Informe Trabajo Práctico Especial
Programación 3 2019

TUDAI - FCE - UNICEN

Integrantes:

Abate, Juan Manuel juan.abate@gmail.com
Nava, Maximo Santiago maximonava94@gmail.com

Repositorio código fuente: https://github.com/JuanManuelAbate/TUDAI-prog3-tpe

Introducción al problema:

Se requirió implementar un sistema de gestión de aeropuertos, rutas y reservas, a partir de un modelo simplificado de los mismos para brindar un servicio centralizado a viajeros frecuentes. A partir de esto, el sistema brinda distintas funcionalidades al usuario. A saber:

1. Listar todos los aeropuertos.
2. Listar todas las reservas realizadas.
3. Servicio 1: Verificar vuelo directo.
4. Servicio 2: Obtener vuelos sin aerolínea.
5. Servicio 3: Vuelos disponibles.

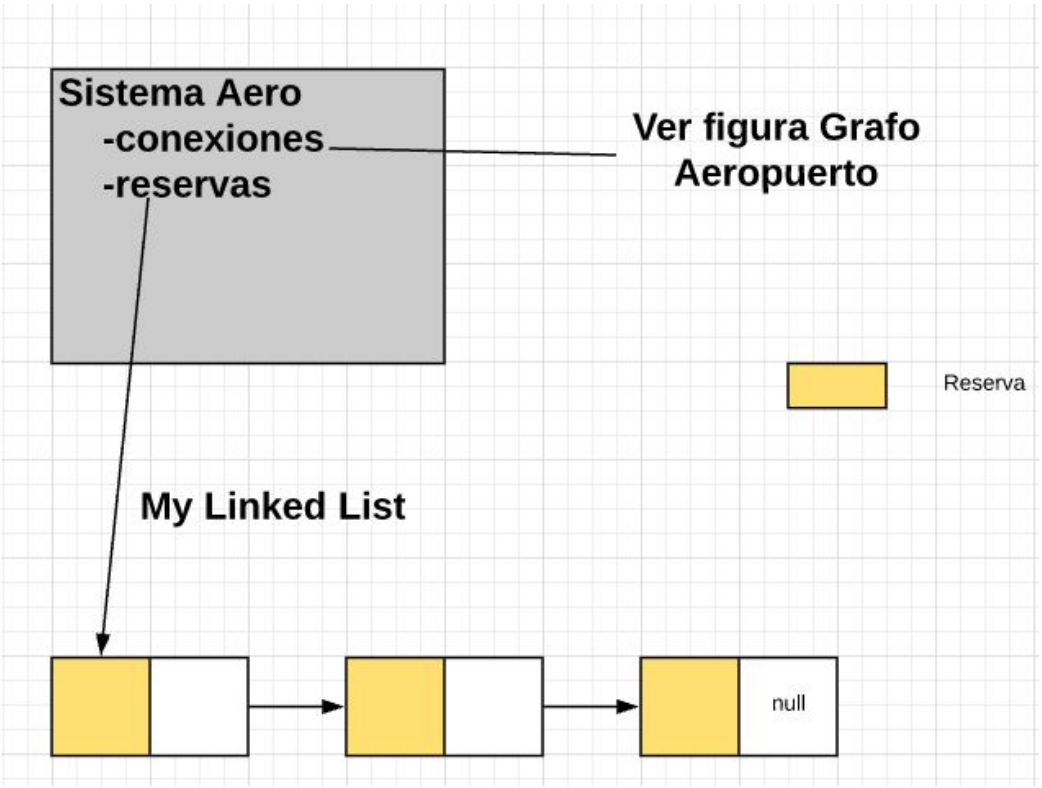
El objetivo del trabajo fue identificar los distintos actores del dominio y generar las soluciones a los distintos incisos requeridos, teniendo en cuenta las estructuras de datos a utilizar y los algoritmos necesarios para resolver estas.

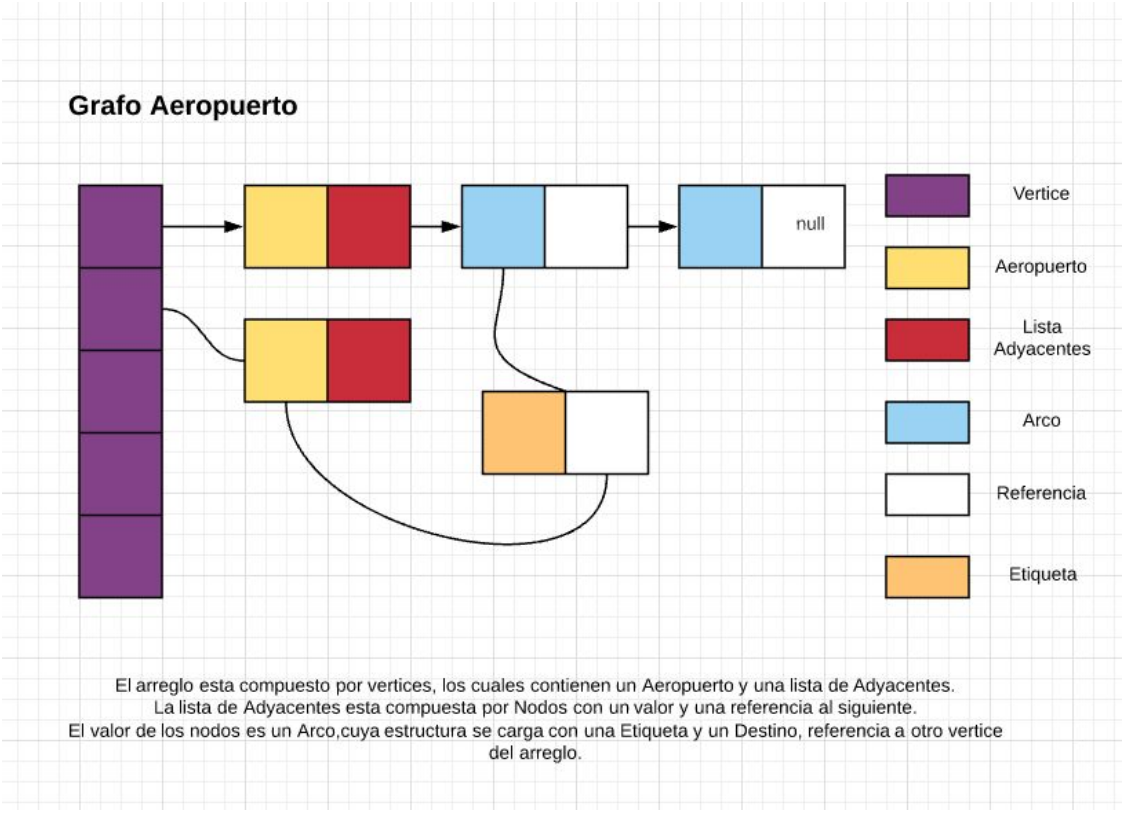
Las tareas involucradas para resolver el trabajo fueron :

- Desde un primer foco, se logró un entendimiento profundo de la problemática y de los puntos requeridos a resolver.
- A partir del punto anterior, se diseñaron las entidades del dominio y estructuras de datos necesarios para modelar el problema.
- Diseñada la solución, se procedió al desarrollo de la misma.
- En última instancia, se realizaron distintos casos de pruebas para cada uno de los servicios requeridos, validando el correcto funcionamiento de los mismos.

Modelado del problema:

Para modelar el problema creamos una clase SistemaAero la cual se encarga de las funcionalidades solicitadas, la gestión de las reservas y todo lo referido a los aeropuertos y las conexiones de los mismos. Para llevar esto acabo la clase cuenta internamente con una lista de Reserva que es la clase que representa una reserva del dominio y un GrafoAeropuerto, que es una especialización de la clase Grafo(grafo doblemente vinculado), esta última se encarga de almacenar los Aeropuertos, sus conexiones y la información de las mismas que se almacena en una clase Etiqueta.





El arreglo esta compuesto por vertices, los cuales contienen un Aeropuerto y una lista de Adyacentes.
La lista de Adyacentes esta compuesta por Nodos con un valor y una referencia al siguiente.
El valor de los nodos es un Arco,cuya estructura se carga con una Etiqueta y un Destino, referencia a otro vertice del arreglo.

Tipos de datos identificados y relaciones entre ellos.

Aeropuerto:
Clase que representa un aeropuerto del dominio. La misma encapsula los siguientes datos:
Nombre
Ciudad
Pais

Reserva:
Clase que representa una reserva realizada. La misma encapsula los siguientes datos:
Aerolinea
Asientos
Origen
Destino

Etiqueta:
Clase que representa la información pertinente a la relación entre dos aeropuertos. La misma encapsula los siguientes datos:
AerolineaAsientos
Cabotaje
Km

Asiento:
Clase que representa y administra los asientos totales y disponibles. La misma encapsula los siguientes datos
Asientos disponibles
Asientos totales

SistemaAero:
Clase que representa el sistema de gestión de aeropuertos y reservas. La misma encapsula los siguientes datos:
Conexiones
Reservas

Relaciones entre sí:

El SistemaAero contiene en sus datos una instancia de la estructura GrafoAeropuerto llamada conexiones y una lista de reservas. La primera estructura guarda las relaciones entre los Aeropuertos y la información de cada una de estas relaciones (Etiqueta).
Por otra parte, la lista de reservas lleva un registro de todas las reservas realizadas en el sistema.
Dentro de la Etiqueta podemos encontrar la relación entre las distintas aerolíneas de la misma con los Asientos totales y disponibles.

Estructuras de datos elegidas para implementar cada uno de los tipos de datos.

Las estructuras que utilizamos son una lista vinculada y un grafo doblemente dirigido.

Ventajas y desventajas de las estructuras elegidas. Complejidad temporal y espacial.

-Lista vinculada (utilizada para la lista de reservas y arcos adyacentes de un vértice), implementada con nodos.

Ventajas:

- Tamaño dinamico.
- Inserción en factor temporal constante.
- Posibilidad de implementar iteradores sobre la misma.

Desventajas:

- Imposibilidad de acceso directo a los elementos.
- Complejidad O(n) a la hora de obtener un elemento determinado.

-Grafo doblemente vinculado (utilizado para la conexiones), implementado con un arreglo vértices, los cuales contienen sus adyacentes en una lista vinculada.

Ventajas:

- Mecanismo sencillo de iteración sobre los vértices.
- Respecto al espacio, nos pareció una implementación más adecuada a la dimensión de nuestro problema teniendo en cuenta la cantidad de conexiones.

Desventajas:

- Consultas costo O(n) para encontrar un vértice o añadir una relación entre estos.

Implementacion de los servicios

Servicio 1:

Para resolver este servicio se realizan los siguientes pasos:

1. El SistemaAero recibe tres parámetros de tipo String, que representan el nombre del aeropuerto de origen, el nombre del aeropuerto de destino y el nombre de la aerolínea a utilizar.
2. Con esos datos el SistemaAero le solicita a su atributo conexiones la etiqueta que guarda la información sobre la conexión de esos dos aeropuertos, en caso de no existir la conexión la respuesta será null.
3. El SistemaAero le solicita a la etiqueta obtenida que verifique si efectivamente contiene la aerolínea deseada, en caso de no contenerla la respuesta será null.
4. El SistemaAero le solicita a la etiqueta la cantidad de asientos disponibles para la aerolínea requerida.
5. El SistemaAero retorna una instancia de la clase de respuesta VueloDirecto con los km del viaje y la cantidad de asientos libres utilizando la información antes adquirida.

En un principio en la clase Etiqueta solo teníamos almacenada la relación entre aerolínea y cantidad total de asientos para la misma, por lo cual para luego calcular los asientos totales debiamos iterar toda la lista de reservas y aumentaba la complejidad, por lo tanto creamos la clase Asiento la cual además de tener los asientos totales tiene los asientos disponibles, luego modificamos la relación de la aerolínea en la etiqueta y gracias a esto disminuimos la complejidad del servicio.

Complejidad temporal

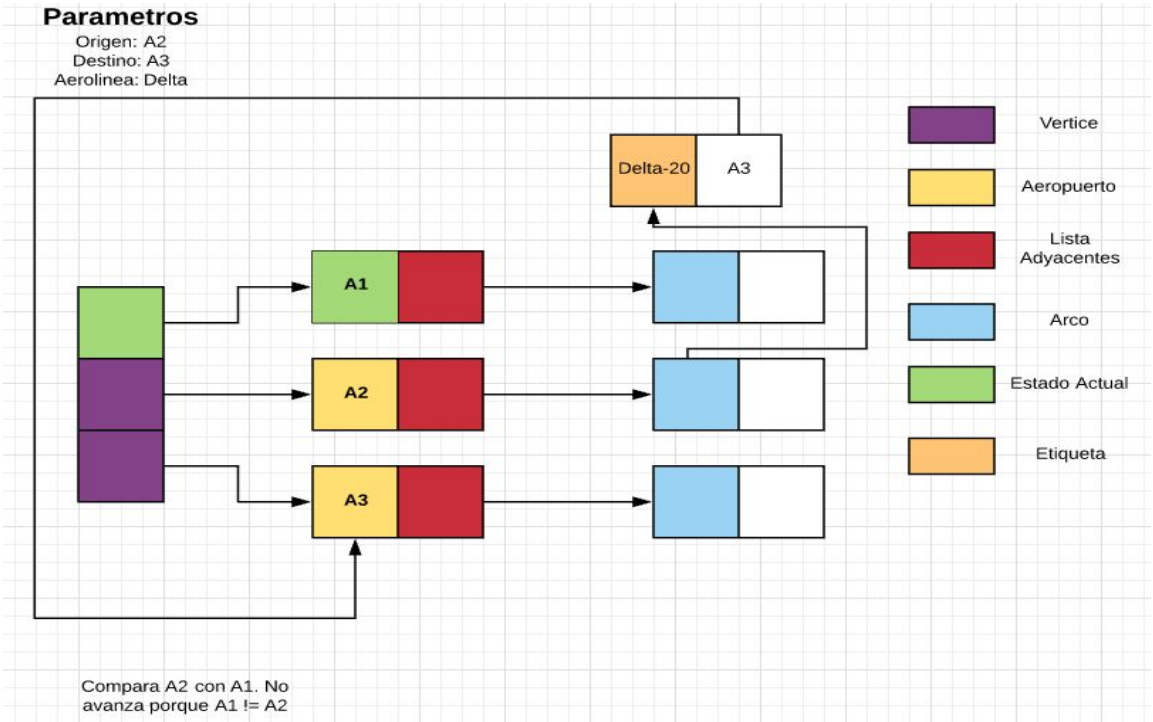
Clase GrafoAeropuerto método verificarVueloDirecto
n = cantidad de vertices.
Encontrar el vértice que contiene al aeropuerto de origen es de complejidad O(n), ya que en el peor de los casos se tendrán que recorrer todos los vértices.
En el peor de los casos por como está planteada la estructura de grafo el vértice en el que se encuentro el aeropuerto de origen podrá tener n-1 arcos adyacentes, sería el caso en el que el vértice deseado tenga conexión con todos los demás vértices, por lo cual la complejidad de esto es O(n-1) ya que se deben recorrer todos en el peor de los casos para encontrar el destino.
Entonces la complejidad total del algoritmo sería la suma las complejidad antes calculadas con lo cual llegamos a O(n + n-1) -> O(2n-1) -> O(n) despreciando constantes.

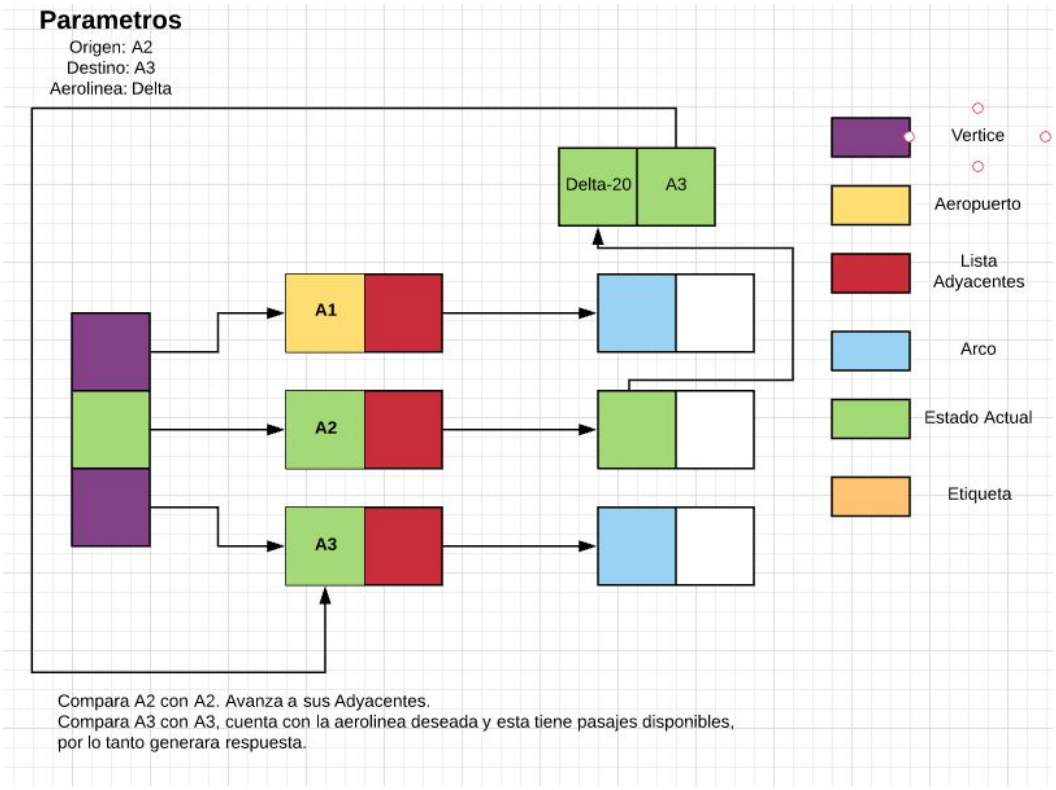
Clase Etiqueta método contieneAerolinea
m = cantidad de aerolíneas en la etiqueta.
Este método utiliza internamente el atributo aerolineasAsientos que es un Map de Java, la respuesta que retorna es la respuesta del método containsKey de la clase HashMap, que según lo investigado en el peor de los casos puede llegar a tener una complejidad de O(m)

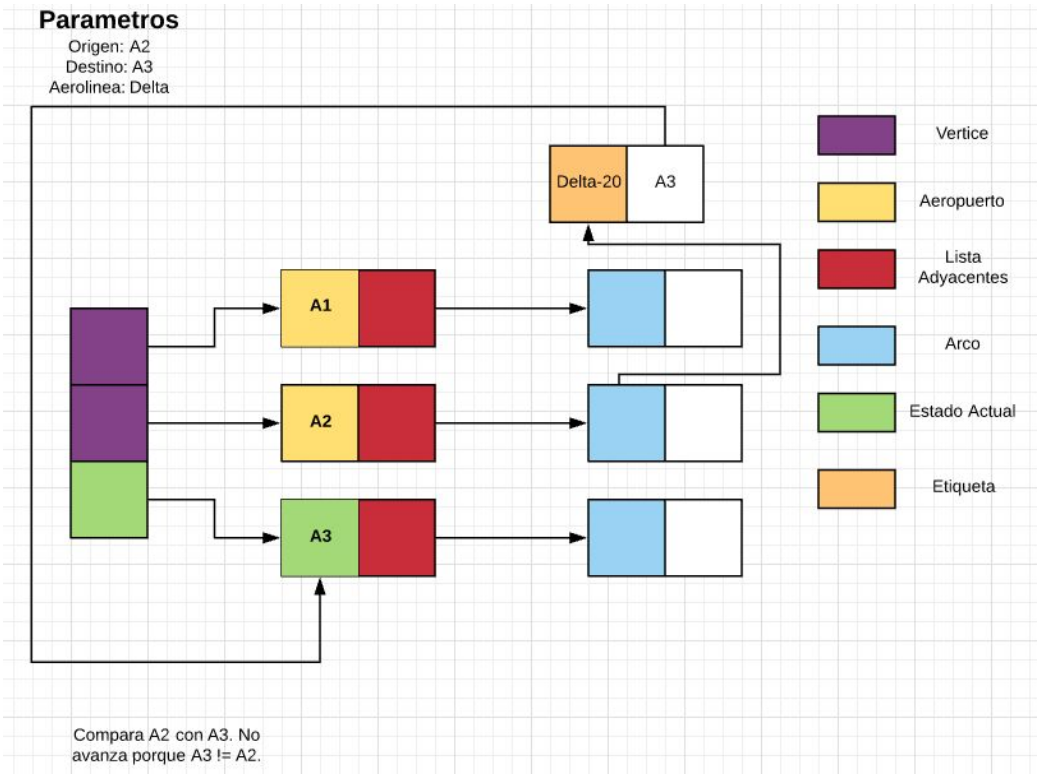
Clase Etiqueta método getAsientosDisponiblesAerolinea
m = cantidad de aerolíneas en la etiqueta.
Este método utiliza internamente el atributo aerolineasAsientos que es un Map de Java, la respuesta que retorna es la respuesta del método get de la clase HashMap que retorna la instancia de la clase Asiento asociada a esa aerolínea y luego a esta instancia se le solicita el método getAsientosDisponibles que tiene una complejidad de O(1), por lo cual, según lo investigado en el peor de los casos puede llegar a tener una complejidad de O(m), ya que el método get de la clase HashMap en el peor escenario puede llegar a esa complejidad.

Clase SistemaAero metodo verificarVueloDirecto
n = cantidad de vertices.
m = cantidad de aerolíneas en la etiqueta.
Este método utiliza un llamado a cada uno de los métodos antes analizados de forma no anidada, por lo cual la complejidad será
O(n + m + m) -> O(n + 2m) -> O(n + m) -> esto puede ser interpretado como el máximo entre O(n) y O(m)

Seguimiento:




Parametros
Origen: A2
Destino: A3
Aerolinea: Delta
Delta-20
A3
A1
A2
A3
Vertice
Aeropuerto
Lista Adyacentes
Arco
Estado Actual
Etiqueta
Compara A2 con A2. Avanza a sus Adyacentes.
Compara A3 con A3, cuenta con la aerolinea deseada y esta tiene pasajes disponibles, por lo tanto generara respuesta.


Parametros
Origen: A2
Destino: A3
Aerolinea: Delta
Delta-20
A3
A1
A2
A3
Vertice
Aeropuerto
Lista Adyacentes
Arco
Estado Actual
Etiqueta
Compara A2 con A3. No avanza porque A3 != A2.

Servicio 2

Para resolver este servicio desarrollamos un algoritmo del tipo backtracking con algunas podas, ya que en el mismo se solicitaban todos los posibles vuelos de un aeropuerto a otro con algunas restricciones, el algoritmo realiza los siguientes pasos

1. El SistemaAero recibe tres parámetros de tipo String, que representan el nombre del aeropuerto de origen, el nombre del aeropuerto de destino y el nombre de la aerolínea a utilizar.
2. Con esos datos el SistemaAero llama al método vuelosDisponibles en su atributo conexiones.
3. El método vuelosDisponibles de la clase GrafoAeropuerto en primera instancia busca el vértice el cual contiene el aeropuerto de origen, luego con este vértice llama a otro método interno el cual es el backtracking que se encarga de recorrer desde este vértice todos los posibles caminos hacia el vértice que contiene el aeropuerto destino.
4. Con las soluciones que el backtracking va encontrando se crean instancias de la clase VueloDisponibleAeroExcluyente, las cuales se van insertando en una lista que contiene el total de las soluciones.

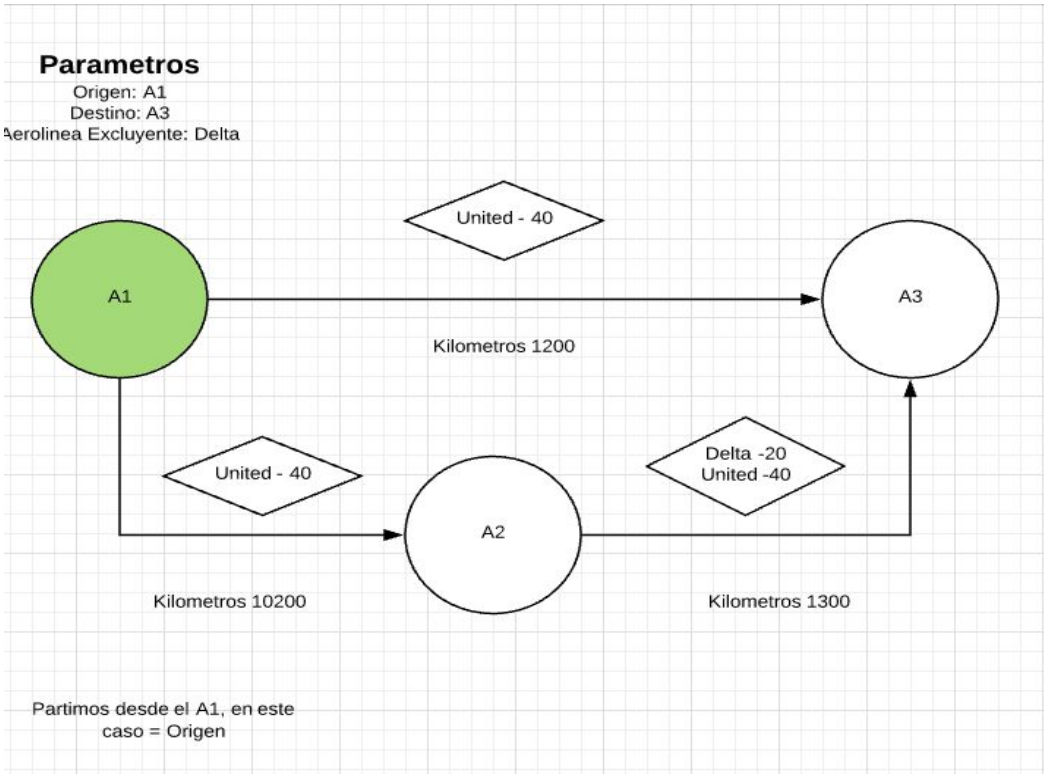
Complejidad temporal:

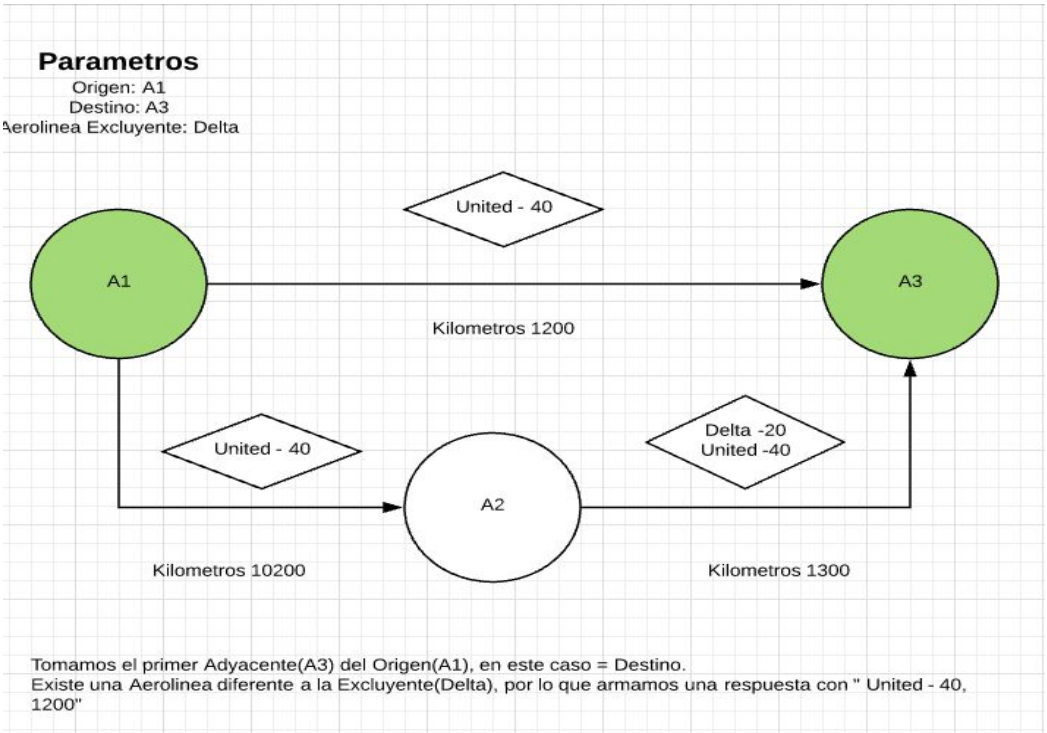
Clase GrafoAeropuerto metodo getVerticePorNombreAeropuerto
n = numero de vertices.
Este método recorre el arreglo de vértices y busca el que contiene el aeropuerto de origen, la complejidad temporal del mismo es O(n).

Clase GrafoAeropuerto método vuelosDisponiblesPrivate
n = numero de vertices.
m = numero de arcos.
En el caso que el grafo sea muy conectado el número de arcos puede llegar a ser n - 1, por lo cual por cada arco en cada vértice se llamará al método, con lo cual se llega a una complejidad de O(n^n)

Clase SistemaAero método vuelosDisponibles
Teniendo en cuenta los anteriores métodos la complejidad temporal es de O(n^n)

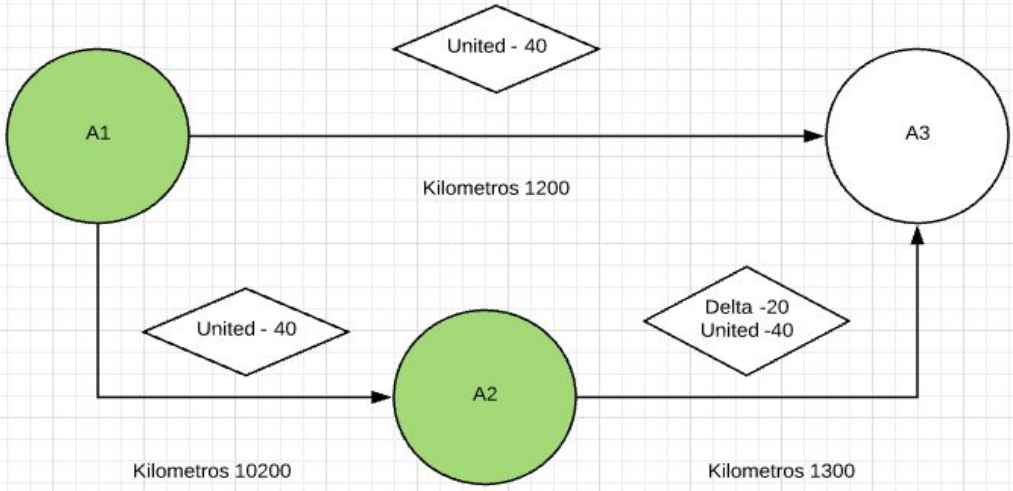
Seguimiento:





Parametros

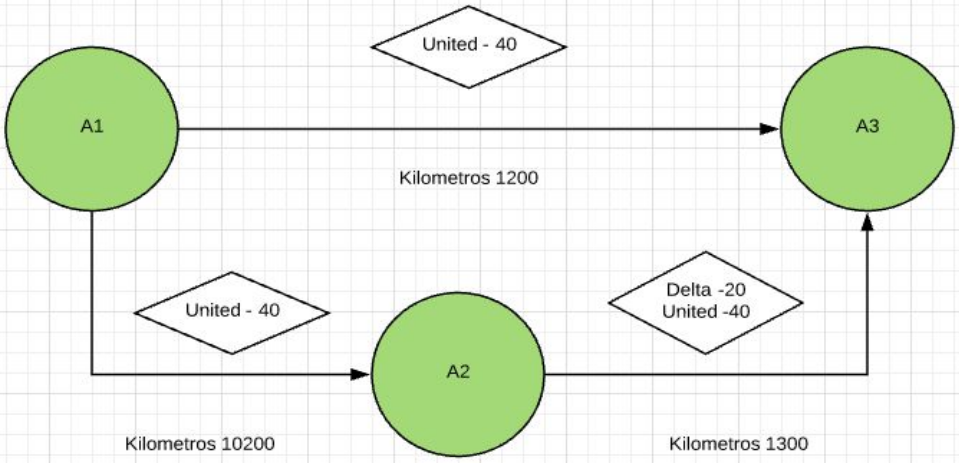
Origen: A1
Destino: A3
Aerolinea Excluyente: Delta



Tomamos el segundo Adyacente(A2) del Origen(A1), diferente a Destino.
Existe una Aerolinea diferente a la Excluyente(Delta), por lo que recorreremos los adyacentes de A2.
Agregamos la cantidad de Kilometros (10200) y Escala = 1 al recorrido.

Parametros

Origen: A1
Destino: A3
Aerolinea Excluyente: Delta



A partir del segundo Adyacente(A2) del Origen(A1), recorremos sus adyacentes y llegamos a Destino (A3).
Como A3 = Destino y existe una Aerolinea entre A2 y A3 distinta de la Excluyente, armamos la respuesta con la suma de kilometros A2-A3 (1300) + la distancia acumulada previamente (10200). Total = 11500. Escalas= 1.

Servicio 3:

Para resolver este servicio se realizan los siguientes pasos:

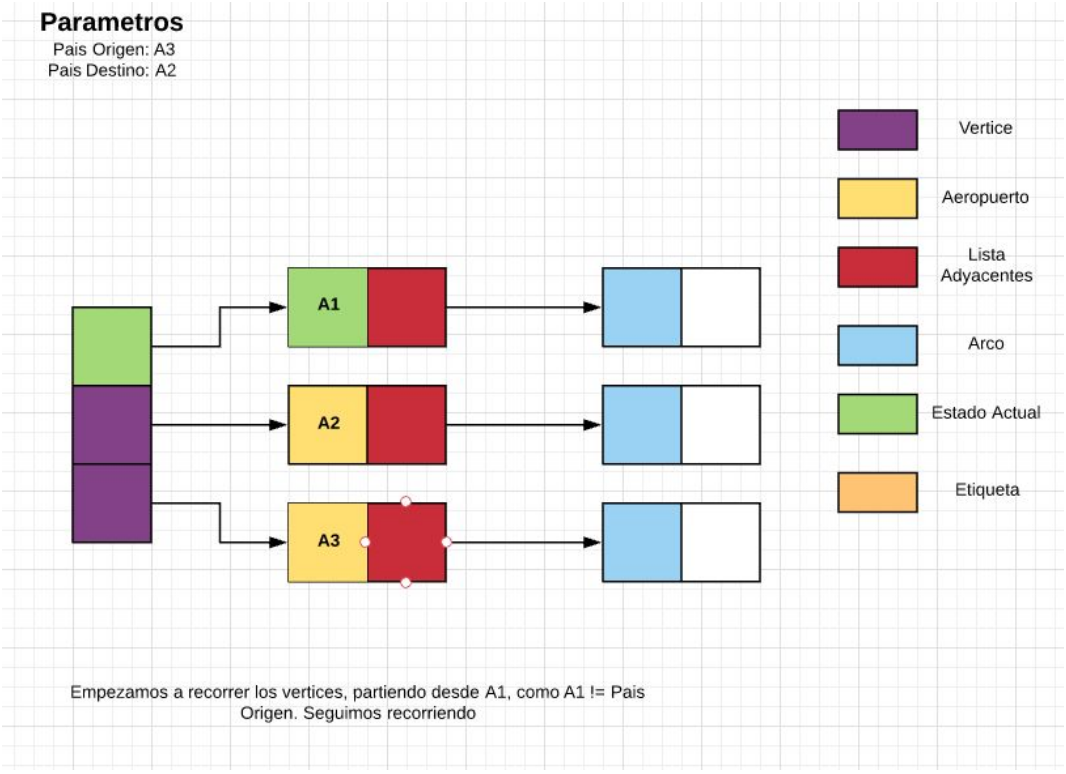
1. El SistemaAero recibe dos parámetros de tipo String, que representan el país de origen y el país de destino.
2. Con estos datos el SistemaAero llama al metodo vuelosDirectosEntrePaises de su atributo conexiones.
3. El metodo vuelosDirectosEntrePaises de la clase GrafoAeropuerto itera cada uno de los vértices del grafo buscando los que tienen almacenados aeropuertos que contengan el país de origen deseado.
4. Para los aeropuertos que sean afirmativos del paso anterior se solicita la lista de adyacentes del mismo y se la itera buscando los los aeropuertos que contengan el país de destino.
5. Se le solicita a las etiquetas de los arcos que contienen el país de destino el conjunto de aerolíneas con pasajes disponibles.
6. Para los casos que hay aerolíneas con pasajes disponibles se crea una instancia de la clase respuesta VueloDirectoAerolineas y se las agrega a una lista solucion.
7. Como paso final se retorna la lista solución generada.

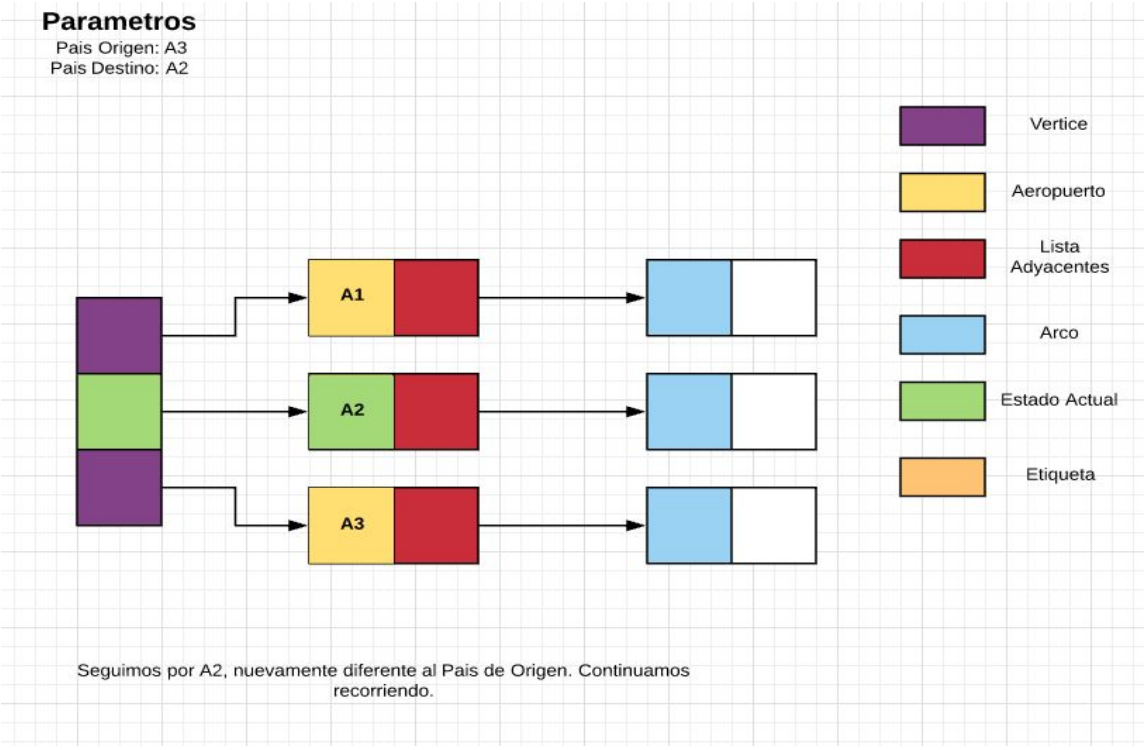
Complejidad temporal:

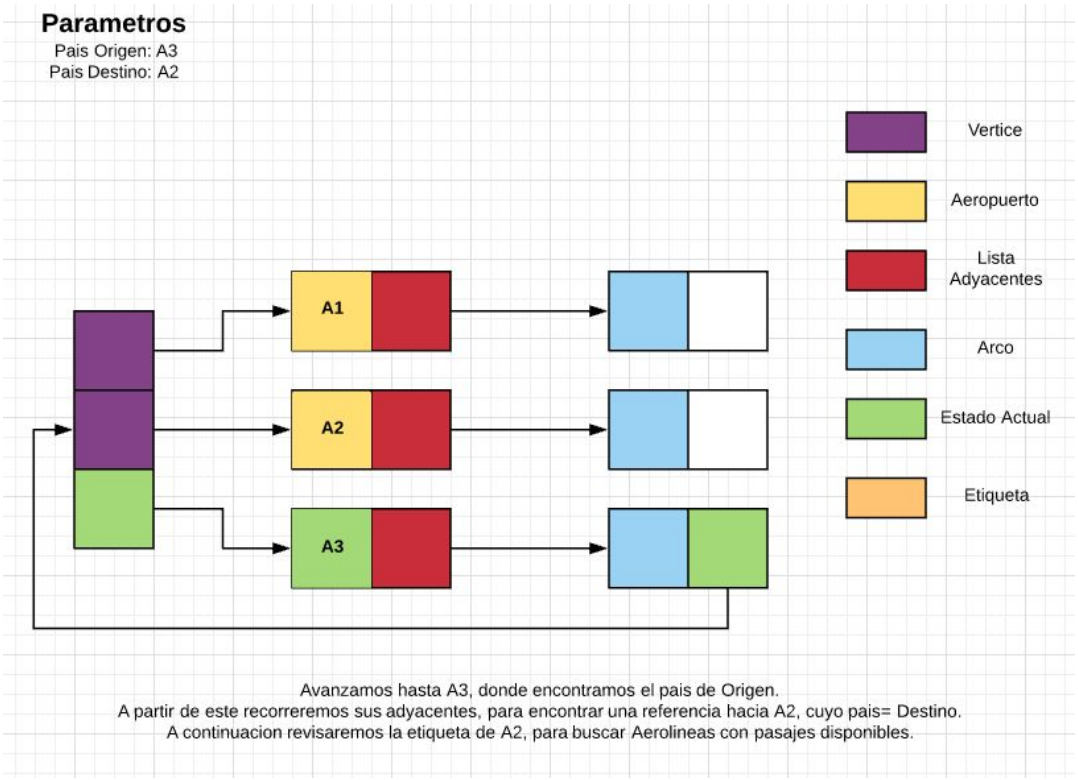
Clase Etiqueta método getAerolineasDisponibles
m = cantidad de aerolíneas en la etiqueta.
Este método utiliza internamente el atributo aerolineasAsientos que es un Map de Java, por cada aerolínea en el Map consulta la cantidad de asientos disponibles y si contiene asientos disponibles agrega la aerolínea y la cantidad a un nuevo mapa. La complejidad seria O(m^2), ya que se debe iterar cada una de las claves del hash y luego realizar un get que tiene complejidad O(m).

Clase GrafoAeropuerto método vuelosDirectosEntrePaises
n = cantidad de vertices.
a = cantidad de arcos adyacentes a un vértice n.
m = aerolíneas en la etiqueta del adyacente.
La complejidad en este caso será de O(n(a(m)) -> O(n*a*m)

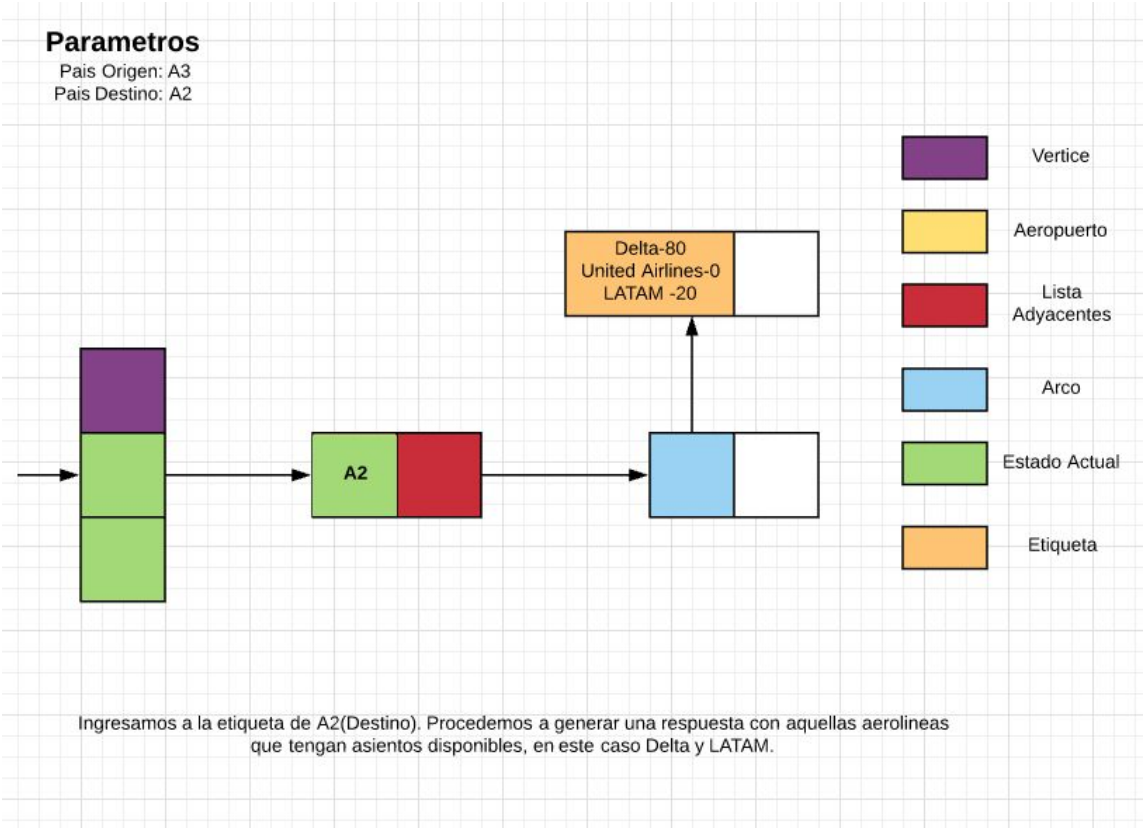
Seguimiento:







Avanzamos hasta A3, donde encontramos el pais de Origen.
A partir de este recorreremos sus adyacentes, para encontrar una referencia hacia A2, cuyo pais= Destino.
A continuacion revisaremos la etiqueta de A2, para buscar Aerolineas con pasajes disponibles.



Ingresamos a la etiqueta de A2(Destino). Procedemos a generar una respuesta con aquellas aerolineas que tengan asientos disponibles, en este caso Delta y LATAM.

Funcionamiento de la aplicación

Para ejecutar la aplicación deberán estar en el mismo directorio los archivos Aeropuertos.csv, Reservas.csv, Rutas.csv y SistemaAero.jar. Se debe navegar por consola hasta el directorio donde se ubicaron los archivos y ejecutar el comando java -jar SistemaAero.jar.

Al ejecutarse el sistema la consola mostrará el siguiente menú:

**********MENU**********
Ingrese un número de las siguientes opciones y presione enter:
1. Listar todos los aeropuertos.
2. Listar todas las reservas realizadas.
3. Servicio 1: Verificar vuelo directo.
4. Servicio 2: Obtener vuelos sin aerolínea.
5. Servicio 3: Vuelos disponibles.
6. Salir.
****

Ingresar en la consola la opción deseada para ejecutar la funcionalidad asociada a esa opción, para cada una se le solicitarán datos específicos los cuales también deben ser ingresados por consola. Luego de la ejecución de una funcionalidad se mostrará el resultado en la consola y además se generará un archivo con el mismo en el directorio donde se encuentre el ejecutable de la aplicación.

Los archivos generados componen su nombre de la siguiente forma:
DD-MM-AAAA HH-MM-SS - <Funcionalidad ejecutada>.csv

Ejemplos:
14-05-2019 23-11-53 - Listar todos los aeropuertos.csv
14-05-2019 23-15-27 - Servicio 3: Vuelos disponibles.csv

Conclusiones

Nos parecieron muy interesantes las estructuras vistas durante el desarrollo y solución del enunciado.
A partir de las herramientas brindadas por la cátedra, hemos podido discutir cuáles de las mismas utilizar para llegar a la solución elegida. A partir de esta discusión, hemos elegido las que, según nuestras creencias, son de mayor implementación para este caso.
Pensándolo en una escala mayor, encontramos que varias de estas estructuras son muy costosas para implementarlas en determinados problemas, debido a su complejidad y a los costos respecto a los recursos del sistema.
También es interesante ver cómo las estructuras elegidas impactan en la complejidad temporal de los distintos algoritmos utilizados.

Informe Trabajo Práctico Especial
Programación 3 2019 segunda parte

TUDAI - FCE - UNICEN

Integrantes:

Abate, Juan Manuel juan.abate@gmail.com
Nava, Maximo Santiago maximonava94@gmail.com

Repositorio código fuente: https://github.com/JuanManuelAbate/TUDAI-prog3-tpe

Introducción al problema:

Se desea realizar un recorrido partiendo desde un aeropuerto dado y terminando en el mismo visitando solamente una vez cada uno de los aeropuertos de la forma más optima posible, considerándose para el caso óptimo la menor cantidad posible de kilómetros totales.

Analisis del problema:

El problema planteado es el clásico problema del viajante, las dos alternativas que planteamos son:

Solución 1: esta solución resuelve el problema utilizando una estrategia greedy comenzando desde el aeropuerto de origen y en cada paso visitando el siguiente aeropuerto más cercano hasta que ya no existan aeropuertos por visitar y por último volviendo al primero. Esta solución no da una solución óptima para todos los casos.

Solución 2: esta solución resuelve el problema utilizando una estrategia backtracking comenzando desde el aeropuerto de origen y realizando todas las combinaciones posibles para volver a llegar al aeropuerto origen. Cada vez que se encuentra una solución se analiza si es la mejor y luego de toda la ejecución se obtiene la solución óptima.

Para mayor practicidad a la hora de probar los algoritmos creamos un grafo como se muestra a continuación:

Aeropuerto	**1**	**2**	**3**	**4**	**5**	**6**
1	-	3	10	11	7	25
2	3	-	8	12	9	26
3	10	8	-	9	4	20
4	11	12	9	-	5	15
5	7	9	4	5	-	18
6	25	26	20	15	18	-

Siendo:

1 = Ministro Pistarini
2 = Comodoro Benitez
3 = John F. Kennedy
4 = Jorge Chavez
5 = Campinas
6 = Humberto Delgado

Los archivos para generar el grafo antes mostrado se encuentran en la carpeta segunda parte del repositorio, para ejecutar la aplicación seguir los lineamientos detallados en la sección funcionamiento de la aplicación que se encuentra en el informe de la primera entrega utilzando estos archivos.

Para utilizar la nueva funcionalidad se agregan dos nuevos puntos en el menú de la aplicación.

Análisis solución 1:

La ventaja de esta solución es que resuelve el problema en un costo computacional polinomial, por lo cual puede llegar a ser aplicable para estructuras relativamente grandes.
Por otro lado debido a que se busca una estrategia para no caer en una solución de fuerza bruta, se tienen que tomar decisiones las cuales no nos aseguran que lleguemos en todos los casos a la mejor solución.

En cuanto a complejidad temporal partiendo de que tenemos un grafo donde todos los vértices están conectados con todos los vértices deberemos iterar para cada vértice cada uno de sus adyacentes por lo cual:
n = número de vertices.
m = número de arcos.
En el caso que el grafo sea muy conectado el número de arcos puede llegar a ser n - 1, por lo cual por cada vértice se tendrán n - 1 iteraciones, con lo cual se llega a una complejidad de O(n^n)

Para la implementación utilizamos la clase List provista por java y creamos un objeto llamado AeropuertoKM donde vamos almacenando el nombre del aeropuerto siguiente a visitar y la distancia al mismo, por lo cual obtenemos una lista con el camino a seguir para resolver el problema.

Análisis de la salida:

```
Ingresar aeropuerto de origen
Ministro Pistarini

Añade el primer aeropuerto a la solucion con distancia 0
Solucion actual:
Ministro Pistarini -> 0.0

Chequea que queden aeropuertos por visitar
Busca el siguiente aeropuerto con menor distancia no visitado en este caso: Comodoro Benitez
Añade el aeropuerto Comodoro Benitez a la solucion con distancia 3.0
Solucion actual:
Ministro Pistarini -> 0.0
Comodoro Benitez -> 3.0

Chequea que queden aeropuertos por visitar
Busca el siguiente aeropuerto con menor distancia no visitado en este caso: John F. Kennedy
Añade el aeropuerto John F. Kennedy a la solucion con distancia 8.0
Solucion actual:
Ministro Pistarini -> 0.0
Comodoro Benitez -> 3.0
John F. Kennedy -> 8.0

Chequea que queden aeropuertos por visitar
Busca el siguiente aeropuerto con menor distancia no visitado en este caso: Campinas
Añade el aeropuerto Campinas a la solucion con distancia 4.0
Solucion actual:
Ministro Pistarini -> 0.0
Comodoro Benitez -> 3.0
John F. Kennedy -> 8.0
Campinas -> 4.0

Chequea que queden aeropuertos por visitar
Busca el siguiente aeropuerto con menor distancia no visitado en este caso: Jorge Chavez
Añade el aeropuerto Jorge Chavez a la solucion con distancia 5.0
Solucion actual:
Ministro Pistarini -> 0.0
Comodoro Benitez -> 3.0
John F. Kennedy -> 8.0
Campinas -> 4.0
Jorge Chavez -> 5.0
```

```
Chequea que queden aeropuertos por visitar
Busca el siguiente aeropuerto con menor distancia no visitado en este caso: Humberto Delgado
Añade el aeropuerto Humberto Delgado a la solucion con distancia 15.0
Solucion actual:
Ministro Pistarini -> 0.0
Comodoro Benitez -> 3.0
John F. Kennedy -> 8.0
Campinas -> 4.0
Jorge Chavez -> 5.0
Humberto Delgado -> 15.0

Chequea que queden aeropuertos por visitar
No quedan mas aeropuertos por visitar
Añade el primer aeropuerto a la solucion con distancia 25.0
Solucion actual:
Ministro Pistarini -> 0.0
Comodoro Benitez -> 3.0
John F. Kennedy -> 8.0
Campinas -> 4.0
Jorge Chavez -> 5.0
Humberto Delgado -> 15.0
Ministro Pistarini -> 25.0
```

Como se puede ver en las imágenes el algoritmo itera por cada una de los vértices partiendo desde el provisto, y en cada iteración va buscando el siguiente aeropuerto más cercano al que se encuentra en ese estado y lo agrega a la solución mostrando la distancia para ir del actual al mismo, una vez que no quedan más aeropuertos por visitar agrega a la solución la distancia desde ir al último visitado hacia el aeropuerto de origen, completando así el ciclo.

En este caso la suma de los kilómetros de la solución es 60, sin embargo la suma más optima para el caso de estudio que estamos utilizando es 58.

Análisis solución 2:

La ventaja de esta solución es que siempre llega a la solución óptima del problema, pero por otro lado para lograr esto se requiere un costo computacional muy alto por lo cual esta solución rápidamente se hace inviable e impractica.

En cuanto a la complejidad temporal partiendo de que tenemos un grafo donde todos los vértices están conectados con todos los vértices lo que hace el algoritmo es crear todas las posibles combinaciones para llegar desde el origen al mismo visitando todos. El algoritmo va procesando las distintas soluciones obtenidas y guardando la mejor, por lo cual en última instancia tendremos la solución óptima para el problema.
Teniendo en cuenta esto siendo:
n = número de vertices.
En el caso que todos estén conectados con todos, se realizará un llamado recursivo por cada vértice a cada uno de sus vértices adyacentes no visitados por lo cual la complejidad temporal será de O(!n).

Para la implementación utilizamos la clase List provista por java y creamos un objeto llamado AeropuertoKM donde vamos almacenando el nombre del aeropuerto siguiente a visitar y la distancia al mismo, se utilizan dos listas, una donde se guardara la solución definitiva y optima y otra donde vamos guardando soluciones candidatas a ser la mejor.

Análisis de la salida:

A continuación se muestran fragmentos de la salida por pantalla del algoritmo, ya que la salida es muy grande dado que para el caso de prueba se tienen 120 estados finales.

```
Ingresar aeropuerto de origen
Ministro Pistarini
Me paro en el vertice: Ministro Pistarini
Chequea que queden aeropuertos por visitar
Recorro los adyacentes no visitados de: Ministro Pistarini
Agrego Humberto Delgado->25.0 a la respuesta parcial
Solucion parcial:
Humberto Delgado -> 25.0

Me paro en el vertice: Humberto Delgado
Chequea que queden aeropuertos por visitar
Recorro los adyacentes no visitados de: Humberto Delgado
Agrego Campinas->18.0 a la respuesta parcial
Solucion parcial:
Humberto Delgado -> 25.0
Campinas -> 18.0

Me paro en el vertice: Campinas
Chequea que queden aeropuertos por visitar
Recorro los adyacentes no visitados de: Campinas
Agrego Jorge Chavez->5.0 a la respuesta parcial
Solucion parcial:
Humberto Delgado -> 25.0
Campinas -> 18.0
Jorge Chavez -> 5.0

Me paro en el vertice: Jorge Chavez
Chequea que queden aeropuertos por visitar
Recorro los adyacentes no visitados de: Jorge Chavez
Agrego John F. Kennedy->9.0 a la respuesta parcial
Solucion parcial:
Humberto Delgado -> 25.0
Campinas -> 18.0
Jorge Chavez -> 5.0
John F. Kennedy -> 9.0

Me paro en el vertice: John F. Kennedy
Chequea que queden aeropuertos por visitar
Recorro los adyacentes no visitados de: John F. Kennedy
Agrego Comodoro Benitez->8.0 a la respuesta parcial
Solucion parcial:
Humberto Delgado -> 25.0
Campinas -> 18.0
Jorge Chavez -> 5.0
John F. Kennedy -> 9.0
Comodoro Benitez -> 8.0
```

En la imagen anterior se puede ver como en primera instancia el algoritmo se para en el vértice origen y llama a la recursión con el primero de sus adyacentes, luego se puede ver como este procedimiento se repite recursivamente sobre los otros vértices.

```
Me paro en el vertice: Comodoro Benitez
Chequea que queden aeropuertos por visitar
*Se llego a estado final
Solucion parcial:
Humberto Delgado -> 25.0
Campinas -> 18.0
Jorge Chavez -> 5.0
John F. Kennedy -> 9.0
Comodoro Benitez -> 8.0
Ministro Pistarini -> 3.0

Se encontro una mejor solucion
Solucion final:
Ministro Pistarini -> 0.0
Humberto Delgado -> 25.0
Campinas -> 18.0
Jorge Chavez -> 5.0
John F. Kennedy -> 9.0
Comodoro Benitez -> 8.0
Ministro Pistarini -> 3.0

Agrego Comodoro Benitez->12.0 a la respuesta parcial
Solucion parcial:
Humberto Delgado -> 25.0
Campinas -> 18.0
Jorge Chavez -> 5.0
Comodoro Benitez -> 12.0

Me paro en el vertice: Comodoro Benitez
Chequea que queden aeropuertos por visitar
Recorro los adyacentes no visitados de: Comodoro Benitez
Agrego John F. Kennedy->8.0 a la respuesta parcial
Solucion parcial:
Humberto Delgado -> 25.0
Campinas -> 18.0
Jorge Chavez -> 5.0
Comodoro Benitez -> 12.0
John F. Kennedy -> 8.0
```

En la imagen anterior se ven dos cosas, la primera es como en un punto se llega a un estado final donde ya no quedan vértices por visitar y se compara la solución parcial con la definitiva, en este caso la parcial es mejor que la definitiva, por lo cual esta pasa a ser la posible mejor solución hasta el momento, la sumatoria de la solución es 68.

Además se puede ver como el algoritmo luego comienza a volver de la recursión, se puede ver como en un principio agregó Jhon F. Kennedy hacia Comodoro Benitez y luego al volver agrega Comodoro Benitez hacia Jhon F. Kennedy.

```
Me paro en el vertice: John F. Kennedy
Chequea que queden aeropuertos por visitar
*Se llego a estado final
Solucion parcial:
Humberto Delgado -> 25.0
Campinas -> 18.0
Jorge Chavez -> 5.0
Comodoro Benitez -> 12.0
John F. Kennedy -> 8.0
Ministro Pistarini -> 10.0

Solucion final:
Ministro Pistarini -> 0.0
Humberto Delgado -> 25.0
Campinas -> 18.0
Jorge Chavez -> 5.0
John F. Kennedy -> 9.0
Comodoro Benitez -> 8.0
Ministro Pistarini -> 3.0
```

En esta imagen se puede ver como llega a otro estado final, pero no cambia la solución ya que la parcial tiene una suma mayor que la final que se tiene hasta el momento, en la imagen se puede ver que la suma de parcial es 78 y la de final es 68.

```
Me paro en el vertice: Humberto Delgado
Chequea que queden aeropuertos por visitar
*Se llego a estado final
Solucion parcial:
Comodoro Benitez -> 3.0
John F. Kennedy -> 8.0
Jorge Chavez -> 9.0
Campinas -> 5.0
Humberto Delgado -> 18.0
Ministro Pistarini -> 25.0

Solucion final:
Ministro Pistarini -> 0.0
Campinas -> 7.0
Jorge Chavez -> 5.0
Humberto Delgado -> 15.0
John F. Kennedy -> 20.0
Comodoro Benitez -> 8.0
Ministro Pistarini -> 3.0
```

Por último en la imagen anterior se ve la última salida por pantalla del algoritmo, llegando al último estado de todos los posibles finales, y en el mismo ya se ve que la solución final es la óptima la cual posee un valor de 58.

Bibliografia

Nos basamos en las filminas brindadas por la cátedra, específicamente en las filminas:

Clase 4 - algoritmos greedy
Clase 5 - backtraking

Conclusiones

Nos resultó muy interesante ver cómo un mismo problema puede ser resuelto con diferentes técnicas y cómo de alguna forma varios factores como el dominio de la aplicación, las decisiones de implementación y las limitaciones técnicas se tocan de alguna forma y son puestas en juego para dar una solución final a una situación planteada.