Método de la Ingeniería

Proyecto Final del curso

Algoritmos y Estructuras de Datos

Alejandro Narvaez

Jorge Morales

Kevin Zarama

Fase 1: Identificación del problema

Las personas que utilizan el medio de transporte “Mio” en Cali, quieren saber las maneras más eficientes y rápidas de llegar a sus destinos al tomar cualquiera de estos buses y en cualquier hora del día. A demás de esto, la empresa ha propuesto una mejora en registrar a las personas que usan este medio masivo para saber cuáles son los lugares más concurridos por las personas. Lo anterior se hace por medio de las nuevas tarjetas *EasyMio* que permiten amarrar al usuario con cada viaje que éste realiza, lo que permitirá informarle al viajero los lugares que ha visitado y la frecuencia con la que lo hace.

Problemática:

Las personas muchas veces llegan muy tarde a sus destinos debido a que desconocen las rutas del “Mio” y toman el bus equivocado. Además de esto, la empresa Mio quiere saber cuáles son los lugares más concurridos por las personas cercanos a la estaciones de este medio de transporte.Por otra parte, la empresa Mio esta muy dedicada a la seguridad y el bienestar de los usuarios, por lo cual, se ha decidido designar unas áreas específicas donde los usuarios que requieran atención especial tenga la posibilidad de bajar del transporte más rápido.

Identificación de necesidades y síntomas

Se necesita un sistema eficiente que permita mostrar la ruta más corta por la que una persona tenga que recorrer para llegar a su destino. Por otro lado, se necesita registrar el número de visitas a un lugar, además de que se requiere priorizar la salida de los usuarios dependiendo de su estado físico, si posee algún tipo de discapacidad.

Requerimientos funcionales

Nombre:	**R1 -** Encontrar la ruta del Mio más corta
Descripción:	Dado un punto inicial y un punto final, encontrar la ruta más corta
Entradas:	Punto inicial y punto final
Salidas:	La ruta más corta por la que tiene que pasar el usuario

Nombre:	**R2 -** Priorizar la salida de los usuarios
Descripción:	Según si el usuario posee algún tipo de discapacidad evaluará el Mio de primero, de lo contrario esperará hasta que salgan todos los usuarios que requieran asistencia.
Entradas:	Cola de prioridad de usuarios
Salidas:	La cola de prioridad pero sin los usuarios que se bajaron en la estación.

Fase 2. Recopilación de Información

- **Grafo**

Un 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. Son objeto de estudio de la teoría de grafos.

Típicamente, un grafo se representa gráficamente como un conjunto de puntos (vértices o nodos) unidos por líneas (aristas).

Desde un punto de vista práctico, los grafos permiten estudiar las interrelaciones entre unidades que interactúan unas con otras. Por ejemplo, una red de computadoras puede representarse y estudiarse mediante un grafo, en el cual los vértices representan terminales y las aristas representan conexiones.

Prácticamente cualquier problema puede representarse mediante un grafo, y su estudio trasciende a las diversas áreas de las ciencias exactas y las ciencias sociales.

- **Pila**

Una pila (*stack* en inglés) es una lista ordenada o estructura de datos que permite almacenar y recuperar datos, el modo de acceso a sus elementos es de tipo LIFO (del inglés *Last In, First Out*, «último en entrar, primero en salir») . Esta estructura se aplica en multitud de supuestos en el área de informática debido a su simplicidad y capacidad de dar respuesta a numerosos procesos. Para el manejo de los datos cuenta con dos operaciones básicas: apilar (*push)*, que coloca un objeto en la pila, y su operación inversa, retirar(o desapilar, *pop*), que retira el último elemento apilado. En cada momento sólo se tiene acceso a la parte superior de la pila, es decir, al último objeto apilado (denominado TOS, *Top of Stack* en inglés). La operación retirar permite la obtención de este elemento, que es retirado de la pila permitiendo el acceso al anterior (apilado con anterioridad), que pasa a ser el último, el nuevo TOS.

- **Cola**

Una cola (también llamada fila) es una estructura de datos, caracterizada por ser una secuencia de elementos en la que la operación de inserción *push* se realiza por un extremo y la operación de extracción *pop* por el otro. También se le llama estructura FIFO (del inglés *First In First Out*), debido a que el primer elemento en entrar será también el primero en salir.

Las colas se utilizan en sistemas informáticos, transportes y operaciones de investigación (entre otros), donde los objetos, personas o eventos son tomados como datos que se almacenan y se guardan mediante colas para su posterior procesamiento. Este tipo de estructura de datos abstracta se implementa en lenguajes orientados a objetos mediante clases, en forma de listas enlazadas.

- **Cola de Prioridad:**

Una cola de prioridades es un tipo de dato abstracto similar a una cola en la que los elementos tienen adicionalmente, una *prioridad* asignada. En una cola de prioridades un elemento con mayor prioridad será desencolado antes que un elemento de menor prioridad. Si dos elementos tienen la misma prioridad, se desencolarán siguiendo el orden de cola.

- **HashMap**

Una tabla hash, matriz asociativa, hashing, mapa hash, tabla de dispersión o tabla fragmentada es una estructura de datos que asocia *llaves* o *claves* con *valores*. La operación principal que soporta de manera eficiente es la *búsqueda*: permite el acceso a los elementos (teléfono y dirección, por ejemplo) almacenados a partir de una clave generada (usando el nombre o número de cuenta, por ejemplo). Funciona transformando la clave con una función hash en un *hash*, un número que identifica la posición (*casilla* o *cubeta*) donde la tabla hash localiza el valor deseado.

- **Matriz de Adyacencia**

La matriz de adyacencia es una matriz cuadrada que se utiliza como una forma de representar relaciones binarias.

- **Lista de Adyacencia:**

En teoría de grafos, 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 de dos nodos, uno denotando el nodo fuente y el otro denotando el nodo destino del arco correspondiente.

Fuentes:

https://es.wikipedia.org/wiki/Cola_(inform%C3%A1tica)

https://es.wikipedia.org/wiki/Pila_(inform%C3%A1tica)

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

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

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

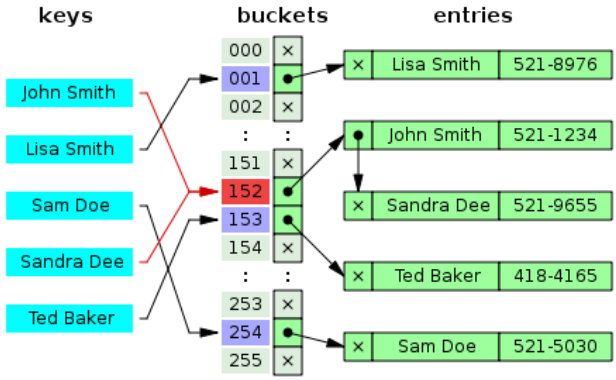
https://es.wikipedia.org/wiki/Lista_de_adyacencia

Fase 3. Búsqueda de soluciones creativas

Con respecto a la selección de soluciones, entre los integrantes del grupo se decidió hacer una lluvia de ideas entre los diferentes tipos de estructuras de datos más relevantes. Por este medio se pudo plantear una idea de cómo es posible aplicar estas estructuras para solucionar uno de los requerimientos que presenta el problema de transporte EasyMio. En definitiva se escogieron varios diferentes tipos de estructuras que pueden ser aplicados para solucionar de diversas maneras los puntos del proyecto.

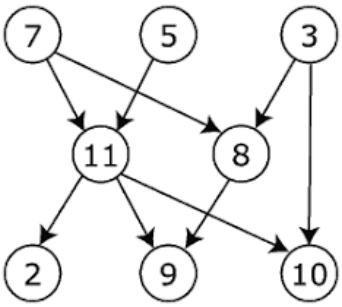
Alternativa 1.HashTable:

La estructura de HashTable es una muy buena opción para manejar datos, ya que permite realizar una buena distribución de los elementos en la tabla hash y permite almacenar grandes cantidades de datos. En este caso para el sistema de transporte EasyMio va a ser de gran importancia porque nos permitirá acceder a los elementos con facilidad.



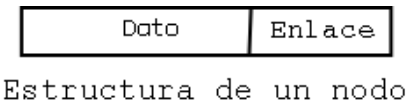
Alternativa 2. Grafo:

La manera mas practica para atacar la problemática principal del proyecto, es hacer uso de una estructura de grafos, debido a que permite modelar cada aspecto que necesita el proyecto para funcionar. Permite plantear las estaciones del Mio como nodos y las aristas del grafo como rutas.

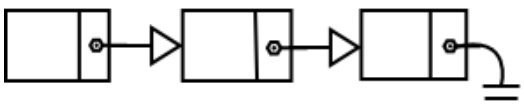


Alternativa 3. Lista:

La idea de modelar la mayoría de programas de forma básica es mediante el uso de listas enlazadas debido a que son simples y cómodas de utilizar, es una estructura muy parecida a un arreglo.



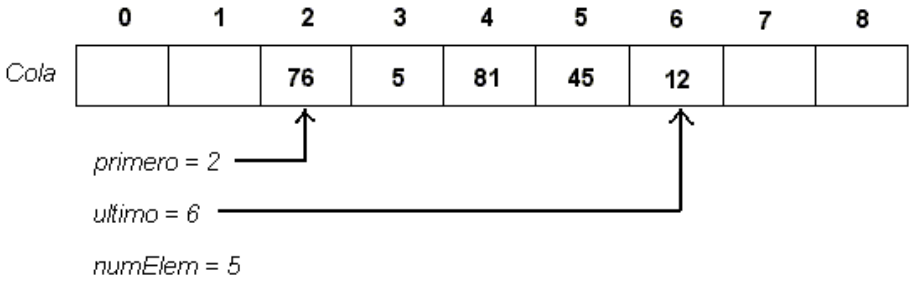
Estructura de un nodo



Lista enlazada

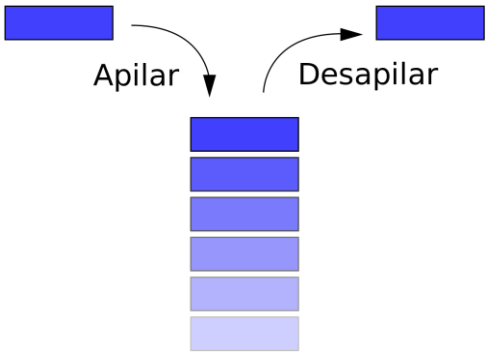
Alternativa 4. PriorityQueue:

La cola de prioridad es una estructura que me permite mantener determinado orden a la hora de manejar los objetos, además de que es un requisito necesario para poder construir un grafo. Por otro lado esta estructura nos sirve para manejar cierto tipo de particularidades que aparecen en el programa.



Alternativa 5. Pila:

Este tipo de estructura nos permite apilar los datos, para este caso esta opción puede ser factible usar debido a que puede ayudar a almacenar los usuarios del Mio, a medida que entren si van apilando, y la única forma de salir es que los últimos que entraron salgan.



Fase 4. Transición de las Ideas a diseños preliminares

La revisión cuidadosa de las alternativas nos conduce a lo siguiente:

Ideas descartadas:
Alternativa 5. Pila:

Esta estructura es bien ineficiente si se desea plantear como método para que los usuarios ingresen al Mio. Debido en el caso donde los primeros usuarios que ingresaron se desean bajar, tocaría desapilar a todos y volver a apilar.

Alternativa 3. Lista:

Esta estructura es muy simple y no necesariamente nos cumplira la mayoria de requerimientos necesarios para que el programa funcione correctamente. Existen

estructuras más complejas como la cola que abarcan y se adaptan mejor para la solución de la problemática.

Ideas aceptadas:

Alternativa 1.HashTable:

La Tabla Hash es una estructura muy útil eficiente que nos permitirá acceder a los elementos de forma rápida, además de que es un requisito para el funcionamiento de un grafo.

Alternativa 2. Grafo:

El grafo es la estructura que más se acopla a la solución del problema hablando en general. Ya que , los grafos nos permiten modelar mapas o programas que identifiquen rutas cortas, siendo de esta forma la estructura perfecta para la problemática de Easy Mio. Por otra parte, esta estructura es eficiente con el manejo de objetos.

Alternativa 4. PriorityQueue:

La cola de prioridad o MinHeap es una estructura que nos permitirá encolar a los usuarios de cierta forma en que se asigne una condición para su encolamiento, en este caso sirve para encolar y dar prioridad a aquellos usuarios que posean una discapacidad. Por otra parte la cola de prioridad es un prerrequisito para el funcionamiento de un grafo.

Fase 5. Evaluación y Selección de la Mejor Solución

Requerimiento 1:

Criterio: elegir la mejor estructura que nos permita encontrar el camino más corto a la siguiente estación para que él mío tome esa ruta y así cumplir con él requerimiento 1.

La única estructura que nos puede ayudar en este requerimiento son los grafos debido a que está estructura es la única que nos permite cumplir con él requerimiento 1.

Requerimiento 2:

Criterio 1: la estructura que nos permite añadir a una persona con prioridad según su estado.

- [4] velocidad alta para añadir un elemento O(1)
- [3] velocidad media para añadir un elemento O(n)
- [2] velocidad baja para añadir un elemento O(nLogn)
- [1] velocidad alta para añadir un elemento O(n^2)

Criterio 2: la estructura que permita retornar un elemento con prioridad

- [4] velocidad alta para añadir un elemento O(1)
- [3] velocidad media para añadir un elemento O(n)
- [2] velocidad baja para añadir un elemento O(nLogn)
- [1] velocidad alta para añadir un elemento O(n^2)

Requerimiento 1			
	Criterio A	Criterio B	Total
Tabla Hash	3	3	3
Priority Queue	2	4	6

la mejor estructura y la que vamos a utilizar es la priority queue debido a que es la estructura que nos permite ordenar a los usuarios según la prioridad de su estado.

Fase 6. Preparación de informes y especificaciones

Especificación del problema

Problema: Hay que administrar las rutas más óptimas para el transporte de los pasajeros, además de la eficiencia que tiene que existir a la hora de bajar los del transporte.
Entradas: la cantidad de usuarios, la cantidad de estaciones y la cantidad de buses en servicio.
Salidas: Determinar cuál es la estación que más se frecuenta

Consideraciones:
Se deben de tener en cuenta los siguientes casos para buscar recorrer la ciudad y desencolar a los usuarios.

Pseudocódigo:

ALGORITMO DE DIJKSTRA

```
proc Dijkstra(G: grafo ponderado simple y conexo, con pesos positivos,
              a: vértice inicial del camino mínimo que se desea hallar)
{G tiene pesos w(v_i,v_j), donde w(v_i,v_j)=∞ si {v_i,v_j} no es una arista de G}
for i:=1 to n       for i:=1 to n
    L(v_i):=∞           C(v_i):={}
L(a):=0             C(a):= a
S:=Ø
{los valores iniciales de las etiquetas se asignan de modo que la etiqueta
 de a es 0 y todas las demás etiquetas son ∞ y S es el conjunto vacío}
while S ≠ V
begin
    u := vértice con L(u) mínima entre los vértices que no están en S
    S := S U {u}
    for todos los vértices v que no están en S
        if L(u)+w(u,v)<L(v) then { L(v):=L(u)+w(u,v); C(v):= C(u),v }
    {esto añade a S un vértice con etiqueta mínima, actualiza las
     etiquetas (L) de los vértices que no están en S y el arreglo de
     caminos C de manera que tenga los caminos mínimos hasta este momento}
end {L =arreglo de longitudes de caminos mínimos entre a y todos los demás,
     C =arreglo de caminos mínimos entre a y todos los demás}
```

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

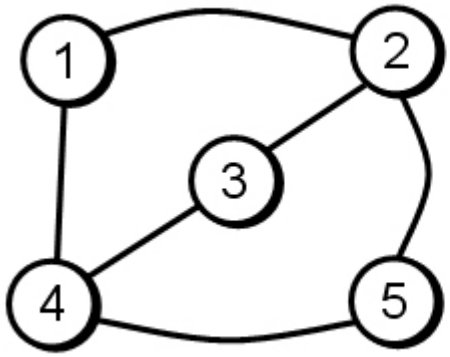
```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
 1  time = time + 1            // white vertex u has just been discovered
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]      // explore edge (u, v)
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK            // blacken u; it is finished
 9  time = time + 1
10  u.f = time
```

```
FLOYD-WARSHALL(W)
1. n ← rows[W]
2. D^(0) ← W
3. for k ← 1 to n
4.     do for i ← 1 to n
5.         do for j ← 1 to n
6.             d_ij^(k) ← min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
7. return D^(n)
```

Tipo Abstracto de Datos (TAD)

TAD Grafo Matriz
Representación:



M	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	0	1
3	0	1	0	1	0
4	1	0	1	0	1
5	0	1	0	1	0

Invariantes:

- n es el número de nodos que tiene el grafo.

Operaciones:

addEdge	Vertex<V>,Vertex<V>,E	--------->void
addVertex	Vertex<V>	--------->void
removeVertex	Vertex<V>	--------->void
removeEdge	Edge<Vertex<V>,E>	------->void
BFS	Vertex<V>	------->void
DFS	Vertex<V>	------->void
Dijsktra	Vertex<V>	------->void
FloydWarshall	Vertex<V>	------->void
Prim	Vertex<V>	------->void
Kruskal	Vertex<V>	------->void
Isdirected		--------->boolean

addEdge(Vertex<V> v1,Vertex<V> v2,E edge)

pre:
v1!=null
v2!=null
edge!=null

post:
Este método se encarga de agregar correctamente una arista en la matriz de adyacencia.

addVertex(Vertex<V> v)

pre:
v!=null

post: Este método se encarga de agregar un vértice en el hash clave valor.

removeVertex(Vertex<V> v)
pre: v!=null
post: Método elimina el nodo v del grafo en una matriz de adyacencia y del HashTable

removeEdge(Edge<Vertex<V>,E> v)
pre: v!=null
post: método que elimina la arista de la matriz de adyacencia de aristas

BFS(Vertex<V> v)
pre: v!=null;
post: Metodo se encarga de buscar el nivel de profundidad de cada nodo del grafo

DFS(Vertex<V> v)
pre: v!=null;
post: Metodo se encarga de buscar el nivel de profundidad de cada nodo del grafo de diferente estrategia de búsqueda del BFS

Dijsktra(Vertex<V> v)
pre:

v!=null
post: método que se encarga de buscar todas las distancias más cortas desde un nodo fuente hacia todos los demás.

FloydWarshall(Vertex<V> v)
pre: v!=null
post: método que se encarga de buscar todas las distancias más cortas de todos los nodos con todos los nodos.

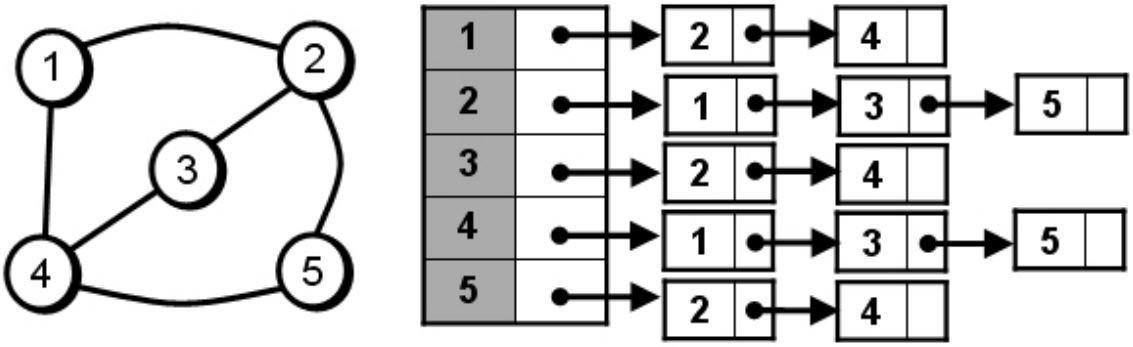
Prim(Vertex<V> v)
pre: v!=null
post: método que se encarga de buscar todas las distancias mínimas del grafo para formar el árbol de recubrimiento mínimo. (Solo varía la estrategia de hallar las aristas mínimas con respecto al kruskal)

Kruskal(Vertex<V> v)
pre: v!=null
post: método que se encarga de buscar todas las distancias mínimas del grafo para formar el árbol de recubrimiento mínimo.. (Solo varía la estrategia de hallar las aristas mínimas con respecto al prim)

IsDirected()
pre:
post: método que se encarga de retornar true si el grafo es dirigido, false en caso contrario

TAD Grafo Lista Adyacencia

Representación:



Invariantes:

- n es el número de vértices

Operaciones:

addEdge	Vertex<V>,Vertex<V>,E	--------->void
addVertex	Vertex<V>	--------->void
removeVertex	Vertex<V>	--------->void
removeEdge	Edge<Vertex<V>,E>	------->void
BFS	Vertex<V>	------->void
Dijsktra	Vertex<V>	------->void
DFS	Vertex<V>	------->void
Prim	Vertex<V>	------->void
Kruskal	Vertex<V>	------->void
Isdirected		--------->boolean

addEdge(Vertex<V> v1,Vertex<V> v2,E edge)

pre:

v1!=null
v2!=null
edge!=null

post:

Este método se encarga de agregar correctamente una arista en la lista de

incidencia.

addVertex(Vertex<V> v)
pre: v!=null
post: Este método se encarga de agregar un vértice en el hash clave valor.

removeVertex(Vertex<V> v)
pre: v!=null
post: Método elimina el nodo v del grafo en una lista de incidencia y del HashTable

removeEdge(Edge<Vertex<V>,E> v)
pre: v!=null
post: método que elimina la arista de la lista de incidencia de aristas

BFS(Vertex<V> v)
pre: v!=null;
post: Metodo se encarga de buscar el nivel de profundidad de cada nodo del grafo

DFS(Vertex<V> v)
pre: v!=null;
post: Metodo se encarga de buscar el nivel de profundidad de cada nodo del grafo de

diferente estrategia de búsqueda del BFS

Dijsktra(Vertex<V> v)
pre: v!=null
post: método que se encarga de buscar todas las distancias más cortas desde un nodo fuente hacia todos los demás.

Prim(Vertex<V> v)
pre: v!=null
post: método que se encarga de buscar todas las distancias mínimas del grafo para formar el árbol de recubrimiento mínimo. (Solo varía la estrategia de hallar las aristas mínimas con respecto al kruskal)

Kruskal(Vertex<V> v)
pre: v!=null
post: método que se encarga de buscar todas las distancias mínimas del grafo para formar el árbol de recubrimiento mínimo.. (Solo varía la estrategia de hallar las aristas mínimas con respecto al prim)

IsDirected()
pre:
post: método que se encarga de retornar true si el grafo es dirigido, false en caso contrario

DIAGRAMA DE OBJETOS:

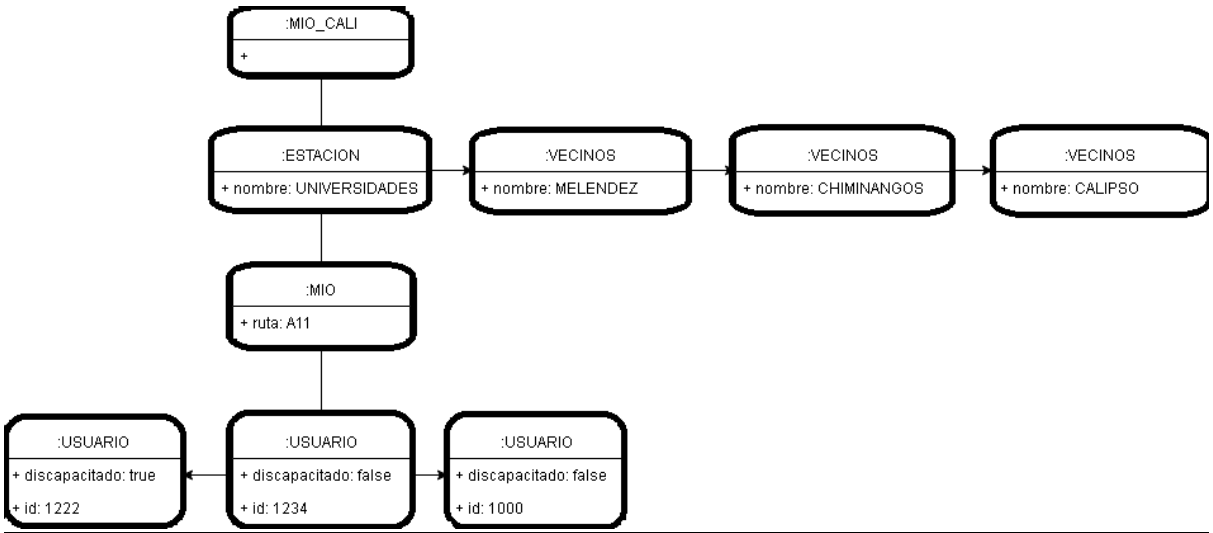


DIAGRAMA DE CLASES:

DISEÑO DE CASOS DE PRUEBA:

OBJETIVO: probar que el método addEdge funciona correctamente				
CLASE: GraphMatix		METODO: addEdge		
CASO:	DESCRIPCION DE LA PRUEBA:	ESCENARIO:	VALORES DE ENTRADA:	RESULTADOS
1	Probar que la matriz comienza vacía y lista	Stage1		true

2	Probar que se añade un Edge correctamente	Stage2	Vertex<String> vertex1 = new Vertex<String>("A"); Vertex<String> vertex2 = new Vertex<String>("B"); Edge<String, Integer> edge = new Edge<String, Integer>("C", 50);	true

OBJETIVO: probar que el método addVertex funciona correctamente				
CLASE: GraphMatix		METODO: addVertex		
CASO:	DESCRIPCION DE LA PRUEBA:	ESCENARIO:	VALORES DE ENTRADA:	RESULTADOS
1	probar que la matriz inicia vacia	stage3		true
2	Probar que se añade un vertex	Stage4	Vertex<String> vertex1 = new Vertex<String>("A");	true

OBJETIVO: probar que el método removeEdge elimina de manera correcta un edge				
CLASE: GraphMatrix		METODO: removeEdge		
CASO:	DESCRIPCION DE LA PRUEBA:	ESCENARIO:	VALORES DE ENTRADA:	RESULTADOS

1	Probar que se elimina de manera correcta un edge	Stage5	“A”	true

OBJETIVO: probar que el método removeVertex elimina de manera correcta un vertex				
CLASE: GraphMatrix		METODO: removeVertex		
CASO:	DESCRIPCION DE LA PRUEBA:	ESCENARIO:	VALORES DE ENTRADA:	RESULTADOS
1	Probar que se elimina de manera correcta un vertex	Stage6	“A”	true

OBJETIVO: probar que el método BFS da un recorrido de la forma correcta al grafo				
CLASE: GraphMatrix		METODO: BFS		
CASO:	DESCRIPCION DE LA PRUEBA:	ESCENARIO:	VALORES DE ENTRADA:	RESULTADOS
1	Probar que se recorre de manera correcta al grafo	Stage7		true

OBJETIVO: probar que el método DFS da un recorrido de la forma correcta al grafo				
CLASE: GraphMatrix		METODO: DFS		
CASO:	DESCRIPCION DE LA PRUEBA:	ESCENARIO:	VALORES DE ENTRADA:	RESULTADOS

1	Probar que se recorre de manera correcta al grafo	Stage8		true

OBJETIVO: probar que el método dijkstra funciona				
CLASE: GraphMatrix		METODO: dijkstra		
CASO:	DESCRIPCION DE LA PRUEBA:	ESCENARIO:	VALORES DE ENTRADA:	RESULTADOS
1	Probar que se retorna el recorrido más corto desde un nodo inicial a otro nodo del grafo	Stage9	“A”	true

OBJETIVO: probar que el método retorna la distancia mínima de cada nodo a cualquier otro nodo				
CLASE: GraphMatrix		METODO: floydWarshallAlgorithm		
CASO:	DESCRIPCION DE LA PRUEBA:	ESCENARIO:	VALORES DE ENTRADA:	RESULTADOS
1	probar que el método retorna la distancia mínima de cada nodo a cualquier otro nodo de manera correcta	Stage10		true

OBJETIVO: probar que el método prim funciona				
CLASE: GraphMatrix		METODO: prim		
CASO:	DESCRIPCION DE LA PRUEBA:	ESCENARIO:	VALORES DE ENTRADA:	RESULTADOS

1	Probar que él metodo retorna un arbol de minimo recobrimiento	Stage9	“A”	true

OBJETIVO: probar que el método kruskal funciona				
CLASE: GraphMatrix		METODO: kruskal		
CASO:	DESCRIPCION DE LA PRUEBA:	ESCENARIO:	VALORES DE ENTRADA:	RESULTADOS
1	Probar que él metodo retorna un arbol de minimo recobrimiento	Stage9	“A”	true

OBJETIVO: probar que el método addVertex funciona correctamente				
CLASE: GraphList		METODO: addVertex		
CASO:	DESCRIPCION DE LA PRUEBA:	ESCENARIO:	VALORES DE ENTRADA:	RESULTADOS
1	Probar que no hay vertex al inicio de la ejecucion	Stage1		true
2	Probar que se añade un vertex de forma correcta	Stage2	Vertex<String> vertex = new Vertex<String>("A");	true

OBJETIVO: probar que el método addEdge funciona correctamente	
CLASE: GraphList	METODO: addEdge

CASO:	DESCRIPCION DE LA PRUEBA:	ESCENARIO:	VALORES DE ENTRADA:	RESULTADOS
1	Probar que no hay edges al inicio de la ejecucion	Stage3		true

2	Probar que se añade un edge de forma correcta	Stage4	Vertex<String> vertex1 = new Vertex<String>("A");
			
			graphList.addVertex(vertex1);
			
			Vertex<String> vertex2 = new Vertex<String>("B");
			
			graphList.addVertex(vertex2);
			
			Edge<Vertex<String>, Integer> edge = new Edge<Vertex<String>, Integer>(vertex2, 31);
				true

OBJETIVO: probar que el método remove Edge elimina un edge				
CLASE: GraphList		METODO:removeEdge		
CASO:	DESCRIPCION DE LA PRUEBA:	ESCENARIO:	VALORES DE ENTRADA:	RESULTADOS
1	Probar que se elimina un edge correctamente	Stage5		true

OBJETIVO: probar que el método removeVertex funciona correctamente

CLASE: GraphList		METODO: removeVertex		
CASO:	DESCRIPCION DE LA PRUEBA:	ESCENARIO:	VALORES DE ENTRADA:	RESULTADOS
1	Probar que se elimina un vertex	Stage6		true

OBJETIVO: probar que el método BFS da un recorrido de la forma correcta al grafo				
CLASE: GraphList		METODO: BFS		
CASO:	DESCRIPCION DE LA PRUEBA:	ESCENARIO:	VALORES DE ENTRADA:	RESULTADOS
1	Probar que se recorre de manera correcta al grafo	Stage7		true

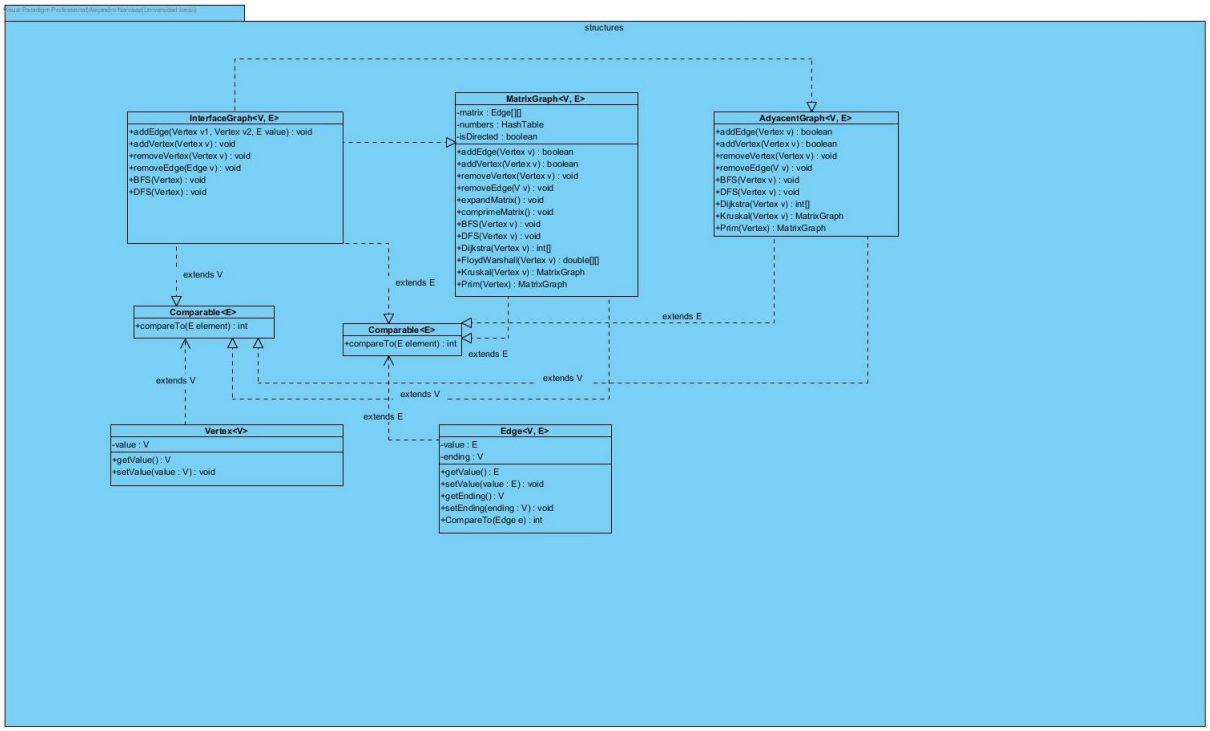
OBJETIVO: probar que el método DFS da un recorrido de la forma correcta al grafo				
CLASE: GraphList		METODO: DFS		
CASO:	DESCRIPCION DE LA PRUEBA:	ESCENARIO:	VALORES DE ENTRADA:	RESULTADOS
1	Probar que se recorre de manera correcta al grafo	Stage8		true

OBJETIVO: probar que el método dijkstra funciona	
CLASE: GraphList	METODO: dijkstra

CASO:	DESCRIPCION DE LA PRUEBA:	ESCENARIO:	VALORES DE ENTRADA:	RESULTADOS
1	Probar que se retorna el recorrido más corto desde un nodo inicial a otro nodo del grafo	Stage9	“A”	true

OBJETIVO: probar que el método prim funciona				
CLASE: GraphList		METODO: prim		
CASO:	DESCRIPCION DE LA PRUEBA:	ESCENARIO:	VALORES DE ENTRADA:	RESULTADOS
1	Probar que él metodo retorna un arbol de minimo recobrimiento	Stage9	“A”	true

OBJETIVO: probar que el método kruskal funciona				
CLASE: GraphList		METODO: kruskal		
CASO:	DESCRIPCION DE LA PRUEBA:	ESCENARIO:	VALORES DE ENTRADA:	RESULTADOS
1	Probar que él metodo retorna un arbol de minimo recobrimiento	Stage9	“A”	true


structures
InterfaceGraph<V, E>
+addEdge(Vertex v1, Vertex v2, E value) : void
+addVertex(Vertex v) : void
+removeVertex(Vertex v) : void
+removeEdge(Edge v) : void
+BFS(Vertex) : void
+DFS(Vertex) : void
MatrixGraph<V, E>
-matrix : Edge[][]
-numbers : HashTable
-isDirected : boolean
+addEdge(Vertex v) : boolean
+addVertex(Vertex v) : boolean
+removeVertex(Vertex v) : void
+removeEdge(V v) : void
+expandMatrix() : void
+comprimeMatrix() : void
+BFS(Vertex v) : void
+DFS(Vertex v) : void
+Dijkstra(Vertex v) : int[]
+FloydWarshall(Vertex v) : double[][]
+Kruskal(Vertex v) : MatrixGraph
+Prim(Vertex) : MatrixGraph
AdyacentGraph<V, E>
+addEdge(Vertex v) : boolean
+addVertex(Vertex v) : boolean
+removeVertex(Vertex v) : void
+removeEdge(V v) : void
+BFS(Vertex v) : void
+DFS(Vertex v) : void
+Dijkstra(Vertex v) : int[]
+Kruskal(Vertex v) : MatrixGraph
+Prim(Vertex) : MatrixGraph
extends V
extends E
Comparable<E>
+compareTo(E element) : int
Comparable<E>
+compareTo(E element) : int
extends E
extends E
extends V
extends V
extends V
extends E
Vertex<V>
-value : V
+getValue() : V
+setValue(value : V) : void
Edge<V, E>
-value : E
-ending : V
+getValue() : E
+setValue(value : E) : void
+getEnding() : V
+setEnding(ending : V) : void
+CompareTo(Edge e) : int