	**----> Hashtable<K, Vertex<I, ID, K, V>>**
setVertices	**----> void**
getNumberOfVertices	**----> int**
setNumberOfVertices	**----> void**

getNumberOfEdges	**----> int**
setNumberOfEdges	**----> void**
getTypeGraph	**-----> char**
setTypeGraph	**-----> void**
getAdjacencyMatrixWeight	**-----> Integer[][]**
setAdjacencyMatrixWeight	**-----> void**
getVertex	**-----> Vertex<I, ID, K, V>**
getEdge	**-----> Edge<I, ID, K, V>**
contains	**-----> boolean**
containsKey	**-----> boolean**
getValue	**-----> V**
getInformationEdge	**-----> I**
getWeightEdge	**-----> int**
isDirectedGraph	**-----> boolean**
existingEdge	**-----> boolean**
add	**-----> boolean**
addEdgeBetweenVertices	**-----> void**
removeEdgeBetweenVertices	**-----> void**
removeVertex	**-----> Vertex<I,ID,K,V>**
generateMatrix	**-----> void**
BFS	**-----> Queue<Vertex<I, ID, K, V>>**
DFS	**-----> Queue<Vertex<I, ID, K, V>>**
Dijkstra	**-----> Integer[]**
FloydWarshall	**-----> Integer[][]**
startNewGraph	**-----> Graph<I, ID, K, V>**
PRIM	**-----> Graph<I, ID, K, V>**
enqueueAllEdges	**-----> void**
KRUSKAL	**-----> Graph<I, ID, K, V>**

Graph(char typeGraph)
Pre: true
Post: A new graph has been created, if typeGraph is I, is undirected, directed otherwise.

getAdjacencyMatrixWeight()

Pre: adjacencyMatrixWeight != null
Post: return Adjacency Matrix

contains(Vertex<I, ID, K, V> vertex)
Pre: true
Post: returns true if the vertex is in the vertices hash table

containsKey(K key)
Pre: true
Post: returns true if the key is in the vertices hash table

getInformationEdge(K keyVertex, ID edgeId)
Pre: Edge!= null
Post: If the edge exists, the information is returned

getWeightEdge(K keyVertex, ID edgeId)
Pre: Edge != null
Post: If the edge exists, the information is returned

IsDirectedGraph()
Pre: True
Post: if the graph is directed, returned true.

existingEdge(K keyVertexSource, K keyVertexTarget)
Pre: True

Post: Returns true if two nodes are connected by an edge.

add(K key, V value)
Pre: The vertex is not in the graph
Post: returns true if the vertex is added to the graph

addEdgeBetweenVertices(K keyVertexSource, K keyVertexTarget, int edgeWeight, ledgeInformation, ID edgeId)
Pre: keyVertexSource != null && keyVertexTarget != null
Post: the two nodes are connected by means of an edge

removeEdgeBetweenVertices(K keyVertexSource, ID edgeId)
Pre: keyVertexSource != null
Post: removes the edge that connects the two nodes

removeVertex(K key)
Pre: vertex != null
Post: that node is removed

generateMatrix()
Pre: Graph != null
Post: adjacency Matrix is created

BFS(K keySourceVertex)
Pre: keySourceVertex != null && Vertex != null
Post: returns the queue of visited nodes

DFS(K keySourceVertex)
Pre: keySourceVertex != null && Vertex != null
Post: returns the queue of visited nodes

Dijkstra(K keySourceVertex)
Pre: keySourceVertex != null && Vertex != null
Post: Returns a minimum distance array of a node to all

FloydWarshall()
Pre: adjacencyMatrixWeight != null
Post: Returns a minimum distance matrix between all the nodes

startNewGraph()
Pre: true
Post: new Graph is created

PRIM(K keyVertex)
Pre: keyVertex != null && vertex != null
Post: returns a new graph with the minimum distance

enqueueAllEdges(PriorityQueue<WayComparator<I, ID, K, V>> pq)
Pre: true
Post: remove all edges

KRUSKAL()
Pre: true
Post: returns a new graph with the minimum distance

6.3.Diseño de pruebas unitarias

Prueba No. 1	**Objetivo de la prueba:** Comprobar que el método add crea y agrega correctamente los vértices al grafo. **Firma del método:** public boolean add(K key, V value)			
Clase para probar	**Método**	**Escenario**	**Valores de entrada**	**Resultado**

Graph	add	Graph<Integer,Integer, Integer, Integer> graph = new Graph<Integer, Integer, Integer,Integer>(Graph.DIRECTED_GRAPH);
		
		graph.add(i,i);
		graph.addEdgeBetweenVertices(i j, 0, 0, edgeId);
		
		0<= i <7
		0<= j <7
		0<= edgeId <49
				Se comprueba que todos los vértices fueron agregados con sus correspondientes llaves y valores.

Prueba No. 2	**Objetivo de la prueba:** Comprobar que el método addEdgeBetweenVertices crea y agrega correctamente las aristas entre los vértices del grafo. **Firma del método:** public void addEdgeBetweenVertices(K keyVertexSource, K keyVertexTarget, int edgeWeight, I edgeInformation, ID edgeId)			
Clase para probar	**Método**	**Escenario**	**Valores de entrada**	**Resultado**

Graph	addEdgeBetweenVertices	Graph<Integer,Integer, Integer, Integer> graph = new Graph<Integer, Integer, Integer,Integer>(Graph.DIRECTED_GRAPH);
		
		graph.add(i,i);
		
		graph.addEdgeBetweenVertices(i j, 0, 0, edgeId);
		
		0<= i <7
		0<= j <7
		0<= edgeId <49
				Se comprueba que fueron creadas las aristas entre todos los vértices.

Prueba No. 3	**Objetivo de la prueba:** Comprobar que el método existingEdge indique correctamente si existe o no al menos una arista entre dos vértices. **Firma del método:** public boolean existingEdge(Vertex<I, ID, K, V> vertexToSearch)			
Clase para probar	**Método**	**Escenario**	**Valores de entrada**	**Resultado**

Graph	existingEdge	Graph<Integer,Integer, Integer, Integer> graph = new Graph<Integer, Integer, Integer,Integer>(Graph.DIRECTED_GRAPH);
		
		graph.add(i,i);
		
		graph.addEdgeBetweenVertices(i j, 0, 0, edgeId);
		
		0<= i <7
		0<= j <7
		0<= edgeId <49
			graph.getVertex(i).existingEdge(graph.getVertex(j)
			
			0<= i <7
			0<= j <7
				Se comprueba que exista al menos una arista entre cada uno de los vértices.

Prueba No. 4	**Objetivo de la prueba:** Comprobar que el método removeEdgeBetweenVertices elimina correctamente la arista indicada de un vértice. **Firma del método:** public void removeEdgeBetweenVertices(K keyVertexSource, ID edgeId))			
Clase para probar	**Método**	**Escenario**	**Valores de entrada**	**Resultado**

Graph	removeEdgeBetweenVertices	Graph<Integer,Integer, Integer, Integer> graph = new Graph<Integer, Integer, Integer,Integer>(Graph.DIRECTED_GRAPH);
		
		graph.add(i,i);
		
		graph.addEdgeBetweenVertices(i j, 0, 0, edgeId);
		
		0<= i <7
		0<= j <7
		0<= edgeId <49
			graph.removeEdgeBetweenVertices(i, edgeId);
			
			0<= i <7
			0<= edgeId <49
				Se comprueba todas las aristas del grafo fueron eliminadas.

Prueba No. 5	**Objetivo de la prueba:** Comprobar que el método removeVertex elimina correctamente el vértice indicado. **Firma del método:** public Vertex<I, ID, K, V> removeVertex(K key)			
Clase para probar	**Método**	**Escenario**	**Valores de entrada**	**Resultado**

Graph	removeVertex	Graph<Integer,Integer, Integer, Integer> graph = new Graph<Integer, Integer, Integer,Integer>(Graph.DIRECTED_GRAPH);
		
		graph.add(i,i);
		
		graph.addEdgeBetweenVertices(i j, 0, 0, edgeId);
		
		0<= i <7
		0<= j <7
		0<= edgeId <49
			graph.removeVertex(i)
			0<= i <7
				Se comprueba todos los vértices fueron eliminados del grafo.

Prueba No. 6	**Objetivo de la prueba:** Comprobar que la estructura genere de forma correcta la matriz de adyacencias que contiene los pesos de las aristas. **Firma del método:** public Integer[][] getAdjacencyMatrixWeight()			
Clase para probar	**Método**	**Escenario**	**Valores de entrada**	**Resultado**

Graph	getAdjacencyMatrixWeight	Graph<Integer,Integer, Integer, Integer> graph = new Graph<Integer, Integer, Integer, Integer>(Graph.INDIRECTED_GRAPH);
		
		Integer[][] matrix =
		{{0 , 12, 40, 12, 32},
		{12, 0, 32, 12, 42},
		{40, 32, 0 , 12, 42},
		{12, 12, 12, 0 , 78},
		{32, 42, 42, 78, 0 }};
		
		graph.add(i,i);
		
		graph.addEdgeBetweenVertices(i, j, matrix[i][j], 0, j);
		0<= i <5
		0<= j <5
				Se comprueba que la matriz de adyacencias generada por la estructura, sea igual a la matriz que se usó para generar el grafo.

Prueba No. 7	**Objetivo de la prueba:** Comprobar que el método BFS realiza el correcto recorrido por un grafo. **Firma del método:** public Queue<Vertex<I, ID, K, V>> BFS(K keySourceVertex)			
Clase para probar	**Método**	**Escenario**	**Valores de entrada**	**Resultado**
Graph	BFS	Graph<Integer,Integer, Integer, Integer> graph = new Graph<Integer, Integer, Integer, Integer>(Graph.INDIRECTED_GRAPH); Integer[][] matrix = {{null, 1 , 2 , null, null, null}, {1 , null, null ,6 , null, null}, {2 , null, null ,7 , 9 , null}, {null, 6 , 7 ,null, null, 10 }, {null, null, 9 ,null, null, null}, {null, null, null ,10 , null, null}};	Queue<Vertex<Integer,Integer,Integer,Integer>> queueBFS = graph.BFS(0)	Se comprueba que el recorrido realizado en el grafo que se generó a partir de la matriz, coincida con un recorrido esperado. Es decir que las colas queueBFS y visited, tengas los mismos elementos, en el mismo orden.

		graph.add(i,i);
		
		visited = {0,1,2,3,4,5}
		graph.addEdgeBetweenVertices(i, j, matrix[i][j], 0, edgeId);
		0<= i <6
		0<= j <6
		0<= edgeId <36		

Prueba No. 8	**Objetivo de la prueba:** Comprobar que el método DFS realiza el correcto recorrido por un grafo. **Firma del método:** public Queue<Vertex<I, ID, K, V>> DFS(K keySourceVertex)			
Clase para probar	**Método**	**Escenario**	**Valores de entrada**	**Resultado**

Graph	DFS	Graph<Integer,Integer, Integer, Integer> graph = new Graph<Integer, Integer, Integer, Integer>(Graph.INDIRECTED_GRAPH);
		
		Integer[][] matrix =
		{{null, 1 , 1 , null, null, null},
		{1 , null, null ,1 , null, null},
		{1 , null, null ,1 , 1 , null},
		{null, 1 , 1 ,null, null, 1 },
		{null, null, 1 ,null, null, null},
		{null, null, null ,1 , null, null}}
		
		graph.add(i,i);
		
		visited = {0,2,3,5,4,1}
		graph.addEdgeBetweenVertices(i, j, matrix[i][j], 0, edgeId);
		0<= i <6
		0<= j <6
		0<= edgeId <36
			Queue<Vertex<Integer,Integer,Integer,Integer>> queueBFS = graph.DFS(0)	Se comprueba que el recorrido realizado en el grafo que se generó a partir de la matriz, coincida con un recorrido esperado. Es decir que las colas queueDFS y visited, tengas los mismos elementos, en el mismo orden.

Prueba No. 9	**Objetivo de la prueba:** Comprobar que el método Dijkstra encuentra el peso mínimo para llegar de un vértice a todos los demás. **Firma del método:** public Integer[] Dijkstra(K keySourceVertex)

Clase para probar	Método	Escenario	Valores de entrada	Resultado

Graph	Dijkstra	Graph<Integer,Integer, Integer, Integer> graph = new Graph<Integer, Integer, Integer, Integer>(Graph.INDIRECTED_GRAPH);
		
		Integer[][] matrix =
		{{0 , 4 , 1 , null, null},
		{4 , 0 , 10 , 6 , null},
		{1 , 10 , 0 , 8 , 6 },
		{null, 6 , 8 , 0 , 5 },
		{null, null, 6 , 5 , 0 }};
		graph.add(i,i);
		
		graph.addEdgeBetweenVertices(i, j, matrix[i][j], 0, edgeId);
		0<= i <5
		0<= j <5
		0<= edgeId <25
			Integer[] distance = {0,4,1,9,7};
			Integer[] distanceDijktra = graph.Dijkstra(0);
				Se comprueba que las distancias generadas por el método Dijkstra coincide con las esperadas.

Prueba No. 10	**Objetivo de la prueba:** Comprobar que el método FloydWarshall genere la matriz con los pesos del recorrido mínimo entre cada uno de los vértices. **Firma del método:** public Integer[][] FloydWarshall()			
Clase para probar	**Método**	**Escenario**	**Valores de entrada**	**Resultado**

Graph	FloydWarshall	Graph<Integer,Integer, Integer, Integer> graph = new Graph<Integer, Integer, Integer, Integer>(Graph.DIRECTED_GRAPH);
		
		Integer[][] matrix =
		{{0 , null, -2 , null},
		{4 , 0 , 3 , null},
		{null, null, 0 , 2 },
		{null, -1 , null, 0 }};
		graph.add(i,i);
		
		graph.addEdgeBetweenVertices(i, j, matrix[i][j], 0, edgeId);
		0<= i <4
		0<= j <4
			Integer[][] distance = {{0, -1,-2, 0},
			{4, 0, 2, 4},
			{5, 1, 0, 2},
			{3, -1, 1, 0}};
			
			Integer[][] distanceFloydWarshall = graph.FloydWarshall();
				Se comprueba que la matriz distance (distancias esperadas) coincida con la matriz distanceFloydWarshall, que contiene los valores generados por el método floydWarshall

		0<= edgeId <16		

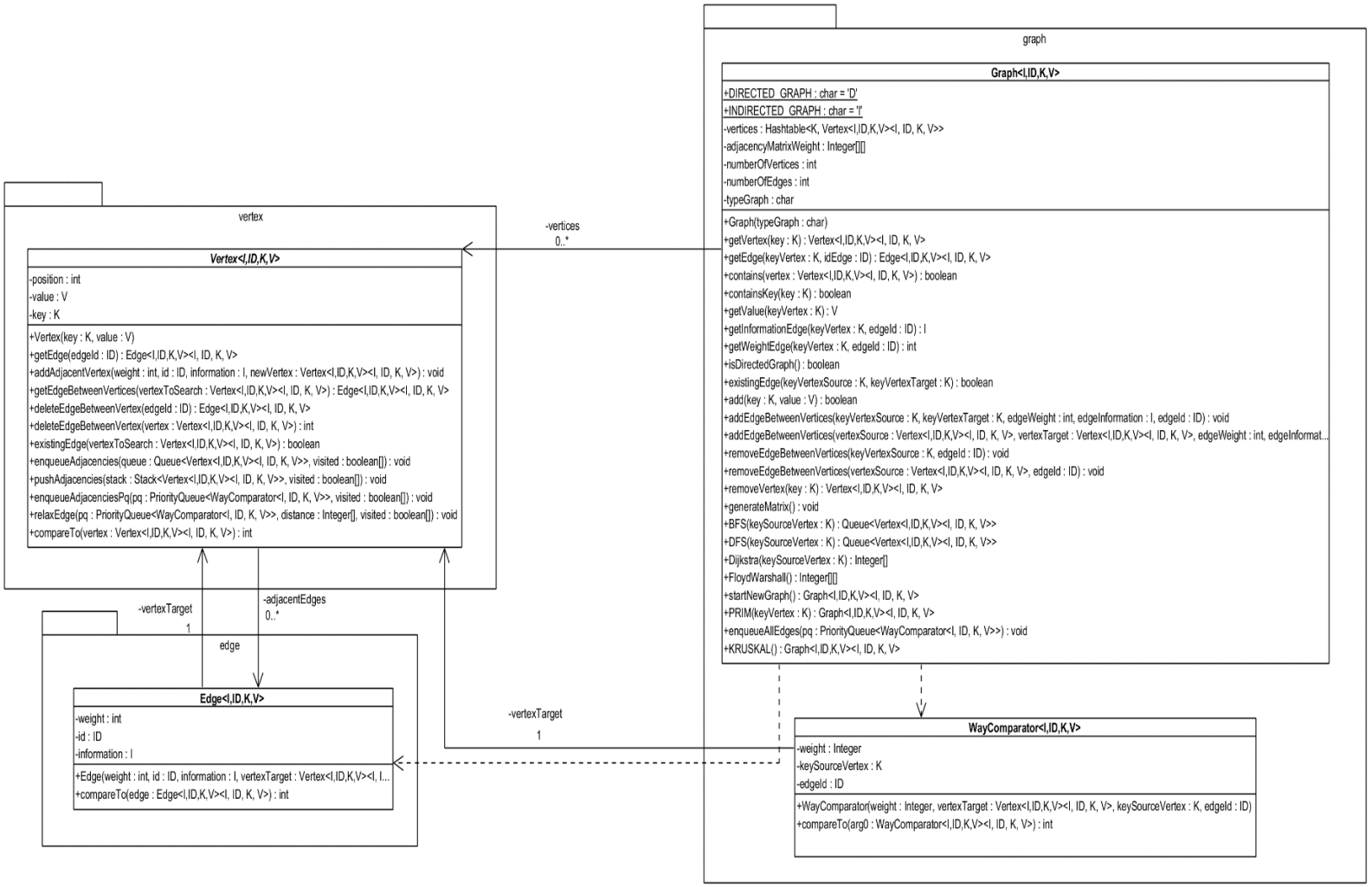
Prueba No. 11	**Objetivo de la prueba:** Comprobar que el método PRIM genera un grafo tal que las aristas que conectan todos los vértices son de peso mínimo. **Firma del método:** public Graph<I, ID, K, V> PRIM(K keyVertex)			
Clase para probar	**Método**	**Escenario**	**Valores de entrada**	**Resultado**

Graph	PRIM	Graph<Integer,Integer, Integer, Integer> graph = new Graph<Integer, Integer, Integer, Integer>(Graph.INDIRECTED_GRAPH);
		
		Integer[][] matrix = {{null, 1 , 2 , null, null, null},
		{1 , null, null ,6 , null, null},
		{2 , null, null ,7 , 9 , null},
		{null, 6 , 7 ,null, null, 10 },
		{null, null, 9 ,null, null, null},
		{null, null, null ,10 , null, null}};
		
		graph.add(i,i);
		
		visited = {0,1,2,3,4,5}
		graph.addEdgeBetweenVertices(i, j, matrix[i][j], 0, edgeId);
		0<= i <6
		0<= j <6
		0<= edgeId <36
			Graph<Integer, Integer, Integer, Integer> graph = this.graph.PRIM(0);
			
			Integer[] dijktraPrePrim = this.graph.Dijkstra(0);
			
			Integer[] dijktraPostPrim = graph.Dijkstra(0);
				Se comprueba que al ejecutar dijkstra desde el vertice que se ejecuta el prim, genera el mismo arreglo de distancias minimas antes y despues de la ejecución del prim.

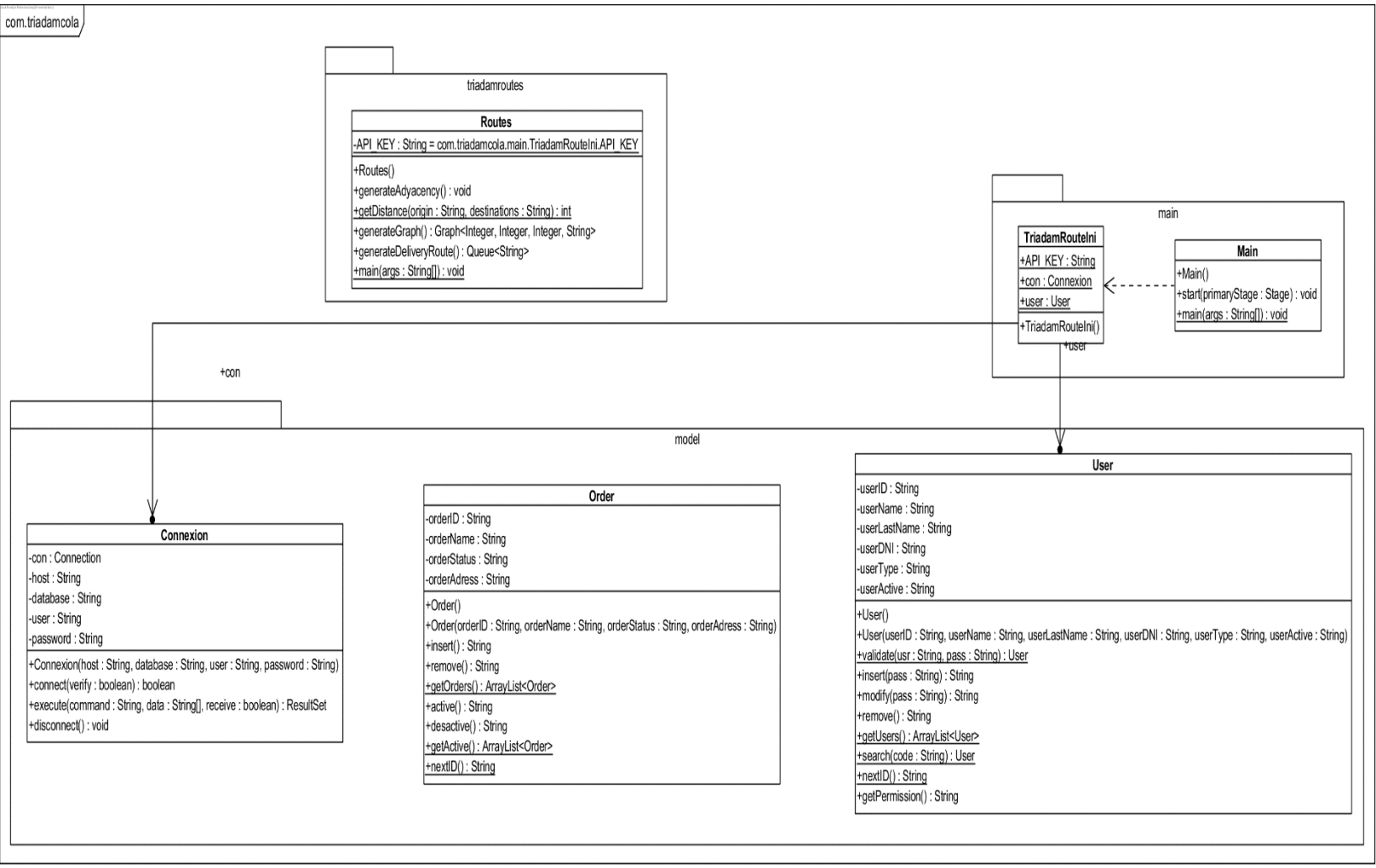
Prueba No. 12	**Objetivo de la prueba:** Comprobar que el método KRUSKAL genera un grafo tal que las aristas que conectan todos los vértices son de peso mínimo. **Firma del método:** public Graph<I, ID, K, V> KRUSKAL()			
Clase para probar	**Método**	**Escenario**	**Valores de entrada**	**Resultado**
Graph	KRUSKAL	Graph<Integer,Integer, Integer, Integer> graph = new Graph<Integer, Integer, Integer,	Graph<Integer, Integer, Integer, Integer> graph =	Se comprueba que la suma de los pesos de todas las aristas pertenecientes al

		Integer>(Graph.INDIRECTED_GRAPH);
		
		Integer[][] matrix =
		{{null, 1 , 2 , null, null, null},
		{1 , null, null ,6 , null, null},
		{2 , null, null ,7 , 9 , null},
		{null, 6 , 7 ,null, null, 10 },
		{null, null, 9 ,null, null, null},
		{null, null, null ,10 , null, null}};
		
		graph.add(i,i);
		
		visited = {0,1,2,3,4,5}
		graph.addEdgeBetweenVertices(i, j, matrix[i][j], 0, edgeId);
		0<= i <6
		0<= j <6
		0<= edgeId <36
			this.graph.KRUSKAL( );	nuevo grafo generado por el método KRUSKAL, coinciden con el peso mínimo esperado para el nuevo grafo.

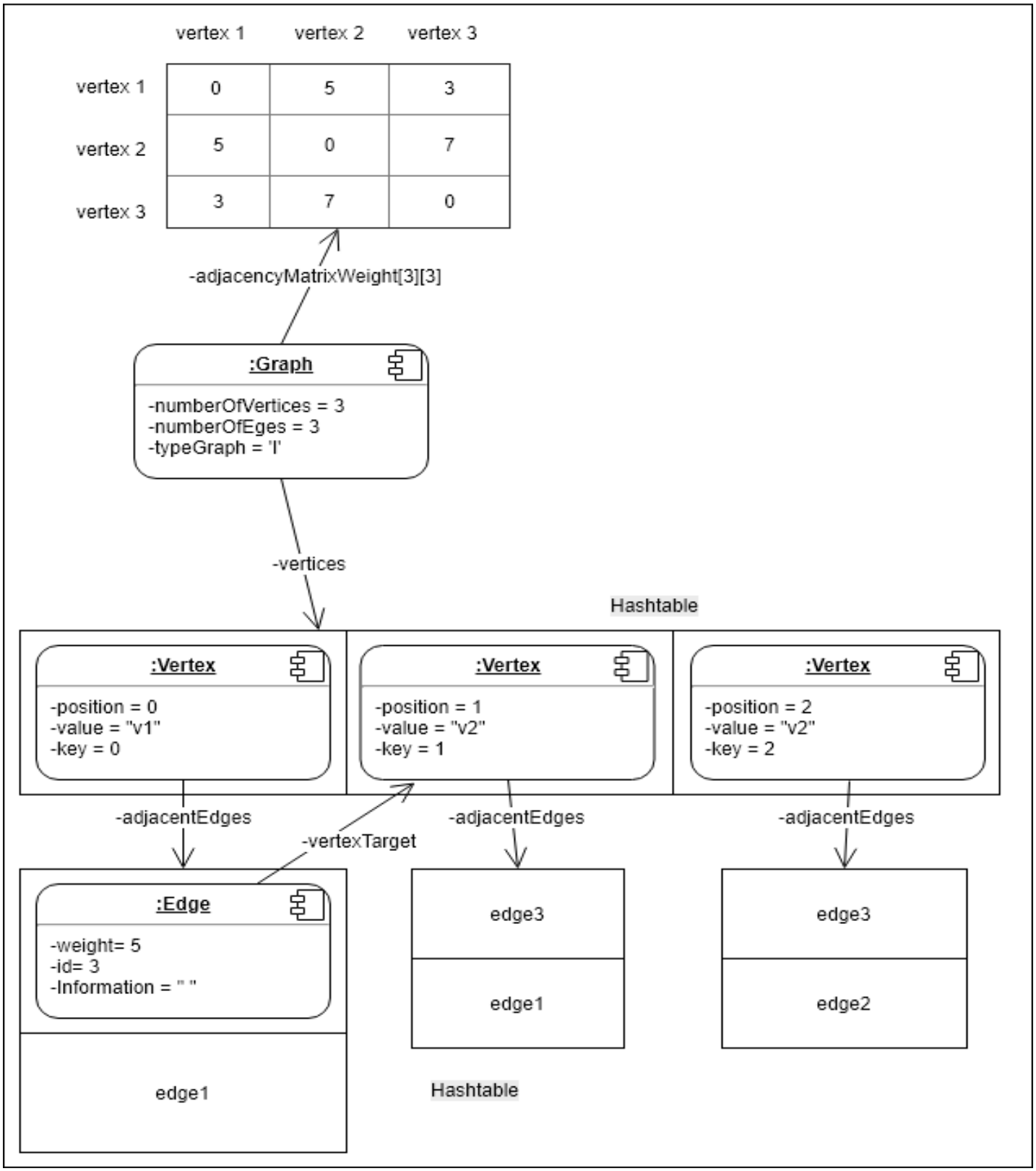
6.4.Diagrama UML del grafo



6.5.Diagrama UML del programa



6.6.Diagrama de objetos del grafo



7. Implementación del diseño:

Enlace al repositorio en GitHub: https://github.com/TheSams117/AED_Project