**Algoritmos y Estructuras de Datos**

**Proyecto de Curso**

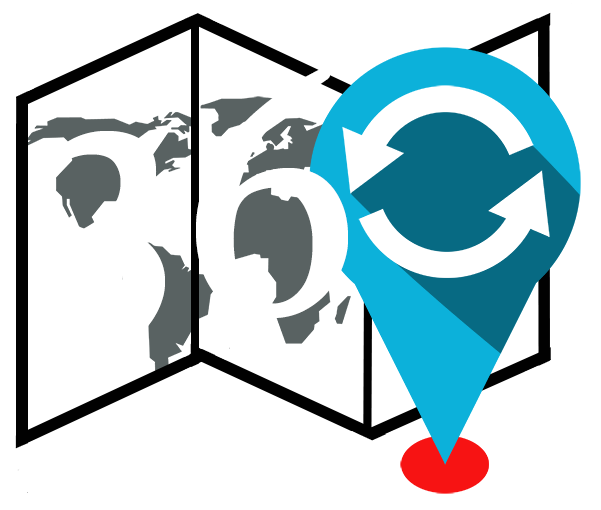
*Nicolás Martinez, Cristian Molina y Juan Manuel Castillo.*

[*url github:*](https://github.com/nicolasmr21/FP_GRAPHS)

*[url Drive](https://docs.google.com/document/d/1EmOzW7odAxO_YLiSXK1zIeJVUtea8kvU0sm00QkGgrc/edit?usp=sharing)*[:](https://docs.google.com/document/d/1EmOzW7odAxO_YLiSXK1zIeJVUtea8kvU0sm00QkGgrc/edit?usp=sharing)

**EL PROBLEMA A SOLUCIONAR:**

***TOUR PLANNING***

Una reconocida aerolínea ha decidido contratar a desarrolladores para implementar un nuevo software que sea capaz de dado ciudades que un cliente quiere visitar en su excursión o viaje, le muestre la ruta más económica o la ruta más corta en distancia entre cada punto que quiere visitar. El trabajo de los desarrolladores es dado un punto de partida y una ciudad destino, encontrar las ruta más económica o corta entre tales ciudades.

Cabe resaltar que el software ya contendrá las ciudades que el usuario puede elegir, tales ciudades son las que están habilitadas por la aerolínea para esta versión.

**Entrada:**

La primera entrada que se requerirá será la de especificar una ciudad de origen, ya que depende de cada ciudad de origen los costos y distancias de los vuelos. La segunda entrada tiene que ver con la ciudad que quiere visitar el usuario, esta va a ser especificada por tal cliente. Por último, se solicitará si desea saber la ruta más corta en distancia o más económica.

**Salida:**

La salida va a ser dependiente de las especificaciones del usuario, pero en el peor caso se mostrará al usuario la ruta más corta o la ruta más económica.

**Paso 1. Identificación del problema**

***Identificación de necesidades y síntomas***

* Encontrar la ruta más económica y corta entre un punto de origen (ciudad de origen) y punto de destino (ciudad destinada).
* Debido a lo sesgada que se encuentran las búsquedas de internet acerca de las promociones que un usuario puede encontrar para sus viajes, es decir, alianzas de páginas con aerolíneas, no permiten ver realmente cuales son las rutas más cortas y económicas.
* Los usuarios de la aerolínea necesitan asesoramiento respecto a las ciudades a visitar durante su excursión.

***Definición del problema***

Una reconocida aerolínea necesita de la implementación de un nuevo software que sea capaz de brindar a sus usuarios que desean visitar en su excursión o viaje ciudades, la información acerca de la ruta más económica o la ruta más corta en distancia entre cada punto que se quiere visitar.

***Requerimientos funcionales***

|  |  |
| --- | --- |
| **Nombre** | **R.F. #1 Calcular la ruta con más corta entre un punto de partida y diferentes ciudades consecutivas.** |
| **Resumen** | **Permite calcular la ruta más corta entre una ciudad origen y una o más ciudades de destino requeridas por el usuario.** |
| **Entrada** | * **Ciudad de origen** * **Mínimo una ciudad de destino.** |
| **Salida** | **Se ha calculado la ruta más corta entre la ciudad de origen y las ciudades de destino.** |

|  |  |
| --- | --- |
| **Nombre** | **R.F. #2 Calcular la ruta más económica en presupuesto entre un punto de partida y diferentes ciudades consecutivas.** |
| **Resumen** | **Permite calcular la ruta más económica entre una ciudad origen y una o más ciudades de destino requeridas por el usuario.** |
| **Entrada** | * **Ciudad de origen** * **Mínimo una ciudad de destino.** |
| **Salida** | **Se ha calculado la ruta más económica entre la ciudad de origen y las ciudades de destino.** |

|  |  |
| --- | --- |
| **Nombre** | **R.F. #3 Contar con una interfaz gráfica de usuario que permita utilizar las funcionalidades que responden a los requerimientos del problema.** |
| **Resumen** | **Permite mostrar una interfaz amigable al usuario para introducir todas sus restricciones.** |
| **Entrada** | **//** |
| **Salida** | **Se ha mostrado la interfaz de usuario.** |

***Requerimientos no funcionales***

|  |  |
| --- | --- |
| **Nombre** | **R.N.F. #1 Desarrollar 2 versiones de Grafo** |
| **Resumen** | **El programa debe admitir el cambio de la implementación utilizada en cualquier momento y funcionar bien indistintamente de la que se esté usando.** |

|  |  |
| --- | --- |
| **Nombre** | **R.N.F. #2 Usar algoritmos conocidos para dar solución al problema.** |
| **Resumen** | **Aplicar al menos dos (2) de los algoritmos de grafos que se estudiarán durante el curso: Recorridos sobre Grafos (BFS, DFS), Caminos de Peso Mínimo (Dijkstra, Floyd-Warshall), Árbol de Recubrimiento Mínimo -MST- (Prim, Kruskal).** |

**Fase 2. Recopilación de la información necesaria**

***Definiciones***

**Fuente:**

**https://www.wikipedia.org/**

**https://www.ecured.cu/**

[**https://edgylabs.com/**](https://edgylabs.com/)

[**https://definicion.de/**](https://definicion.de/)

[**https://www.hackerearth.com/**](https://www.hackerearth.com/)

<https://www.ingenieriaindustrialonline.com/>

*API*

Application Programming Interface. El concepto hace referencia a los procesos, las funciones y los métodos que brinda una determinada biblioteca de programación a modo de capa de abstracción para que sea empleada por otro programa informático.(Fuente: Definición)

*Google Maps Platform*

Es una plataforma de creación de mapas de Google Maps, que incluye el desarrollo de aplicaciones atractivas para la Web y para dispositivos móviles que incluyan instrucciones para llegar en automóvil, imágenes de Street View y más.

*Grafo*

Es una representación gráfica de diversos puntos que se conocen como nodos o vértices, los cuales se encuentran unidos a través de líneas que reciben el nombre de aristas. Al analizar los grafos, los expertos logran conocer cómo se desarrollan las relaciones recíprocas entre aquellas unidades que mantienen algún tipo de interacción. (Fuente: Wikipedia).

*Grafo Simple*

Grafo que no presenta lazos en sus vértices ni más que una arista entre cualquier par de vértices. (Fuente: Ecured).

*Grafo Conexo*

Se dice conexo si, para cualquier par de vértices u y v en G, existe al menos una trayectoria (una sucesión de vértices adyacentes que no repite vértices) de u a v. (Fuente: Wikipedia)

*Grafo Completo*

Un grafo completo es un grafo simple donde cada par de vértices está conectado por una arista. Un grafo completo de n vértices tiene aristas, y se nota . Es un grafo regular con todos sus vértices de grado. (Fuente: Wikipedia).

*Grafo Bipartito*

Un grafo bipartito es un grafo G=(N,E) cuyos vértices se pueden separar en dos conjuntos disjuntos U y V, de manera que las aristas sólo pueden conectar vértices de un conjunto con vértices del otro. (Fuente: Wikipedia).

*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.(Fuente: Wikipedia).

*DFS*

Es un algoritmo recursivo que utiliza la idea de retroceso. Implica búsquedas exhaustivas de todos los nodos al avanzar, si es posible, o al retroceder. A diferencia de BFS, un algoritmo DFS atraviesa un árbol o gráfico desde el vértice principal hasta sus vértices hijos y nietos en una sola ruta hasta que llega a un callejón sin salida. (Fuente: Edgylabs).

*Backtracking*

Backtracking significa que cuando se está moviendo hacia adelante y ya no hay más nodos a lo largo de la ruta actual, se mueve hacia atrás en la misma ruta para encontrar nodos para atravesar. Todos los nodos serán visitados en la ruta actual hasta que todos los nodos no visitados hayan sido atravesados, después de lo cual se seleccionará la siguiente ruta.(Fuente: Hackerearth.).

*Algoritmo de Dijkstra*

Es un modelo que se clasifica dentro de los algoritmos de búsqueda. Su objetivo, es determinar la ruta más corta, desde el nodo origen, hasta cualquier nodo de la red. Su metodología se basa en iteraciones, de manera tal que en la práctica, su desarrollo se dificulta a medida que el tamaño de la red aumenta, dejándolo en clara desventaja, frente a métodos de optimización basados en programación matemática. (Fuente: Ingenieriaindustrialonline)

*Algoritmo de 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.(Fuente: Ecured).

**Fase 3. Búsqueda de soluciones creativas**

El grupo ha concertado entender el entorno en que se presenta el problema acordando utilizar como plataforma para la implementación del programa, la plataforma ó API que brinda Google Maps en la que se obtendrán los mapas necesarios y, con base a estos, incluir los algoritmos de grafos necesarios para el modelamiento del problema. Dentro las estrategias propuestas para la implementación de los viajes podría ser:

* Utilizando Recorridos sobre Grafos(BFS, DFS)
* Utilizando Caminos de Peso Mínimo(Dijkstra, Floyd-Warshall)
* Utilizando Árbol de Recubrimiento Mínimo(Prim, Kruskal)

**ALTERNATIVAS PARA RECORRIDO DEL GRAFO:**

**Recorrido en profundidad:** Recorrer un grafo consiste en “visitar”cada uno de los nodos representados en ciudades a través de las aristas, es decir, rutas del mismo. Se trata de realizar recorridos de grafos de manera eficiente. Para ello, se pondría una marca en un nodo(ciudad) en el momento en que es visitado, de tal manera que, no se repitan rutas ya utilizadas.

**Recorrido en amplitud:** Se establece un una jerarquía partiendo de la raíz, dividiendo el grafo en niveles de acuerdo a la cercanía de ciudades vecinas, al realizar este proceso las ciudades se van marcando con color gris, blanco o negro (por lo general) para darle a entender al recorrido cuales ya se visitaron y cuales se encuentran próximos a visitar sus sucesores.

**ALTERNATIVAS PARA LA RESOLUCIÓN DEL PROBLEMA:**

En base a los algoritmos vistos en clase y los planteados en el enunciado del proyecto, se encuentran dos problemáticas de las cuales se pueden partir, en las que, una de las dos se seleccionará en pro del acoplamiento que deberá tener en base a nuestro enunciado del problema.

**Problema del camino más corto:** En la teoría de grafos,el problema de los caminos más cortos es el problema que consisten 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 es encontrar el camino más rápido para ir de una ciudad a otra en un mapa. En este caso, los vértices representan las ciudades, y las aristas las carreteras que las unen, cuya ponderación viene dada por el tiempo que se emplea en atravesarlas.

Dentro de este paradigma, se encuentran disponibles:

* **Algoritmo de Dijkstra -** Resuelve el problema de los caminos más cortos desde un único vértice origen hasta todos los otros vértices del grafo
* **Algoritmo de Bellman - Ford** Resuelve el problema de los caminos más cortos desde un origen si la ponderación de las aristas es negativa.
* **Algoritmo de Búsqueda A\*** Resuelve el problema de los caminos más cortos entre un par de vértices usando la heurística para intentar agilizar la búsqueda.
* **Algoritmo de Floyd - Warshall** Resuelve el problema de los caminos más cortos entre todos los vértices.
* **Algoritmo de Johnson -** Resuelve el problema de los caminos más cortos entre todos los vértices y puede ser más rápido que el de Floyd - Warshall en grafos de baja densidad.

**Problema Árbol de recubrimiento mínimo:** Dado un grafo conexo, no dirigido y con pesos en las aristas, un árbol de recubrimiento mínimo es un árbol compuesto por todos los vértices y cuya suma de sus aristas es la de menor peso.

Dentro de este paradigma, se encuentran los más conocidos:

* **Algoritmo de Prim -** El algoritmo encuentra un subconjunto de aristas que forman un árbol con todos los vértices, donde el peso total de todas las aristas en el árbol es el mínimo posible. Si el grafo no es conexo, entonces el algoritmo encontrará el árbol recubridor mínimo para uno de los componentes conexos que forman dicho grafo no conexo.
* **Algoritmo de Kruskal -** 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).

**Fase 4. Transición de la formulación de ideas a los diseños preliminares**

Para la transformación de búsqueda de soluciones creativas a diseños preliminares planteamos una tabla con criterios los cuales nos permiten entender mejor el funcionamiento de cada posible estrategia de recorrido a usar obteniendo un total en base a los criterios cumplidos, seleccionando los que mayor puntaje haya obtenido.

* **Criterio 1.**  La complejidad es inferior o igual a O(v**²)**
* **Criterio 2.**  Encuentra el camino más corto en base al valor de sus aristas
* **Criterio 3.** Puede ser implementado en grafos simples
* **Criterio 4.**  Pinta de colores los nodos para una búsqueda más eficiente.
* **Criterio 5.** Realiza la búsqueda usando la arista(ruta) con menor valor como partida.

**Fase 5. Evaluación y selección de la mejor solución**

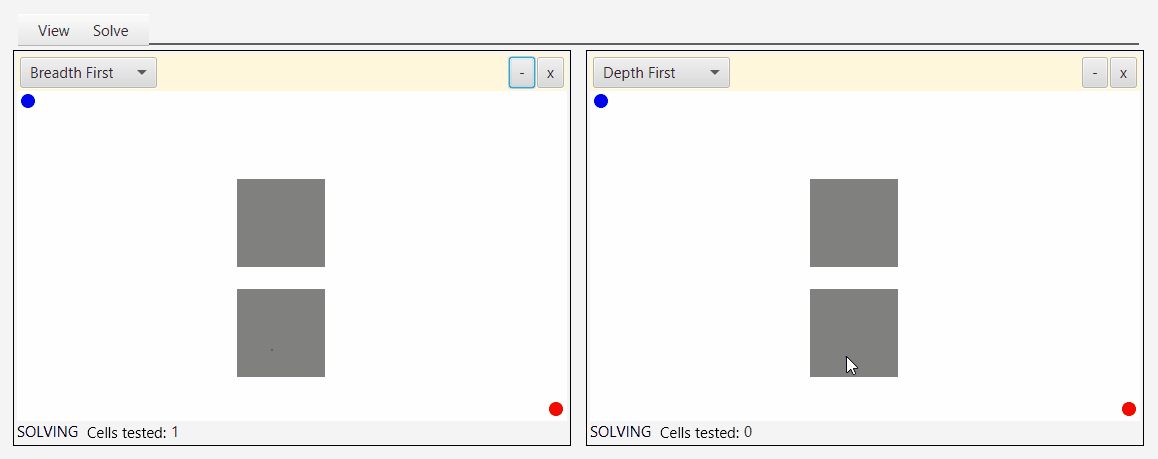
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Estructura** | **Criterio 1** | **Criterio 2** | **Criterio 3** | **Criterio 4** | **Criterio 5** | **TOTAL** |
| **BFS** | x |  | x | x |  | 3 |
| **DFS** | x |  | x | x |  | 3 |
| **Dijkstra** | x | x | x | x |  | 4 |
| **Warshall** |  | x |  | x |  | 2 |
| **Kruskal** | x | x |  | x | x | 4 |

**Fase 5. Evaluación y selección de la mejor solución**

* Se puede observar un empate entre los algoritmos de Dijkstra y de Kruskal por lo cual es pertinente usar ambos. Cada uno perteneciente a un paradigma distinto es decir, el primero mencionado de caminos de peso mínimo y el segundo de árbol de recubrimiento mínimo. Sin embargo, uno de los requerimientos no funcionales propuestos por el enunciado general invita a usar otro paradigma, en este caso, en segunda posición se encuentran empatados los algoritmos de BFS y DFS pertenecientes al recorrido sobre grafos.

***Mejor solución de acuerdo a la problemática planteada***

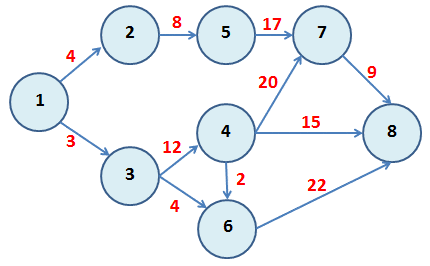
*Funcionamiento del algoritmo BFS y DFS*



En el ejemplo se observan dos puntos, uno de origen (azul) y el de destino (rojo).En el BFS toma mucho más tiempo a comparación de DFS ya que este recorre todos los caminos posibles horizontalmente, y hasta no evaluar la diagonal de vértices no puede seguir con la siguiente diagonal. Sin embargo, a pesar de evaluar más nodos, es esto lo que le permite obtener el camino más corto de un nodo a otro. El concepto de eficacia vs eficiencia es evidente en el que el BFS realiza el papel de eficiencia y el DFS el de eficacia al encontrar un camino del nodo azul al rojo en el menor tiempo posible. La desventaja de estos dos algoritmos es que cuenta las aristas como pasos, es decir, no tiene en cuenta el peso de las rutas.

*Funcionamiento del algoritmo Dijkstra*

Observe como ejemplo el grafo que se encuentra abajo, en donde, los nodos o vértices representados en nodos pueden tomarse como las ciudades enumeradas. Entre cada ciudad hay una ruta o arista definida en kilómetros por el número en rojo Para este caso supongamos que 1 representa la ciudad A y queremos llegar a la ciudad B es decir, la número 8.

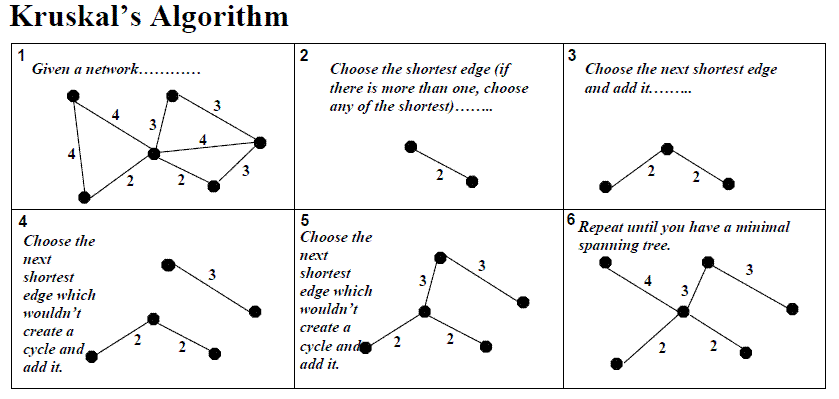
**

El algoritmo entra en juego recibiendo el nodo origen (Nodo Ciudad A) y nodo destino (Nodo Ciudad B) en el que evaluará todas las alternativas posible para llegar al camino deseado teniendo como determinante el menor valor posible en kilómetros de distancia de una ciudad a otra.

* **Ruta** **1-2-5-7-8:** 4+8+17+9=38 km
* **Ruta 1-3-4-7-8:** 3+12+20+9=44 km
* **Ruta 1-3-4-6-8:** 3+12+2+22=39 km
* **Ruta 1-3-4-8:** 3+12+15= 30 km
* **Ruta 1-3-6-8:** 3+4+22= 29 km

A pesar de haber sido una búsqueda relativamente “lenta”, ha encontrado el camino más corto de la ciudad A a la B atravesando los nodos o ciudades necesarias para obtener 29 km de recorrido.

*Funcionamiento del Algoritmo de Kruskal*

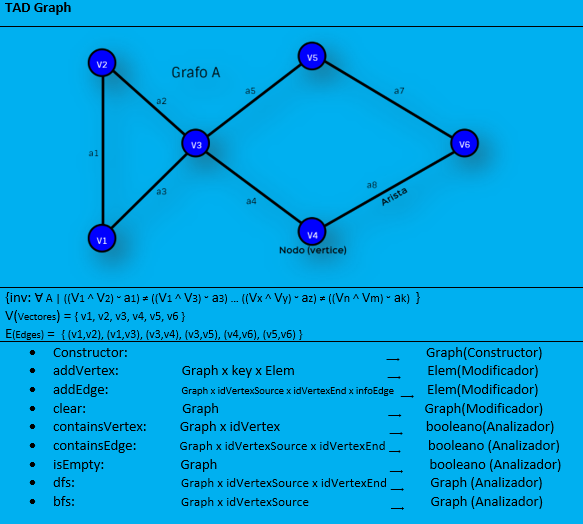
**

Fuente:https://stackoverflow.com/questions/1195872/kruskal-vs-prim

El algoritmo de Kruskal y el algoritmo de árbol de expansión mínimo de Prim son dos algoritmos populares para encontrar los árboles de expansión mínimos, sin embargo, en este caso nos centraremos en el algoritmo de Kruskal.  
  
El algoritmo de Kruskal utiliza un enfoque diferente para encontrar un árbol de expansión mínimo. El algoritmo de Kruskal trata cada vértice como un árbol independiente y se conecta uno con otro solo si tiene el costo más bajo en comparación con todas las demás opciones disponibles.  
  
De acuerdo a la imagen de arriba imaginemos que cada vértice representa una ciudad y sus aristas, una vez más, las rutas. En su viaje al país visitado, planea visitar todos las ciudades importantes, pero les falta poco tiempo. Para hacer que su itinerario funcione, usted decide utilizar el algoritmo de Kruskal.  
  
Simplifiquemos el mapa(grafo) convirtiéndolo en un gráfico como se muestra a continuación y nombrando las ciudades importantes en el mapa con letras y distancia en metros (x 100).  
Comprendamos cómo se usa el algoritmo de Kruskal en el ejemplo del mundo real utilizando el mapa anterior.  
  
**Paso 1**: Eliminar todos los bucles y bordes paralelos para no volver a pasar por ciudades en las que ya realizaron una visita.  
  
**Paso 2:** Buscar la ciudad con menor recorrido y a partir de ahi seguir buscando los caminos más cortos en el mapa(grafo)  
  
**Paso 3**: Representar la conexión de una ruta representada desde la ciudad "x" hasta la ciudad "y"   
  
**Paso 4**: Repetir el paso anterior hasta encontrarnos con el camino por todas las ciudades más corto.  
  
Esto nos da el siguiente gráfico(paso 6), que es el árbol de expansión mínimo para el problema dado.

**Fase 6. Preparación de informes y especificaciones . Diseño de diagrama de clases.**

***Diseño de TADs***

******

|  |
| --- |
| **addVertex( *V element, K key* )**    **“Agrega un vértice (nodo) al grafo”**    **{pre: Se debe introducir como parámetro el elemento y su índice a agregar.}**    **{post: Se agrega el vértice en caso de que no exista otro con los mismos valores. De no ser así; se lanza una excepción informando el error .}** |

|  |
| --- |
| **addEdge( *K idVertexSource, K idVertexEnd, A infoEdge*)**    **“Se agrega una arista(Edge) al grafo”**    **{pre:Se debe introducir como parámetro el vértice de origen y el de destino final.}**    **{post: Se agrega la arista que une el camino entre dos vértices en caso de que existan, de no ser así; se lanzará una excepción como también si la arista ya existe.}** |

|  |
| --- |
| **clear()**    **“Remueve o elimina todos los vértices, y por tanto, aristas que componen el grafo.”**    **{pre: Debe existir el grafo para la implementación del método.}**    **{post: Se han eliminado los elementos (vértices y aristas) del grafo.}** |

|  |
| --- |
| **containsVertex(*K idVertex*)**    **“Prueba la existencia de un vértice en el grafo”**    **{pre:Se introduce por parámetro el índice del vértice a ser buscado.}**    **{post: Verdadero si el vértice existe y falso en caso contrario.}** |

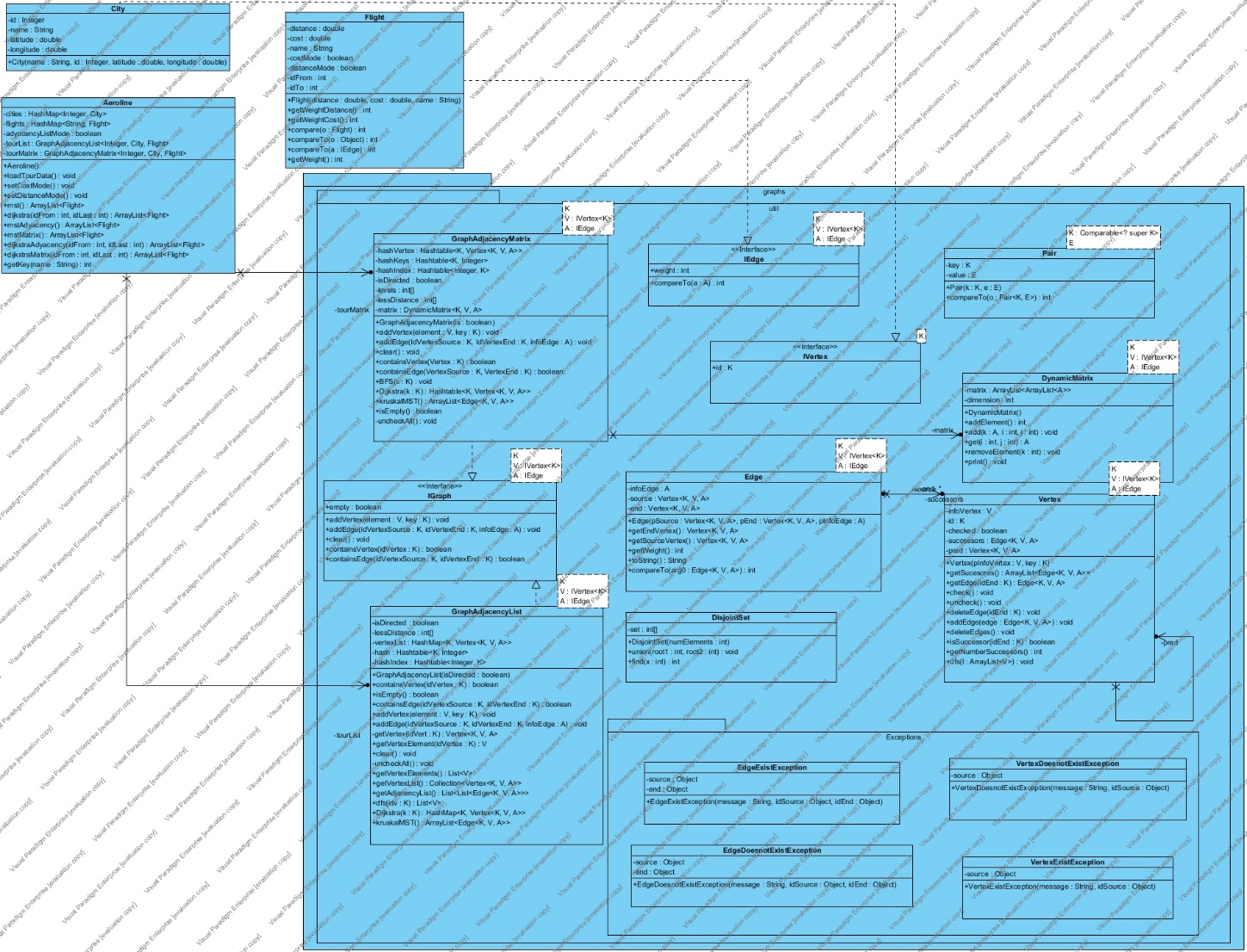
|  |
| --- |
| **containsEdge( *K idVertexSource, K idVertexEnd* )**      **“Prueba la existencia de una arista que une dos vértices en el grafo”**    **{pre:Se introduce por parámetro el índice del vértice de origen y el de llegada.}**    **{post: Verdadero si la arista existe y falso en caso contrario.}** |

|  |
| --- |
| **isEmpty()**    **“Determina si un grafo se encuentra vacío.”**    **{pre: Debe existir el grafo para la implementación del método.}**  **{post: true si se encuentra vacío y falso en caso contrario.}** |

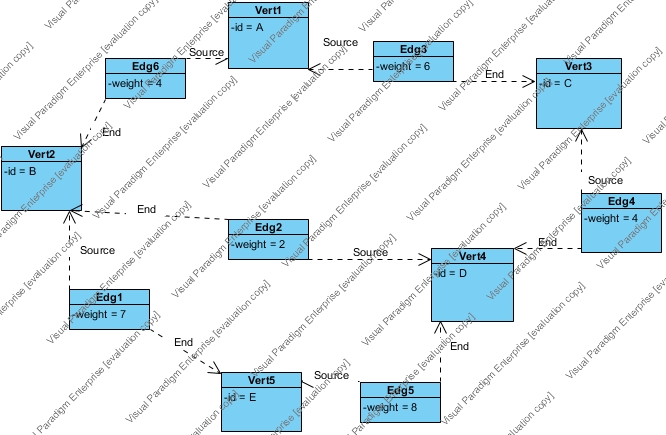
|  |
| --- |
| **dfs( *K idVertexSource*)**    **“Busca un camino entre dos vértices, desde el origen al destino. El orden es importante al ser un grafo dirigido, marcando los caminos ya recorridos.”**  **{pre: Se debe introducir como parámetro el vértice donde iniciará la búsqueda}**  **{post: Lista con los vértices que componen el camino ordenados en el orden a visitar, o lista vacía de no existir camino.}** |

|  |
| --- |
| **bfs( *K idVertexSource*)**    **“Realiza la búsqueda desde el nodo de origen dado por parámetro y se exploran todos los nodos conectados por medio de una arista. Marcando con un color característico los ya visitados, repitiendo el mismo paso hasta recorrer todo el grafo.”**  **{pre: Se debe introducir como parámetro el nodo de origen donde partirá la búsqueda del grafo}**  **{post: Lista con los vértices que componen el camino ordenados en el orden a visitar, o lista vacía de no existir camino.}** |

**Diagrama de clases:**

****

**Diagrama de objetos:**

****

***Diseño de pruebas***

*AdyacencyTest*

***Scene1():*** *Un nodo llamado Medellín y otro nodo llamado Cali con arista correspondiente de 620 (Cali-Medellín) que representa los kilómetros en distancia de un vértice a otro y un coste de 1’000.000 de pesos.*

***Scene2():*** *Un nodo llamado Cartagena con arista de 760 (Cartagena-Medellín) que representa los kilómetros de un vértice a otro y un coste de 3’000.000 de pesos.*

***Scene3():*** *Un nodo llamado Pereira y otro llamado Cali los cuales no tienen conexión con alguno de los vértices agregados anteriormente.*

|  |  |  |  |
| --- | --- | --- | --- |
| Método | Escenario | Entrada | Salida |
| **addVertexTest()** | *Scene1()*  *Scene2()*  *Scene3()* | Ninguna. | Se prueba que se agreguen correctamente los tres nodos propuestos en los 2 primeros escenarios. En el 3ro deberá lanzar una excepción ya que el nodo “Cali” ya existe. |
| **addEdgeTest()** | *Scene1()*  *Scene2()*  *Scene3()* | Ninguna. | Se prueba que se agreguen correctamente las aristas de los escenarios 1 y 2. El escenario 3 deberá lanzar una excepción al no contar con aristas indicará que no existe. |
| **clearTest()** | *Scene1()*  *Scene2()*  *Scene3()* | Ninguna. | Se prueba que se pueda borrar los nodos y aristas almacenados anteriormente. |
| **containsVertexTest()** | *Scene1()*  *Scene2()*  *Scene3()* | Ninguna. | Se prueba que por medio de un booleano afirme si hay o no vértices (nodos) en el grafo) |
| **containsEdgeTest()** | Scene1()  Scene2()  Scene3() | Ninguna. | Se prueba que por medio de un booleano se confirme si posee o no aristas como lo es en el caso 1-2 y lo contrario en el 3. |
| **isEmptyTest()** | *Scene1()*  *Scene2()*  *Scene3()* | Ninguna. | Se prueba por medio de un booleano si el grafo se encuentra vacío. |
| **DFSTest()** | *Scene1()*  *Scene2()* | Ninguna. | Se prueba que haya un camino entre dos vértices, desde el origen al destino, el orden es importante al ser un grafo dirigido. |
| **testKruskal()** | Ninguno. | Ninguna. | Se prueba mediante el algoritmo de kruskal la adición tanto de aristas como de vértices instanciados manualmente, necesarios para su implementación, a su vez, invocando a las excepciones en caso de ser necesarias. |
| **testDijkstra()** | Ninguno. | Ninguna. | Se prueba mediante el algoritmo de Dijkstra la adición tanto de aristas como de vértices instanciados manualmente, necesarios para su implementación, a su vez, invocando a las excepciones en caso de ser necesarias. |

*MatrixTest*

***Scene1():*** *Un nodo llamado Medellín con id “204”, latitud 12312.2 y longitud 12312.1 y otro nodo llamado Cali con id “1002”, latitud 1356.0 y longitud 1342.1 con arista correspondiente(Cali-Medellín) mediante la aerolínea Avianca.*

***Scene2():*** *Un nodo llamado Cartagena con id “340”, latitud 12312.1 y longitud 6545.2 con arista correspondiente(Cartagena-Medellín) mediante la aerolínea Avianca.*

***Scene3():*** *Un nodo llamado Pereira con id “237”, latitud 12312.8 y longitud 43534.3 el cual no tiene conexión con alguno de los vértices agregados anteriormente.*

|  |  |  |  |
| --- | --- | --- | --- |
| Método | Escenario | Entrada | Salida |
| **addVertexTest()** | *Scene1()*  *Scene2()*  *Scene3()* | Ninguna. | Se prueba que se agreguen correctamente los tres nodos propuestos en los 2 primeros escenarios ademas de los nodos Bogotá y Palmira. En el 3ro deberá lanzar una excepción ya que el nodo ya existe. |
| **addEdgeTest()** | *Scene1()*  *Scene2()*  *Scene3()* | Ninguna. | Se prueba que se agreguen correctamente las aristas de los escenarios 1 y 2. teniendo conexión con la ciudad de Bogotá.. El escenario 3 deberá lanzar una excepción al no contar con aristas indicará que no existe. |
| **clearTest()** | *Scene1()*  *Scene2()*  *Scene3()* | Ninguna. | Se prueba que se pueda borrar los nodos y aristas almacenados anteriormente. |
| **containsVertexTest()** | *Scene1()*  *Scene2()*  *Scene3()* | Ninguna. | Se prueba que por medio de un booleano afirme si hay o no vértices (nodos) en el grafo) |
| **containsEdgeTest()** | Scene1()  Scene2()  Scene3() | Ninguna. | Se prueba que por medio de un booleano se confirme si posee o no aristas como lo es en el caso 1-2 y lo contrario en el 3. |
| **isEmptyTest()** | *Scene1()*  *Scene2()*  *Scene3()* | Ninguna. | Se prueba por medio de un booleano si el grafo se encuentra vacío. |
| **BFSTest()** | *Scene1()*  *Scene2()* | Ninguna. | Se prueba que haya un camino entre dos vértices, desde el origen al destino, el orden es importante al ser un grafo dirigido. |
| **testKruskal()** | Ninguno. | Ninguna. | Se prueba mediante el algoritmo de kruskal la adición tanto de aristas como de vértices instanciados manualmente, necesarios para su implementación, a su vez, invocando a las excepciones en caso de ser necesarias. |
| **testDijkstra()** | Ninguno. | Ninguna. | Se prueba mediante el algoritmo de Dijkstra la adición tanto de aristas como de vértices instanciados manualmente, necesarios para su implementación, a su vez, invocando a las excepciones en caso de ser necesarias. |