**Trabajo Práctico Especial**

**Programación III**

**Integrante: Mateo Albert**

**E-Mail:** [**mateo.albert45@gmail.com**](mailto:mateo.albert45@gmail.com)

**Introducción al problema**

Luego de leer el enunciado identifique que el problema se basa en crear un aeropuerto utilizando la estructura de grafo donde cada nodo sería un aeropuerto y cada arista una ruta, lo primero que decidí fue realizar un esquema de la estructura del problema, de esta manera, identifique las distintas clases y atributos que voy a utilizar.

**Modelado del Problema**

En base al esquema realizado para identificar la estructura de los datos decidí crear las siguientes clases, con sus respectivos atributos, de la siguiente forma:

**Datos identificas y relaciones**

* Clase Sistema: Su función es almacenar los distintos aeropuertos y reseras, dentro de una misma clase.
* Clase Aeropuerto: Funciona como el nodo en un grafo, se relaciona con otros Aeropuertos a través de la clase Ruta.
* Clase Ruta: Permite conectar uno o más aeropuertos entre sí.
* Clase Aerolínea: Sirve para poder almacenar más de una aerolínea en la ruta.
* Clase Reserva: Se utiliza para saber en una determinada ruta la cantidad de asientos reservados que hay para una aerolínea particular.
* Clase Camino: Fue creada para facilitar la impresión y almacenamiento de información para el servicio 2

**Estructura de datos**

* Clase Sistema:
* Aeropuertos: Vector <Aeropuerto>
* Reservas: Vector <Reserva>
* Clase Aeropuerto:
* Nombre: String
* País: String
* Ciudad: String
* Color: String
* Estado: String
* Rutas: Vector<Ruta>
* Clase Ruta:
* Origen: Aeropuerto
* Destino: Aeropuerto
* Kilómetros: int
* Cabotaje: boolean
* Aerolíneas: Vector<Aerolínea>
* Clase Aerolínea:
* Nombre: String
* Pasajes: int

* Clase Reserva:
* Origen: Aeropuerto
* Destino: Aeropuerto
* Aerolinea: String
* Asientos\_Reservados: int
* Clase Camino:
* Rutas: Vector<Ruta>
* Kilómetros: double
* Escalas: int

**Ventajas y desventajas Estructura**

Utilizar grafo con lista de adyacencia:

Decidí utilizar una lista de adyacencia debido a que no ocupa tanta memoria como la matriz ya que el espacio que utiliza es proporcional a la cantidad de nodos que tiene, pero la desventaja es que tiene un tiempo de ejecución de O(n).

Vector:

El crear un vector tiene la ventaja de despreocuparme del tamaño del arreglo al momento de insertar o eliminar los elementos, además que en este caso es necesario debido a que no conocemos cual va a ser la cantidad de elementos que debemos insertar en el mismo. Por el lado de las desventajas, una de ellas, es que cada vez que inserto un elemento debe copiar el arreglo actual a un arreglo mayor tamaño esto sucede en un tiempo O(n) y en algunos casos puede ser innecesario.

Crear la clase Aerolínea:

Debido a que cada ruta tiene un conjunto de aerolíneas con sus respectivos pasajes, decidí crear la clase aerolínea para así no tener que crear muchas instancias de la clase ruta con casi la misma información, de esta manera, la estructura es más fácil de comprender y de utilizar

Crear la clase Camino:

Debido a la necesidad de ir guardando la información para el servicio 2 decidí crear una nueva clase llamada Camino donde cada ruta que haya para llegar de un aeropuerto a otro quedara en esta clase junto con los kilómetros totales y la cantidad de escalas.

**Implementación de servicio 1**

Para realizar el primer servicio del aeropuerto realice cuatro métodos:

Vuelo\_directo: Mediante este método se pide que ingresen los aeropuertos origen y destino y la aerolínea con la que se desea viajar, así, en este método se recorre todas las rutas del aeropuerto origen, en caso de que la ruta conecte con el aeropuerto destino se calcula la cantidad de pasajes, y si hay pasajes disponibles se calcula la cantidad de estos.

Cant\_pasajes: Debido a que cada ruta tiene un vector de aerolíneas decidí recorrer este vector y en caso de que el nombre de la aerolínea pasado por parámetro sea igual al de la aerolínea que está comparando retorna la cantidad de pasajes que tiene para esta ruta.

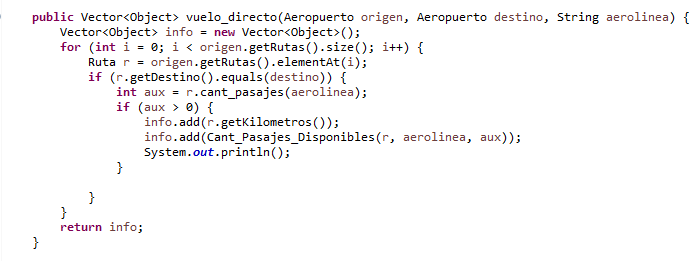
Cant\_Pasajes\_Disponibles: Este método mediante la ruta, aerolínea y cantidad de pasajes que son pasados como parámetro permite recorrer las reservas que tiene cargado el sistema para que en caso de que estas se apliquen a la ruta retorne la cantidad de pasajes que hay disponibles.

Es\_igual: Funciona como un “equals” entre una reserva y una ruta, para poder identificar a que vuelo hace referencia la reserva.

calcular\_pasajes\_disponibles: permite calcular la cantidad de pasajes que quedan disponibles.

Este algoritmo tiene una complejidad o(n)^2

**Seguimiento del algoritmo Vuelo\_directo:**



public void vuelo\_directo(Aeropuerto origen, Aeropuerto destino, String aerolinea) {

for (int i = 0; i < origen.getRutas().size(); i++) {

Ruta r = origen.getRutas().elementAt(i);

Se recorren las rutas del aeropuerto origen pasado como parámetro y se guardan en la variable r de tipo Ruta

--

if (r.getDestino().equals(destino)) {

Chequea que el destino de la ruta seleccionada sea igual al aeropuerto destino pasado como parámetro

--

int aux = r.cant\_pasajes(aerolinea);

en caso de que se encuentre una ruta con ese destino se calcula la cantidad de pasajes que tiene esa ruta con su determinada aerolínea

--

if (aux > 0) {

info.add(r.getKilometros());

info.add(Cant\_Pasajes\_Disponibles(r, aerolinea, aux));

Si la variable aux es mayor a uno, quiere decir que habrá al menos un pasaje disponible

**Implementación de servicio 2**

Para realizar el primer servicio del aeropuerto realice los siguientes métodos:

Vuelos\_Disponibles: este algoritmo sirve para poder setear todos los nodos del grafo en el estado “Sin visitar” y utilizar el método recursivo Buscar\_Vuelos.

Buscar\_Vuelos: Mediante este algoritmo se pueden ir recorriendo cada uno de los aeropuertos hasta llegar al aeropuerto destino y se calcula la cantidad de kilómetros que tiene el camino completo.

contiene\_aerolinea\_distinta\_a: verifica si la ruta contiene alguna aerolínea que no sea la seleccionada por el usuario.

**Seguimiento del algoritmo Buscar\_Vuelos:**



for (int i = 0; i < actual.getRutas().size(); i++) {

Ruta temporal = actual.getRutas().elementAt(i);

if (temporal.contiene\_aerolinea\_distinta\_a(aerolinea)) {

Aeropuerto aux = temporal.getDestino();

Se recorre las distintas rutas del aeropuerto actual y se busca la que no contenga la aerolínea que paso el usuario como parámetro, en caso de encontrarse se guarda en una variable.

---

vec.add(actual);

Rutas.add(temporal);

cantidad\_kilometros += temporal.getKilometros();

Se agrega el aeropuerto a un vector para ir construyendo el recorrido del camino, lo mismo sucede con las rutas, además se va contando la cantidad de kilómetros

--

if (aux.equals(destino)) {

vec.add(aux);

Camino camino = new Camino(Rutas, Rutas.size(), cantidad\_kilometros);

caminos.add(camino);

imprimir\_info\_camino(camino);

vec.remove(aux);

}

Cheque si el nodo siguiente al actual es el aeropuerto destino, y si es así, se agrega a si mismo al vec de recorrido, crea un nuevo camino y le pasa como parámetro los atributos necesarios, además se remueve del vector.

--

else if (aux.getEstado().equals("Sin Visitar")) {

aux.setEstado("Visitado");

caminos.addAll(Buscar\_Vuelos(aux, destino, vec, aerolinea, cantidad\_kilometros, Rutas, caminos));

aux.setEstado("Sin Visitar");

}

en caso de no ser el aeropuerto destino explora el siguiente nodo en caso de que no haya sido visitado anteriormente, y se vuelve a llamar a si mismo con otros parámetros.

--

cantidad\_kilometros -= temporal.getKilometros();

vec.remove(actual);

Rutas.remove(temporal);

Se restan los kilómetros de los caminos que son sacados del vector, al igual que se retiran de los vectores las rutas y aeropuertos que ya no son necesarios.

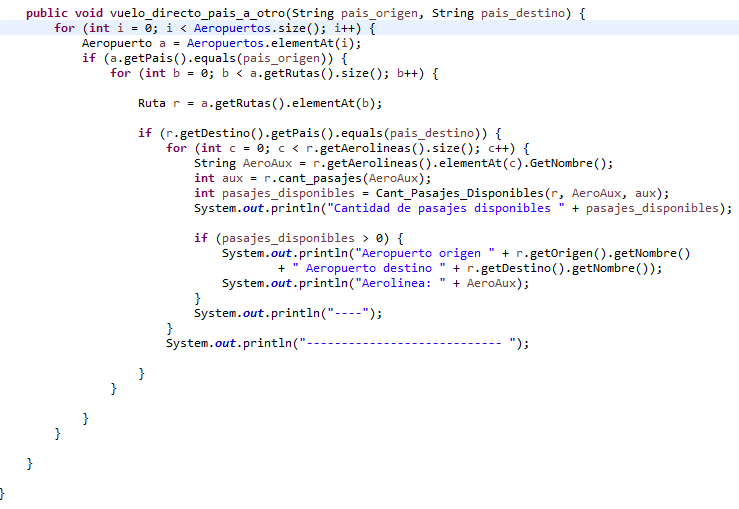
Este algoritmo tiene una complejidad o(n)^2

**Implementación de servicio 3**

Los nuevos métodos que utilice para implementar este servicio son:

vuelo\_directo\_pais\_a\_otro: Permite saber cuáles son los distintos vuelos que hay de un país a otro

**Seguimiento del algoritmo vuelo\_directo\_pais\_a\_otro:**



for (int i = 0; i < Aeropuertos.size(); i++) {

Aeropuerto a = Aeropuertos.elementAt(i);

if (a.getPais().equals(pais\_origen)) {

Recorre el vector de Aeropuertos que tiene cargado el sistema y busca el aeropuerto con el mismo país que paso el usuario como parámetro

--

for (int b = 0; b < a.getRutas().size(); b++) {

Ruta r = a.getRutas().elementAt(b);

if (r.getDestino().getPais().equals(pais\_destino)) {

Busca entre las destinas rutas la que tiene como destino un aeropuerto en el país que solicito el usuario

---

for (int c = 0; c < r.getAerolineas().size(); c++) {

String AeroAux = r.getAerolineas().elementAt(c).GetNombre();

int aux = r.cant\_pasajes(AeroAux);

int pasajes\_disponibles = Cant\_Pasajes\_Disponibles(r, AeroAux, aux);

Recorre cada una de las aerolíneas de la ruta para calculas la cantidad de pasajes que disponen actualmente.

--

if (pasajes\_disponibles > 0) {

System.out.println("Aeropuerto origen " + r.getOrigen().getNombre()+ " Aeropuerto destino " + r.getDestino().getNombre());

System.out.println("Aerolinea: " + AeroAux);

}

En caso de que la cantidad de pasajes disponibles sea mayor a 0 se mostrara toda la información solicitada.

Este algoritmo tiene una complejidad o(n)^2

**Conclusiones**

Luego de realizar el trabajo puede llegar a la conclusión de que el mismo fue muy útil para poder interiorizarme acerca de cómo es el funcionamiento de un grafo y que tan importantes es ir adaptando nuestros algoritmos, con los tiempos de ejecución y la utilización de la adecuado utilización del espacio de memoria requerido, para poder llegar a un óptimo funcionamiento de nuestra aplicación, esto se ve reflejado al momento de seleccionar la estructura de trabajo que se utilizó para desarrollar la este trabajo practico especial. Finalmente me hubiera gustado tener un poco más de tiempo para lograr optimizar de una mejor manera los diversos servicios, pero esto me servirá para realizar futuros trabajos prácticos con más anticipación.