

Terminales de Transporte M.I.O

Enunciado

Viendo los buenos resultados que obtuvieron en la realización de un prototipo de gestión de bases de datos para la Facultad de Ingeniera de la Universidad. Su equipo ha sido contactado por **METROCALI S.A** para realizar un prototipo de un nuevo proyecto para **Masivo de Integrado de Occidente MIO**, que busca mejorar la red de terminales del sistema de transporte de la ciudad de Santiago de Cali.

El proyecto busca mejorar la distancia entre las terminales y estaciones de transporte, proporcionando la información de las mejores rutas entre cada una de las existentes, como también hacer supuestos donde se agreguen terminales o estaciones si la ruta mas corta existente no cumple las expectativas.

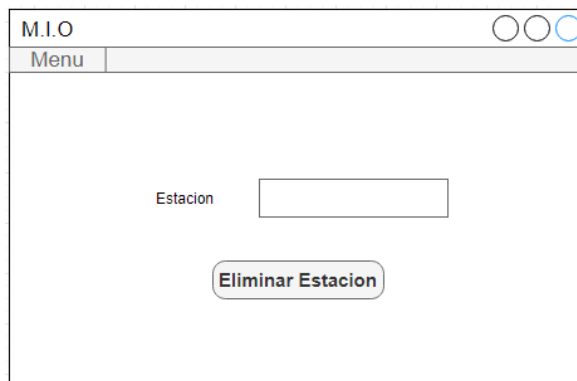
Usted y su equipo tendrán que desarrollar un software que se encargue de estudiar la ruta mas corta entre dos estaciones, como también, debe contar con tres operaciones principales, agregar, eliminar y buscar.



The screenshot shows a window titled 'M.I.O' with a 'Menu' button. The main area contains two text input fields labeled 'Punto inicial:' and 'Punto final:'. Below these fields is a button labeled 'Estimar Ruta'.

Agregar estación/terminal:

El programa debe contar con la opción de agregar cuantas estaciones se necesite buscando mejorar alguna ruta. Debe permitir conectar la nueva estación desde una ya existente y a su vez, a las terminales “cercanas”



The screenshot shows a window titled 'M.I.O' with a 'Menu' button. The main area contains a text input field labeled 'Estacion'. Below this field is a button labeled 'Eliminar Estacion'.

Eliminar estación/terminal:

El programa debe permitir eliminar una estación/terminal del sistema simulando que ya no será tomada en cuenta en la línea del sistema integrado de transporte en la ciudad. Debe permitir eliminar una estación o terminal del sistema sin afectar la conexión entre todas las pertenecientes a la línea de transporte.

Buscar estación/terminal

El programa debe contener la opción de poder buscar una estación/terminal del sistema para determinar si existe dentro del sistema integrado de transporte. Si no existe debe permitir agregarlo conectado a cualquier estación existente y las demás “cercanas”

Su programa, debe tener una interfaz grafica simple con el usuario, que permita ingresar fácilmente una estación/terminal como punto de inicio como también una de punto final. Si el usuario no agrega un punto final el programa deberá calcular la mejor ruta desde el punto de inicio hasta la ultima estación/terminal, solo pasando una vez por cada estación.

La interfaz gráfica deberá mostrar una lista con el nombre de las estaciones (existentes o agregadas por el usuario en el sistema), las cuales se tuvieron en cuenta al momento de buscar la mejor ruta posible.

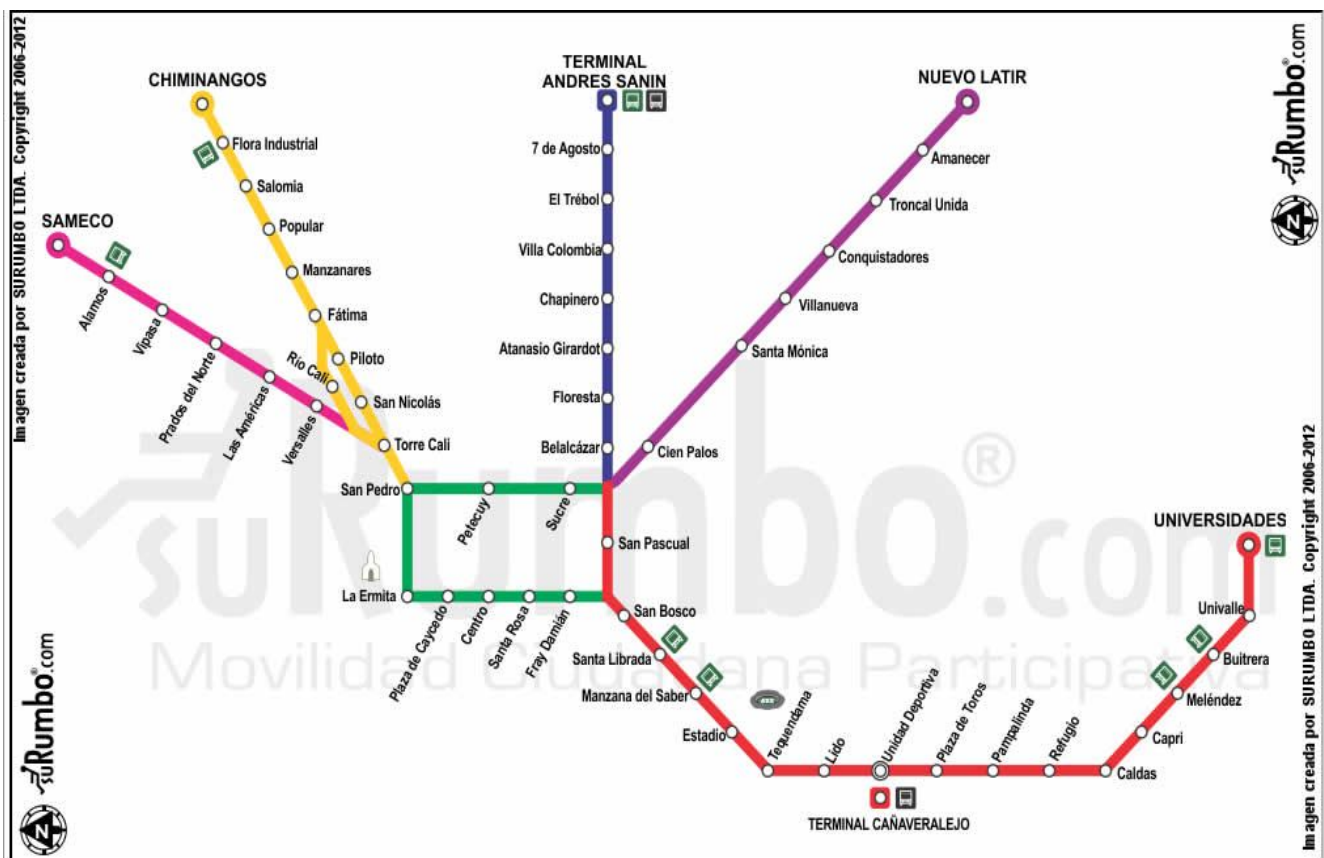
Deberá tener en cuenta, que, para llevar a cabo este prototipo, se le solicita utilizar una estructura de datos vista en clase llamada **GRAFOS**; como también se le solicita tener dos implementaciones (representaciones) para que el comité ejecutivo pueda escoger la solución que crean es la mejor opción para llevarla a cabo en el proyecto. Las implementaciones de la estructura deben ser completamente genéricas.

Recuerde que, las dos implementaciones deben estar completamente probadas, en consecuencia, su equipo deberá proveer un diseño e implementación de pruebas unitarias para cada representación hecha; demostrando que, dicha estructura responde de la mejor manera para solucionar el problema.

Contexto del problema

El **Masivo Integrado de Occidente (MIO)** es el sistema integrado de transporte masivo (SITM) de la ciudad colombiana de Santiago de Cali. El sistema es operado por buses articulados, padrones y complementarios, los cuales se desplazan por medio de corredores troncales, pretroncales y complementarios cubriendo rutas troncales, pretroncales y alimentadoras. Fue inaugurado el 15 de noviembre de 2008 en fase de prueba. A partir del 1 de marzo de 2009 empezó su funcionamiento en firme.

Se tiene el trabajo de realizar un prototipo que le ayude a determinar a **Metrocali S.A** las rutas más cortas entre las estaciones existentes del sistema de transporte o simular el sistema de transporte con más terminales conectadas entre sí.



1. Mapa de una de las rutas del MIO

Etapa N°1: Identificación del problema

- **Necesidades:**
 - Identificar los vértices y las aristas.
 - Identificar el peso de las aristas.

Una vez identificado los vértices del grafo y las aristas con sus respectivos pesos se procede a identificar los algoritmos pertinentes que satisfacen la solución:

Búsqueda en anchura	BFS
Búsqueda en profundidad	DFS
Camino Mínimo	Dijkstra
	Floyd Warshall
Árbol recubridor	Prim
	Kruskal

Requerimientos Funcionales

- *Buscar la ruta más corta entre dos estaciones:* El programa debe tener la capacidad de estimar el camino (ruta) más corto entre dos vértices (estaciones), un punto de inicio y otro de final. *El programa recibe:* un vértice como punto inicial, y un vértice como punto final.
- *Buscar la ruta más corta entre todas las estaciones:* El programa debe ser capaz de estimar el camino (ruta) más corto entre un vértice (estación) como punto de inicio y los demás vértices (estaciones) del grafo. *El programa recibe:* un vértice como punto inicial.
- *Agregar un vértice (estación) al grafo:* El programa permite agregar un vértice(estación) que requiera el usuario, y convirtiendo dicho vértice en adyacentes a otros, cambiando las aristas entre los vértices a partir del lugar donde se agrega. *El programa recibe:* un vértice que indica el lugar del grafo donde se va a agregar, un vértice nuevo que será agregado a la estructura de datos.
- *Eliminar un vértice (estación) del grafo:* El programa cuenta con la funcionalidad que permite eliminar un vértice del grafo, sin afectar la conexión entre los demás vértices del grafo, es decir, los nodos adyacentes del nodo eliminado pasarán a tener aristas con otros vértices. *El programa recibe:* un vértice que se va a eliminar de la estructura y el sistema.

- Buscar un vértice (estación) en el grafo: El programa permite buscar y verificar si un vértice pertenece al grafo utilizado en la solución del problema. *El programa recibe:* un vértice que será buscado en el grafo.

Requerimientos No Funcionales

- El programa debe ser persistente. Cada vez que se corra el programa se debe poder cargar la información utilizada la última vez que se utilizó el sistema, y guardar toda la nueva información suministrada.
- El programa no debe ser demasiado pesado, para que cualquier equipo pueda manejar toda la información necesaria sin restricciones.
- El programa debe mostrar una barra de progreso o de carga, cuando se estime una ruta, apenas se completa la carga deberá mostrar la ruta en pantalla que equivale al camino más corto
- La interfaz gráfica debe ser simple e intuitiva, que permite un fácil manejo de software.

Etapa N°2 Recopilación de información.

La **teoría de grafos**, también llamada **teoría de gráficas**, es una rama de las matemáticas y las ciencias de la computación que estudia las propiedades de los grafos.¹

Grafo: es una estructura de datos no lineal que consta de nodos y aristas. Los nodos a veces también se conocen como vértices y los edge son líneas o arcos que conectan dos nodos en el gráfico. Más formalmente, un grafo se puede definir como:

- *“Un gráfico consiste en un conjunto finito de vértices (o nodos) y un conjunto de bordes que conectan un par de nodos.”²*

Vértice: un vértice o nodo es la unidad fundamental de la que están formados los grafos. Los dos vértices que conforman una arista se llaman **puntos finales** ("endpoints", en inglés), y esa arista se dice que es **incidente** a los vértices. Un vértice w es **adyacente** a otro vértice v si el grafo contiene una arista (v,w) que los une.³

¹https://es.wikipedia.org/wiki/Teoría_de_grafos

²<https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/>

³[https://es.wikipedia.org/wiki/Vértice_\(teoría_de_grafos\)#:~:text=En%20teoría%20de%20grafos%2C%20un.que%20están%20formados%20los%20grafos.&text=Un%20vértice%20w%20es%20adyacente.los%20vértices%20adyacentes%20a%20v.](https://es.wikipedia.org/wiki/Vértice_(teoría_de_grafos)#:~:text=En%20teoría%20de%20grafos%2C%20un.que%20están%20formados%20los%20grafos.&text=Un%20vértice%20w%20es%20adyacente.los%20vértices%20adyacentes%20a%20v.)

Arista: una arista corresponde a una relación entre dos vértices de un grafo.

Para caracterizar un grafo G son suficientes únicamente el conjunto de todas sus aristas, comúnmente denotado con la letra E (del término en inglés edge), junto con el conjunto de sus vértices, denotado por V . Así, dicho grafo se puede representar como $G(V,E)$, o bien $G = (V,E)$.⁴

Camino: secuencia de vértices dentro de un grafo tal que exista una arista entre cada vértice y el siguiente. Se dice que dos vértices están conectados si existe un camino que vaya de uno a otro, de lo contrario estarán desconectados. Dos vértices pueden estar conectados por varios caminos. El número de aristas dentro de un camino es su longitud. Así, los vértices adyacentes están conectados por un camino de longitud 1, y los segundos vecinos por un camino de longitud 2.

Si un camino empieza y termina en el mismo vértice se le llama ciclo.⁵

Etapa N°3 Búsqueda de soluciones creativas.

Para este punto, podemos pensar en soluciones propias, pero con la intención de lograr la mejor solución y obtener varias ideas en la implementación de los grafos hemos optado por realizar una lluvia de ideas creativas.

1. Desarrollar una interfaz en la que el usuario pueda crear y mover las estaciones (grafos) a su antojo.
2. Basarse en una implementación de grafos que sea similar al problema que deseamos solucionar.
3. Desarrollar una interfaz en la que las estaciones (grafos) están predefinidas y se puedan activar y desactivar.
4. Desarrollar una interfaz en la que las estaciones (grafos) están predefinidas y se puede realizar una estimación del tiempo que puede tardar un vehículo en llegar a esa estación.

Etapa N°4 Transición de ideas a diseños preliminares.

En este punto decidimos descartar dos opciones y evaluar las dos restantes para ver cual era la mejor solución.

Opciones descartadas:

1. Desarrollar una interfaz en la que el usuario pueda crear y mover las estaciones (grafos) a su antojo.

⁴[https://es.wikipedia.org/wiki/Arista_\(teoría_de_grafos\)#:~:text=En%20teoría%20de%20grafos%2C%20una%20dos%20vértices%20de%20un%20grafo.&text=En%20un%20grafo%2C%20dos%20vértices,es%20incidente%20a%20dicha%20arista.](https://es.wikipedia.org/wiki/Arista_(teoría_de_grafos)#:~:text=En%20teoría%20de%20grafos%2C%20una%20dos%20vértices%20de%20un%20grafo.&text=En%20un%20grafo%2C%20dos%20vértices,es%20incidente%20a%20dicha%20arista.)

⁵[https://es.wikipedia.org/wiki/Camino_\(teoría_de_grafos\)](https://es.wikipedia.org/wiki/Camino_(teoría_de_grafos))

Para la implementación de una interfaz en la que el usuario arrastre y mueva las estaciones a su gusto, resultaría inviable debido a la complejidad de las librerías de JavaFx para este tipo de aplicaciones y futuro mantenimiento del código.

3. Desarrollar una interfaz en la que las estaciones (grafos) están predefinidas y se puedan activar y desactivar.

En un contexto real, ninguna estación puede estar desactivada o inactiva y el usuario no puede desactivar la gran mayoría de estas. Esto porque el sistema de transporte integrado MIO debe atender una gran demanda de usuarios y no tendría sentido desactivar todas o la gran mayoría de estaciones que los usuarios usan para moverse.

Opciones que aprobamos como futura solución:

2. Basarse en una implementación de grafos que sea similar al problema que deseamos solucionar.

La realidad es que la implementación de la estructura de datos grafo y sus algoritmos tiene un nivel de complejidad de nivel intermedio-alto (como el algoritmo dijkstra). Por eso, para ahorrar tiempo en el proyecto y trabajar desde una base, nos sale muy rentable esta solución.

4. Desarrollar una interfaz en la que las estaciones (grafos) están predefinidas y se puede realizar una estimación del tiempo que puede tardar un vehículo en llegar a esa estación.

Una de las principales quejas de los usuarios del sistema de transporte integrado MIO es la demora de los buses y las malas estimaciones de tiempo. Esta implementación tiene una buena aproximación del problema en la vida real.

Etapas N°5 Evaluación y selección de la mejor solución.

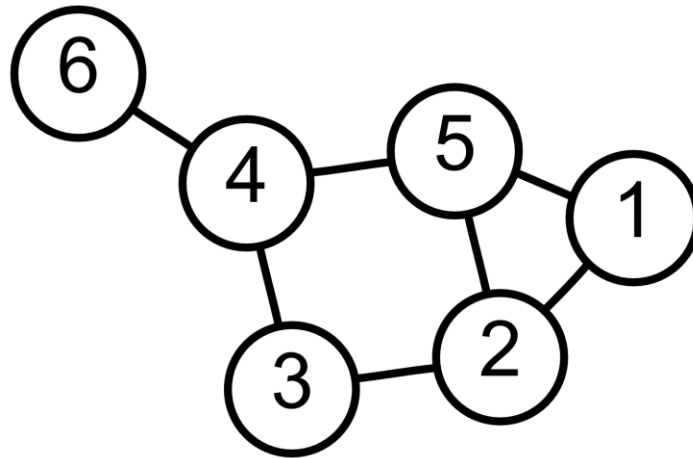
La solución ganadora obtuvo el mayor puntaje en cada uno de los siguientes criterios:

1. Ruta de aprendizaje: Apropiación de los conocimientos de la estructura de datos grafo y sus algoritmos.
2. Aproximación de la realidad: La solución tiene una buena aproximación del contexto del problema.
3. Nivel de dificultad: Nivel de dificultad para implementar la solución.
4. Uso de buenas prácticas: La solución está basada en unas buenas prácticas.

Solución.	Criterio 1	Criterio 2	Criterio 3	Criterio 4	Total
2	5	5	10	5	25
4	10	10	5	10	35

La solución que decidimos implementar fue la número 4. Porque, posee una apropiación más fuerte de los conocimientos necesarios para implementar la estructura de datos grafo y sus algoritmos. Por otro lado, cumple con una aproximación más acertada del contexto del problema. Sin embargo, la solución 2 posee un nivel de implementación más sencillo porque nos basamos en una solución implementada por otras personas, así que solo tendríamos que adaptar el código para la solución del problema. Finalmente, como desarrolladores conocemos buenas prácticas de programación que podemos implementar para que nuestra estructura de datos grafo sea más “limpia”, mientras que si nos basamos en la implementación de otros desarrolladores, podemos cometer el error de usar malas prácticas para la adaptación del código.

TAD Grafo



Invariante del grafo: $G = (V, E)$

Operaciones primitivas:

- CrearGrafo: $\rightarrow G$
- AgregarVertice: $G \times V \rightarrow G$
- AgregarArista: $G \times E \rightarrow G$
- EliminarVertice: $G \times V \rightarrow G$
- EliminarArista: $G \times E \rightarrow G$
- EstaVacio: $G \rightarrow \text{Booleano}$
- RetornarAristas: $\rightarrow \text{Entero}$
- RetornarVertices: $\rightarrow \text{Entero}$
- EsDirigido: $G \rightarrow \text{Booleano}$
- DarPeso: $E \rightarrow \text{Entero}$

Operaciones primitivas constructoras.

CrearGrafo()

“Crea un nuevo grafo vacío.”

{pre: TRUE}

{post: Se crea un grafo vacío}

Operaciones primitivas modificadoras.

AgregarVertice(Grafo, vértice)

“Agrega un vértice a un grafo”

{pre: $G \neq \text{NIL}$ }

{post: El vértice es agregado al grafo}

AgregarArista(Grafo, arista)

“Agrega una arista a un grafo”

{pre: $G \neq \text{NIL}$ }

{post: La arista es agregada al grafo}

Operaciones primitivas destructoras.

EliminarVertice(Grafo, vértice)

“Elimina un vértice de un grafo”

{pre: $G \neq \text{NIL}$ }

{post: Elimina un vértice del grafo}

EliminarArista(Grafo, arista)

“Elimina una arista de un grafo”

{pre: $G \neq \text{NIL}$ }

{post: Elimina una arista del grafo}

Operaciones primitivas analizadoras.

EstaVacio(Grafo)

“Comprueba si el grafo no posee vértices ni aristas”

{pre: $G \neq \text{NIL}$ }

{post: Retorna TRUE si el grafo este vacío. De lo contrario retorna FALSE.

RetornarAristas(Grafo)

“Retorna el número de aristas de un grafo”

{pre: $G \neq \text{NIL}$ }

{post: Retorna el número de aristas del grafo. De lo contrario, retorna 0}

RetornarVertices(Grafo)

“Retorna el número de vértices de un grafo”

{pre: $G \neq \text{NIL}$ }

{post: Retorna el número de vértices del grafo. De lo contrario, retorna 0}

EsDirigido(Grafo)

“Retorna un valor booleano indicando si el grafo es dirigido”

{pre: $G \neq \text{NIL}$ }

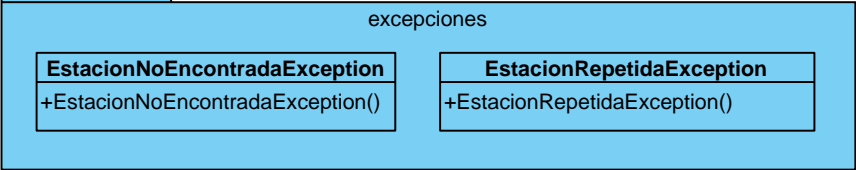
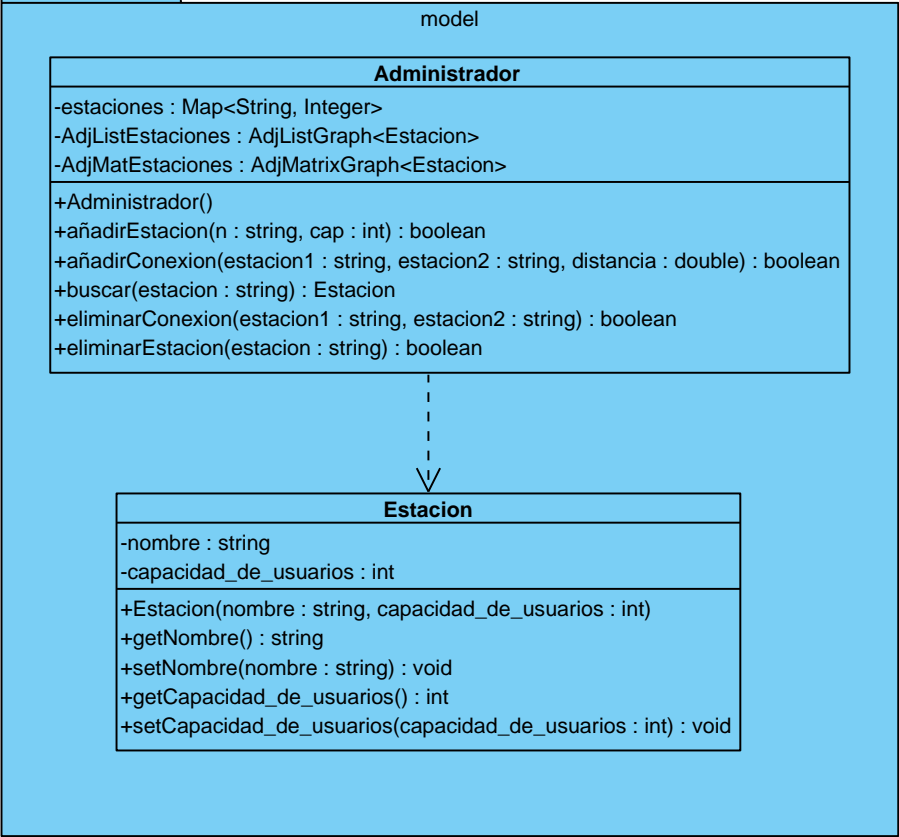
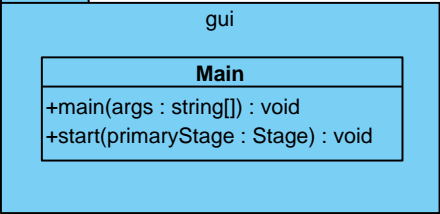
{post: Retorna TRUE si el grafo es dirigido. De lo contrario, retorna FALSE}

DarPeso(Arista)

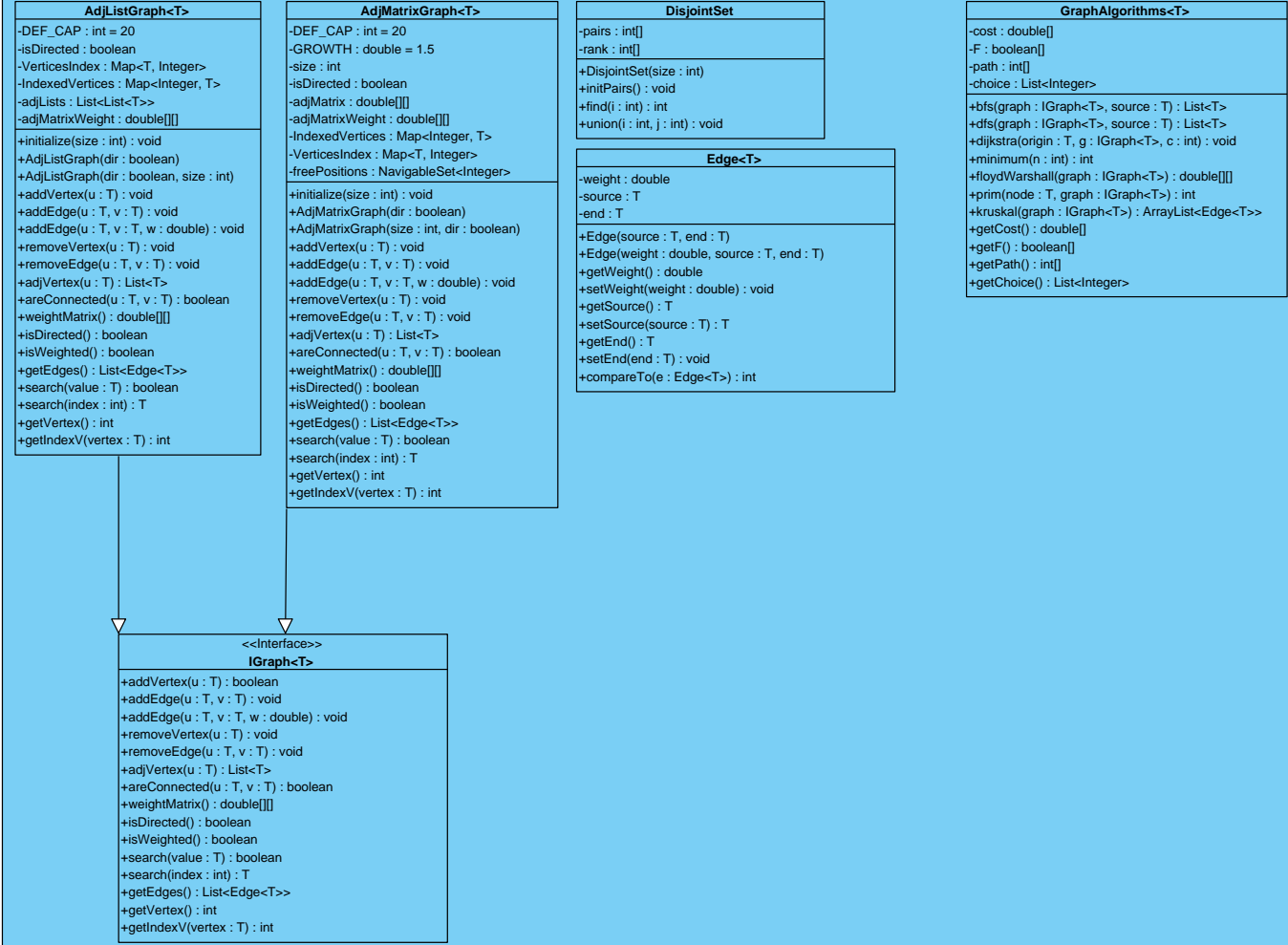
“Retorna el peso de una arista”

{pre: $E \neq \text{NIL}$ }

{post: Retorna el peso de la arista. De lo contrario, retorna 0}



data_structures



Diseño de Pruebas Unitarias

Escenarios:

Nombre	Clase	Método
setUpStage1	AdjListGraphTest	new AdjListGraph<String>(false)
setUpStage2	AdjListGraphTest	new AdjListGraph<String>(false) addVertex("Pueblo Paleta") addVertex("Ciudad Verde") addVertex("Ciudad Plateada")
setUpStage3	AdjListGraphTest	new AdjListGraph<String>(false) addVertex("Pueblo Paleta") addVertex("Ciudad Verde") addVertex("Ciudad Plateada") alg.addEdge("Pueblo Paleta", "Ciudad Verde", 8) alg.addEdge("Ciudad Verde", "Ciudad Plateada", 15)

Pruebas:

Clase	Método	Escenario	Entrada	Salida
AdjListGraphTest	addVertexTest()	setUpStage1	"Pueblo Paleta"	Correcto, porque "Pueblo Paleta" es el primer vértice del grafo.
AdjListGraphTest	addEdgeTest1()	setUpStage2	("Pueblo Paleta", "Ciudad Verde") ("Ciudad Verde", "Ciudad Plateada")	Correcto, porque se las aristas que se agregan en la prueba conectan, "Pueblo Paleta" con "Ciudad Verde" y "Ciudad Verde" con "Ciudad Plateada"
AdjListGraphTest	addEdgeTest2()	setUpStage2	("Pueblo Paleta", "Ciudad Verde") ("Ciudad Verde", "Ciudad Plateada")	Correcto, porque se la arista que se agrega en la prueba conecta "Pueblo Paleta" con "Ciudad Verde" y "Ciudad Verde", "Ciudad Plateada" no son adyacentes.
AdjListGraphTest	deleteVertexTest()	setUpStage2();	("Pueblo Paleta")	Correcto, porque en la prueba se elimina un vértice del escenario que al buscarlo obviamente retorna que no existe
AdjListGraphTest	deleteEdgeTest()	setUpStage3();	("Pueblo Paleta", "Ciudad Verde")	Correcto, porque en la prueba se elimina la arista entre las entradas y al verificar la matriz de pesos es igual al infinito al no existir conexión no existe peso.
AdjListGraphTest	searchTest1()	setUpStage1	0	Correcto, porque la lista de adyacencia está vacía.
AdjListGraphTest	searchTest2()	setUpStage2()	("Pueblo Paleta", (0)) ("Ciudad Plateada") ("Hola")	Correcto, porque el primer vértice de la lista de adyacencia es "Pueblo Paleta" ya que es el primero agregado, "Ciudad

				Plateada" si se encuentra en el grafo y la última búsqueda es nula ya que no existe ("Hola")
AdjListGraphTest	weightMatrixTest()	setUpStage3()	("Pueblo Paleta", (0)) ("Ciudad Plateada") ("Hola")	Correcto porque los vértices agregados representan una posición en la matriz, y esa matriz tienen el peso específico de las aristas agregadas como también las posiciones restantes no tendrán un peso específico entonces será infinito.
AdjListGraphTest	getIndexVTest()	setUpStage2()	(0, "Pueblo Paleta") (1, "Ciudad Plateada")	Correcto, porque el primer vértice en el grafo es "Pueblo Paleta" entonces su índice será 0 como también el segundo vértice "Ciudad Plateada" será índice 1.
AdjListGraphTest	getVertexTest()	setUpStage2()	(3, alg.getVertex()) ("Ciudad Celeste") (4, alg.getVertex())	Correcto, porque en primera instancia el tamaño del grafo es igual a 3 vértices y después de agregar uno mas en el test completa la cantidad (4).

Escenarios:

Nombre	Clase	Método
setUpStage1	AdjMatrixGraphTest	new AdjListGraph<String>(false)
setUpStage2	AdjMatrixGraphTest	new AdjListGraph<String>(false) addVertex("Pueblo Paleta") addVertex("Ciudad Verde") addVertex("Ciudad Plateada")
setUpStage3	AdjMatrixGraphTest	new AdjListGraph<String>(false) addVertex("Pueblo Paleta") addVertex("Ciudad Verde") addVertex("Ciudad Plateada") alg.addEdge("Pueblo Paleta", "Ciudad Verde", 8) alg.addEdge("Ciudad Verde", "Ciudad Plateada", 15)

Pruebas:

Nombre	Método	Escenario	Entrada	Salida
AdjMatrixGraphTest	addVertexTest()	setUpStage1	"Pueblo Paleta"	Correcto, porque "Pueblo Paleta" es el primer vértice del grafo y el primer agregado.
AdjMatrixGraphTest	addEdgeTest1()	setUpStage2	("Pueblo Paleta", "Ciudad Verde") ("Ciudad Verde", "Ciudad Plateada")	Correcto, porque se las aristas que se agregan en la prueba conectan, "Pueblo Paleta" con "Ciudad Verde" y "Ciudad Verde" con "Ciudad Plateada"
AdjMatrixGraphTest	addEdgeTest2()	setUpStage2	("Pueblo Paleta", "Ciudad Verde")	Correcto, porque se la arista que se agrega en la prueba conecta "Pueblo Paleta" con

			("Ciudad Verde", "Ciudad Plateada")	"Ciudad Verde" y "Ciudad Verde", "Ciudad Plateada" no son adyacentes.
AdjMatrixGraphTest	deleteVertexTest()	setUpStage2();	("Pueblo Paleta")	
AdjMatrixGraphTest	deleteEdgeTest()	setUpStage3();	("Pueblo Paleta", "Ciudad Verde")	Correcto, porque en la prueba se elimina la arista entre las entradas y al verificar la matriz de pesos es igual al infinito al no existir conexión no existe peso.
AdjMatrixGraphTest	searchTest1()	setUpStage1	0	Correcto, porque la lista de adyacencia está vacía.
AdjMatrixGraphTest	searchTest2()	setUpStage2()	("Pueblo Paleta", (0)) ("Ciudad Plateada") ("Hola")	Correcto, porque el primer vértice de la lista de adyacencia es "Pueblo Paleta" ya que es el primero agregado, "Ciudad Plateada" si se encuentra en el grafo y la última búsqueda es nula ya que no existe ("Hola")
AdjMatrixGraphTest	weightMatrixTest()	setUpStage3()	("Pueblo Paleta", (0)) ("Ciudad Plateada") ("Hola")	Correcto porque los vértices agregados representan una posición en la matriz, y esa matriz tienen el peso específico de las aristas agregadas como también las posiciones restantes no tendrán un peso específico entonces será infinito.
AdjMatrixGraphTest	getIndexVTest()	setUpStage2()	(0, "Pueblo Paleta") (1, "Ciudad Plateada")	Correcto, porque el primer vértice en el grafo es "Pueblo Paleta" entonces su índice será 0 como también el segundo vértice "Ciudad Plateada" será índice 1.
AdjMatrixGraphTest	getVertexTest()	setUpStage2()	(3, alg.getVertex()) ("Ciudad Celeste") (4, alg.getVertex())	Correcto, porque en primera instancia el tamaño del grafo es igual a 3 vértices y después de agregar uno mas en el test completa la cantidad (4).

Escenarios:

Nombre	Clase	Método
setUpStage1	GraphAlgorithmsAdjMatrixTest	new AdjMatrixGraph<String>(false) ("Pueblo Paleta") ("Ciudad Verde") ("Ciudad Plateada") ("Ciudad Celeste") ("Pueblo Paleta", "Ciudad Verde", 8) ("Ciudad Verde", "Ciudad Plateada", 15) ("Pueblo Paleta", "Ciudad Celeste", 5) ("Ciudad Celeste", "Ciudad Plateada", 6)
setUpStage2	GraphAlgorithmsAdjMatrixTest	new AdjMatrixGraph<String>(false) ("A")

		("B") ("C") ("D") ("A", "B", 8) ("A", "C", 5) A", "D", 3) ("B", "D", 6) (C", "D", 1)
--	--	---

Pruebas:

Nombre	Método	Escenario	Entrada	Salida
GraphAlgorithmsAdjMatrixTest	bfsTest1()	setUpStage1	("Pueblo Paleta", b.get(0)) ("Ciudad Verde", b.get(1)) ("Ciudad Celeste", b.get(2)) ("Ciudad Plateada", b.get(3)) ("Ciudad Plateada", b.get(0)) ("Ciudad Verde", b.get(1)) ("Ciudad Celeste", b.get(2)) ("Pueblo Paleta", b.get(3));	Correcto, los recorridos resultantes desde el vértice origen pasado por parámetro al método son iguales a los esperados en la prueba
GraphAlgorithmsAdjMatrixTest	bfsTest2()	setUpStage2	("A", b.get(0)) ("B", b.get(1)) ("C", b.get(2)) ("D", b.get(3)) ("D", b.get(0)) ("A", b.get(1)) ("B", b.get(2)) ("C", b.get(3))	Correcto, los recorridos resultantes desde el vértice origen pasado por parámetro al método son iguales a los esperados en la prueba
GraphAlgorithmsAdjMatrixTest	dijkstraTest2()	setUpStage2	(0, cost[0]) (8, cost[1]) (4, cost[2]) (3, cost[3]) (3,GraphAlgorithms.getPath() [2]) (3, cost[0]) (6, cost[1]) (1, cost[2]) (0, cost[3]) (3,GraphAlgorithms.getPath() [2])	
GraphAlgorithmsAdjMatrixTest	primTest1()	setUpStage1	(19,GraphAlgorithms.prim("Pueblo Paleta",graph)) (13,GraphAlgorithms.prim("Ciudad Verde",graph)) (19,GraphAlgorithms.prim("Ciudad Celeste", graph)) (19,GraphAlgorithms.prim("Ciudad Plateada", graph))	
GraphAlgorithmsAdjMatrixTest	primTest2()	setUpStage2	(10, raphAlgorithms.prim("A", graph))(7,GraphAlgorithms.prim("B",graph)) (10, GraphAlgorithms.prim("C", graph))	

			(10, GraphAlgorithms.prim("D", graph))	
GraphAlgorithmsAdjMatrixTest	kruskalTest1()	setUpStage1	(e.getSource(),graph.search(graph.getIndexV("Pueblo Paleta"))) (e.getEnd(),graph.search(g raph.getIndexV("Ciudad Verde"))) (e.getWeight(), 0);	

Escenarios:

Nombre	Clase	Método
setUpStage1	GraphAlgorithmsAdjListTest	new AdjListGraph<String>(false) ("Pueblo Paleta") ("Ciudad Verde") ("Ciudad Plateada") ("Ciudad Celeste") ("Pueblo Paleta", "Ciudad Verde", 8) ("Ciudad Verde", "Ciudad Plateada", 15) ("Pueblo Paleta", "Ciudad Celeste", 5) ("Ciudad Celeste", "Ciudad Plateada", 6)
setUpStage2	GraphAlgorithmsAdjListTest	new AdjListGraph<String>(false) ("A") ("B") ("C") ("D") ("A", "B", 8) ("A", "C", 5) ("A", "D", 3) ("B", "D", 6) ("C", "D", 1)

Pruebas:

Nombre	Método	Escenario	Entrada	Salida
GraphAlgorithmsAdjListTest	bfsTest1()	setUpStage1	("Pueblo Paleta", b.get(0)) ("Ciudad Verde", b.get(1)) ("Ciudad Celeste", b.get(2)) ("Ciudad Plateada", b.get(3)) ("Ciudad Plateada", b.get(0)) ("Ciudad Verde", b.get(1)) ("Ciudad Celeste", b.get(2)) ("Pueblo Paleta", b.get(3));	Correcto, los recorridos resultantes desde el vértice origen pasado por parámetro al método son iguales a los esperados en la prueba
GraphAlgorithmsAdjListTest	bfsTest2()	setUpStage2	("A", b.get(0)) ("B", b.get(1)) ("C", b.get(2)) ("D", b.get(3)) ("D", b.get(0)) ("A", b.get(1)) ("B", b.get(2)) ("C", b.get(3))	Correcto, los recorridos resultantes desde el vértice origen pasado por parámetro al método son iguales a los esperados en la prueba
GraphAlgorithmsAdjListTest	dijkstraTest2()	setUpStage2	(0, cost[0]) (8, cost[1]) (4, cost[2])	

			(3, cost[3]) (3,GraphAlgorithms.getPath() [2]) (3, cost[0]) (6, cost[1]) (1, cost[2]) (0, cost[3]) (3,GraphAlgorithms.getPath() [2])	
GraphAlgorithmsAdjListTest	primTest1()	setUpStage1	(19,GraphAlgorithms.prim("P ueblo Paleta",graph)) (13,GraphAlgorithms.prim("Ci udad Verde",graph)) (19,GraphAlgorithms.prim("Ci udad Celeste", graph)) (19,GraphAlgorithms.prim("Ci udad Plateada", graph))	
GraphAlgorithmsAdjListTest	floydTest	setUpStage2	(8, m[1][0]) (8, m[0][1]) (4, m[2][0]) (4, m[0][2]) (3, m[3][0]) (3, m[0][3]) (8, m[1][0]) (8, m[0][1]) (4, m[2][0]) (4, m[0][2]) (3, m[3][0]) (3, m[0][3])	Correcto porque los pesos de la matriz resultante del floyWarshal son iguales a los esperados en la prueba
GraphAlgorithmsAdjListTest	primTest2()	setUpStage2	(10, raphAlgorithms.prim("A", graph))(7,GraphAlgorithms.pr im("B",graph)) (10, GraphAlgorithms.prim("C", graph)) (10, GraphAlgorithms.prim("D", graph))	
GraphAlgorithmsAdjListTest	kruskalTest1()	setUpStage1	(e.getSource(),graph.search(graph.getIndexV("Pueblo Paleta")))) (e.getEnd(),graph.search(gra ph.getIndexV("Ciudad Verde")))) (e.getWeight(), 0);	

