



Programación 3 - TUDAI 2019

Trabajo Práctico Especial

17/05/2019

Cánneva, Juan Bernardo

jcanneva@gmail.com

<https://github.com/jcanneva/Programacion-3/>

Introducción

A partir del problema dado, se realizó la implementación de un sistema de servicios de viajes. En el cual se decidió que para su desarrollo la solución fuese representada a través de un grafo, en donde cada vértice es simbolizado por un aeropuerto y cada arista con una ruta.

En ese contexto es donde se presenta el siguiente informe, en donde se detalla la solución del mismo como así también el desarrollo de los servicios ofrecidos, tales como listar aeropuertos y reservas de rutas u obtener diferentes vuelos en base a distintas rutas a partir de las características que puede presentar la misma, como ser de cabotaje o la distancia que posee.

Cabe aclarar que los datos que el sistema posee fueron provistos por la cátedra a través de distintos archivos csv, y que las respuestas generadas por los servicios implementados también son dadas en un archivo de salida csv.


A través de la captura de requerimientos realizada se logró fijar los siguientes objetivos:

1. Brindar una solución a todos los servicios dados
2. Implementar esa solución a través de un grafo
3. Generar respuestas eficientes a los servicios a través de distintos algoritmos

Modelado el problema

Para implementar del problema planteado, se realizó un modelado del mismo, adquiriendo una idea previa antes de desarrollar el funcionamiento que sistema debía tener.

En un principio rápidamente se obtuvieron la clases de Grafo, Ruta y Aeropuerto, representando así la solución a través de un grafo y darle el comportamiento que el mismo posee.



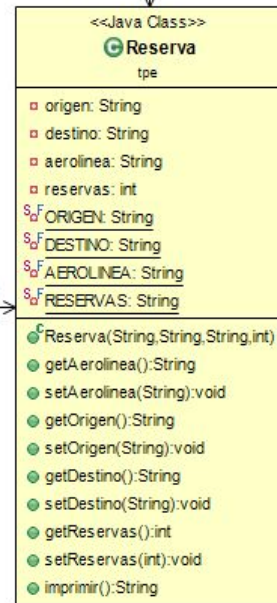
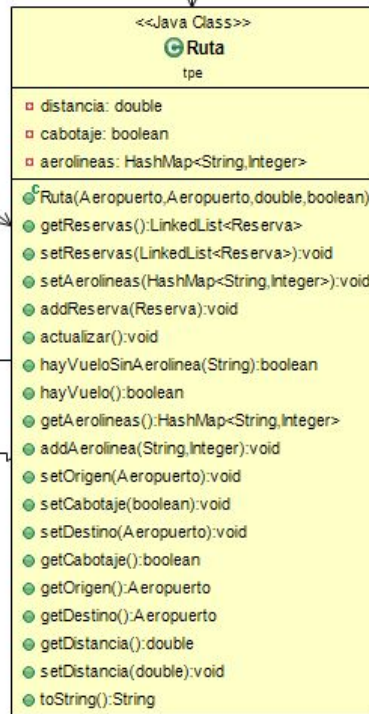
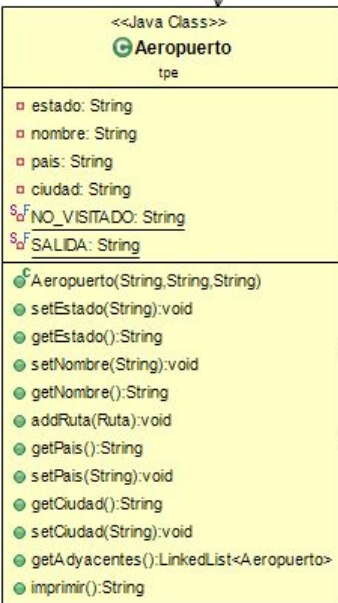
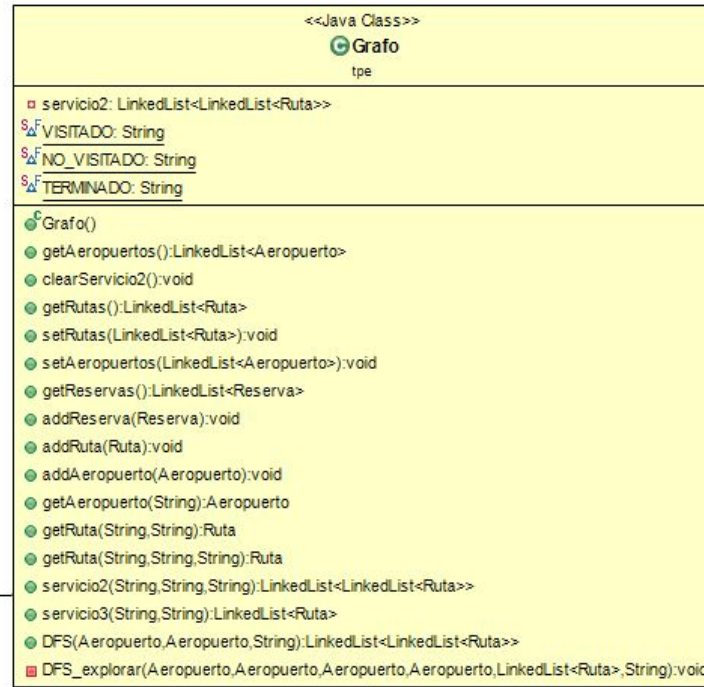
Seguido de esto se busco que estructura convenía utilizar, analizando varias de las provistas por cátedra, y viendo a la vez, las ventajas y desventajas que había en cada una a la hora de su implementación.

Las estructuras analizadas fueron las siguientes:

1. *ArrayList*: Lista implementada en base al array. Su tamaño varía dinámicamente, el cual tiene un costo adicional. Su complejidad también varía. La ventaja es que al insertar un elemento tiene un costo bajo $O(1)$ (si es al final), al igual que si se desea obtener un elemento. La desventaja es que si se quiere insertar en una posición o se quiere eliminar un elemento, su complejidad es el tamaño de la lista $O(n)$ siendo n el tamaño de la misma.
2. *LinkedList*: Lista implementada con nodos y referencias a otros nodos. Su tamaño, también se modifica dinámicamente. Insertar o eliminar un elemento al principio o final es $O(1)$, pero si se quiere obtener un elemento específico su complejidad es $O(n)$, ya que se deben recorrer todos los elementos (en el peor de los casos).

En base a las estructuras analizadas se decidió utilizar para el desarrollo de la implementación, *LinkedList*, debido a que si bien su acceso para obtener un elemento puede ser el tamaño de la lista en el peor de los casos, si se piensa en la escalabilidad, es decir, si el sistema creciera y se desean hacer altas, bajas o modificaciones al tener punteros de los elementos, el costo sería mucho menor que si se utiliza *ArrayList*.

Después de haber elegido las estructuras y haber modelado el problema tentativamente se llegó al siguiente diagrama de clases:



- Clase Grafo (sistema)

```
Atributos {  
    LinkedList<Aeropuerto> aeropuertos  
    LinkedList<Ruta> rutas  
    LinkedList<Reserva> reservas  
    LinkedList<LinkedList<Ruta>> servicio2  
}
```

Para desarrollar el grafo se decidió utilizar la representación de lista de listas, es por eso que todos los atributos de esta clase son listas y se implementaron a través de LinkedList por los motivos que se explicaron anteriormente.

A su vez esta es la clase principal ya que sus atributos son del tipo de las demás clases, y porque, su comportamiento es el que da respuesta a los servicios que se buscó desarrollar.

Para eso se crearon los siguientes métodos:

1. getAeropuerto(String n1): devuelve un aeropuerto en base al String que es el nombre del aeropuerto
2. getRuta(String n1, String n2): devuelve una ruta en base a los String que son los nombres de los aeropuertos origen y destino
3. servicio2(String n1, String n2, String n3): devuelve una lista de listas de rutas en base a los String que son nombres de aeropuertos origen y destino, y nombre de aerolínea
4. servicio3(String n1, String n2): devuelve una lista de rutas en base a los Sting que son nombres de países

- Clase Aeropuerto

```
Atributos{  
    LinkedList<Ruta> rutas  
    String estado  
    String nombre  
    String país  
    String ciudad  
}
```

Además de los atributos de String nombre, estado, país y ciudad posee una LinkedList que apunta a distintas rutas propias del aeropuerto.

Además de los métodos de obtener y establecer (get/set), posee addRuta, para agregar una ruta determinada y el método getAdyacentes, el cual devuelve una lista de Aeropuertos que conectan con el mismo.

- Clase Ruta

Atributos {

```
    LinkedList<Reserva> reservas;  
    Aeropuerto origen;  
    Aeropuerto destino;  
    double distancia;  
    boolean cabotaje;  
    HashMap<String, Integer> aerolineas;
```

}

La clase posee una LinkedList de reservas y un HashMap aerolíneas donde la clave es el nombre de la aerolínea y el valor los asientos que posee. También tiene una referencia al origen y al destino, ambos aeropuertos, un boolean para saber si es entre un mismo país y un double con los kilómetros.

Los principales métodos de la clase son un void actualizar, el cual actualiza para cada aerolínea los asientos que posee según las reservas del momento. También posee dos funciones booleanas para saber si hay al menos un vuelo con pasajes para saber si se puede viajar por la ruta y la otra similar pero filtrando además una aerolínea dada.

- Clase Reserva

Atributos{

```
    String origen;  
    String destino;  
    String aerolinea;  
    int reservas;
```

}

Esta clase no estaba prevista, pero se decidió su implementarla luego de ver cómo venían los datos dados en los archivos csv.

Se puede definir como una clase diccionario, ya que no tiene comportamiento en sí, porque sus métodos sólo son obtener y establecer (get/set) y sólo guarda información.

Implementación de los servicios

I. Servicio 1: Verificar vuelo directo

Para resolver este servicio se implementó un método `getRuta` en el grafo, en el cual se itera a través de las rutas y por cada ruta se pregunta si cumple con los requisitos de la búsqueda. Se decidió esta implementación ya que el grafo posee como atributo una lista con todas las rutas .

La complejidad del algoritmo utilizado es $O(n)$.

La complejidad resulta así debido a que mientras se itera la lista de rutas se pregunta si esa ruta cumple con las condiciones. Estas son, si contiene la aerolínea dada, y ahí el acceso al `HashMap` de aerolíneas en la Ruta es $O(1)$, al igual que las condiciones que preguntan si el nombre es el mismo.

Para el seguimiento del algoritmo se utilizó el menú y el timer para ver los tiempos de ejecución con el dataset dado y con un dataset de menores elementos.

Data set con menor cantidad de elementos:

```
Ingrese aeropuerto origen:  
Ministro Pistarini  
Ingrese aeropuerto destino:  
John F. Kennedy  
Ingrese aerolínea deseada  
United Airlines  
Tiempo de ejecucion 0.1242  
Existe el vuelo  
Distancia: 8535.74  
Pasajes disponibles: 76
```

Data set con mayor cantidad de elementos:

```
Ingrese aeropuerto origen:  
Ministro Pistarini  
Ingrese aeropuerto destino:  
John F. Kennedy  
Ingrese aerolínea deseada  
United Airlines  
Tiempo de ejecucion 0.2678  
Existe el vuelo  
Distancia: 8535.74  
Pasajes disponibles: 76
```


II. Servicio 2: Obtener vuelos sin aerolínea

Para este resolver el Servicio 2 se creó el método `servicio2` y el atributo `servicio2`, el cual es una lista de listas donde se guardan las soluciones, ambos en la clase `Grafo`. El método obtiene los aeropuertos origen y destino, en base a los String ingresados y a su vez llama a un función privada `DFS` la cual retorna una copia de la lista de listas. Lo que este método hace es básicamente, inicializar una lista en vacío, setear el estado del origen en visitado y llamar al método `DFS_explorar`, con los parámetros de: la lista, el origen y el destino. Cuando vuelve, retorna la copia.

A continuación se adjunta una imagen con un breve pseudocódigo del método `DFS_explorar`.

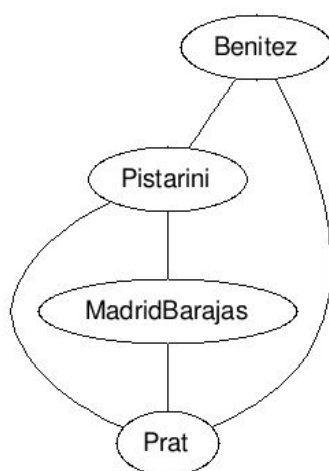
```
DFS_explorar(actual, destino, lista) {
    si actual = destino
        solucion.add(lista);
    sino {
        para cada ruta de actual {
            si el destino de la ruta es no visitado {
                actual = ruta.destino;
                actual = visitado;
                lista.add(ruta);
                DFS_explorar(actual, destino, lista);
                lista.remove(ruta);
                actual = no visitado;
            }
        }
    }
}
```

El algoritmo que se utiliza es `backtracking`. Lo que hace es preguntar si el aeropuerto en el que estoy es igual a destino, si lo es, es solución entonces agrega la lista a lista de listas. Si no es solución itera las rutas del actual, para cada ruta pregunta si el destino no está visitado, sino lo está, el actual se vuelve destino, se setea actual como visitado y se agrega a la lista la ruta. Después se llama al mismo método recursivamente con actual destino y la lista. Cuando vuelve de la recursión se elimina la ruta de la lista y se setea el actual como no visitado.

La elección de este algoritmo se dio porque se necesitaba seleccionar y deseleccionar un elemento, en este caso las rutas.

La complejidad de este algoritmo es $O(n!)$, en el peor de los casos.

Abajo se realizó un seguimiento del algoritmo sobre un grafo pormenorizado.



Suponiendo que el recorrido sería de Madrid-Barajas a Comodoro Benitez.

El algoritmo empieza preguntado si es solución, como no lo es pide las rutas de Madrid-Barajas, en este caso serían las rutas con destino Pistarini y Prat, y las explora, es decir, pregunta si el destino, en el primer caso sería Pistarini, no está visitado. Si no lo está, Pistarini se vuelve el actual, se setea como visitado y va a explorar Pistarini, repitiendo la ejecución anterior.

Luego de explorar Pistarini, va a explorar Prat y vuelve a repetir el proceso.

Imagen de la salida sin filtrar por aerolínea

```

Rutas: De Madrid-Barajas a Comodoro Benitez

Madrid-Barajas -- Ministro Pistarini [Delta, LATAM] Ministro Pistarini -- El prat [Aerolineas, LATAM] El prat -- Comodoro Benitez [United Airlines, Delta]
1)Escalas: 2 2)Distancia: 21705.98

Madrid-Barajas -- Ministro Pistarini [Delta, LATAM] Ministro Pistarini -- Comodoro Benitez [United Airlines, Delta]
1)Escalas: 1 2)Distancia: 11213.730000000001

Madrid-Barajas -- El prat [Delta, Avianca] El prat -- Ministro Pistarini [Aerolineas, LATAM] Ministro Pistarini -- Comodoro Benitez [United Airlines, Delta]
1)Escalas: 2 2)Distancia: 12125.95

Madrid-Barajas -- El prat [Delta, Avianca] El prat -- Comodoro Benitez [United Airlines, Delta]
1)Escalas: 1 2)Distancia: 1633.7
  
```

III. Servicio 3: Vuelos disponibles

Este servicio se resolvió similarmente al servicio 1, ya que él mismo realiza una iteración sobre la lista de rutas que posee el grafo. Por cada iteración pregunta si el nombre del país de origen y de destino son iguales a los que ingresó el usuario. Si

coinciden se agrega a una lista salida, en la cual se lleva el registro de todas las rutas que cumplen con esa condición, para luego retornarla.

La complejidad del algoritmo continúa siendo $O(n)$, donde n es el tamaño de la lista de rutas.

Ejemplo con dataset de menor cantidad:

```
Ingrese pais:
USA
Ingrese pais:
ARG
Tiempo de ejecucion 0.1382
John F. Kennedy -- Ministro Pistarini
John F. Kennedy -- Jorge Newbery
```

Ejemplo con dataset de mayor cantidad:

```
Ingrese pais:
USA
Ingrese pais:
ARG
Tiempo de ejecucion 0.2564
John F. Kennedy -- Ministro Pistarini
John F. Kennedy -- Jorge Newbery
John F. Kennedy -- Armando Tola
Logan -- Armando Tola
```

Funcionamiento de la aplicación

La aplicación arranca instanciando la clase Grafo y levantando los datos de los archivos con el CSVReader, en el grafo. Una vez hecho esto actualiza el grafo con las reservas del día. Para realizar esta actualización se iteran las rutas del grafo, y por cada ruta se iteran las reservas y luego las aerolíneas, donde coinciden se realiza la actualización. Esta actualización es muy costosa, ya que por cada ruta iterada se iteran todas sus reservas y a su vez todas las aerolíneas. Cabe aclarar sobre la entrada de los datos que hubiese sido más eficiente que los datos sobre la aerolínea y las reservas estuviesen en un mismo archivo csv para poder manipularlos de forma más sencilla.

Una vez que el sistema se encuentra actualizado se llama al menú con el grafo, y se listan los distintos servicios que se le ofrecen al usuario.

Cada vez que el usuario realiza una consulta al grafo la respuesta es guardada en un archivo csv de salida. Por cada respuesta se genera un archivo diferente.

Se deja aclarado también que dentro de los dataset hay una carpeta llamada “Menor” que contiene los elementos menores con los que fueron realizadas las pruebas de ejecución.

Además se dejó el timer implementado para que se puedan ver los tiempos de cada servicio.

Conclusiones

Al terminar este trabajo se pudo lograr cumplir los objetivos propuestos en un principio. Sin embargo, en el contexto en que se desarrolló el sistema se podrían haber realizado soluciones más eficientes, como por ejemplo realizar una clase dar respuesta a cada servicio, evitando así realizar diferentes operaciones a la hora de mostrar por pantalla los diferentes requisitos que fueron dados.

Otra mejora que siempre se tuvo en cuenta fue la de aplicar alguna estructura que brinde un acceso más eficiente, para no tener que hacer recorridos innecesarios.

Pero como no se sabe que tanto puede crecer la aplicación se decidió sacrificar el acceso en pos de que en un futuro se logre tener un sistema eficiente y escalable.