**Trabajo Práctico Especial**

**Programación III**

**Parte2**

**Integrante: Mateo Albert**

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

**Carrera: TUDAI**

**Introducción al problema**

Luego de leer el enunciado podemos identificar que el objetivo del mismo es, con la estructura que estábamos utilizando anteriormente, lograr crear dos nuevos servicios que permitan recorrer todos los aeropuertos pasando solo una vez por ellos y en la menor cantidad de kilometros posibles. Para ello no fue necesario realizar ningún cambio en el modelado del aeropuerto.

**Análisis del problema planteado**

Para poder resolver el problema planteado decidí utilizar en ambos algoritmos un sistema que, a media que se recorren los aeropuertos, estos se marcan como visitados y se agregan a un vector de aeropuertos visitados.

En el caso de Backtracking, debido a que este algoritmo me permitir obtener todas las soluciones posibles, decidí ir almacenándolas para luego compararlas y saber cuál solución es la que recorrió menor distancia.

En el caso de Greedy, como estos tipos de algoritmo toman decisiones en función de la información que está disponible en cada momento, no hay forma de comparar cual es la mejor solución, por eso no se puede garantizar que la solución que obtiene sea la más optima.

**Backtracking**

Al momento de utilizar un algoritmo de tipo Backtracking, a nivel de implementación el principal problema que nos encontramos es que es que tiene un tiempo de procesamiento demasiado largo, además de que a nivel de código este se vuelve mucho más fácil de implementar que un algoritmo tipo Greedy.

La principal limitación que tiene este algoritmo, dentro del contexto del problema, es que en caso de el grafo aumente de tamaño, esta solución se haría inviable debido que tardaría demasiado.

Las principales ventajas del Backtracking son: En caso de existir una o más soluciones el Backtracking podrá calcularlas, es de fácil implementación y puede adaptarse a diversos tipos de problemas.

Como mencione anteriormente, en este caso es muy útil y recomendable utilizar Backtracking debido a que nos va a proporcionar la mejor solución posible, pero en caso de el grafo fuera de mayor tamaño este tipo de algoritmo no sería tan útil.

Este tipo de algoritmo nos otorga la mejor solución posible para resolver este problema.

**Greedy**

Los algoritmos de tipo Greedy tienen una implementación un poco más compleja que el Backtracking, y nos permiten obtener una solución mucho más rápido, pero no da la certeza de que esta solución sea la más optima.

La principal limitación que tiene este algoritmo, es que tienen un enfoque muy corto, es decir, que al momento de tomar una decisión no hay forma de saber cuáles serán los efectos que tengan a futuro.

Las ventajas de Greedy son: Las soluciones que retorna están muy cerca las soluciones optimas y tienen un menor tiempo de procesamiento que los algoritmos de tipo Backtracking.

A mi parecer para este problema, yo no encuentro útil utilizar un algoritmo de este tipo debido a que la solución que se necesita debe ser la más optima.

Este tipo de algoritmo nos otorga una solución que posiblemente no sea la mejor para resolver este problema.

**Implementación algoritmo Backtracking**

Para realizar el servicio que consiste en obtener el camino más corto mediante Backtraking realice 4 métodos:

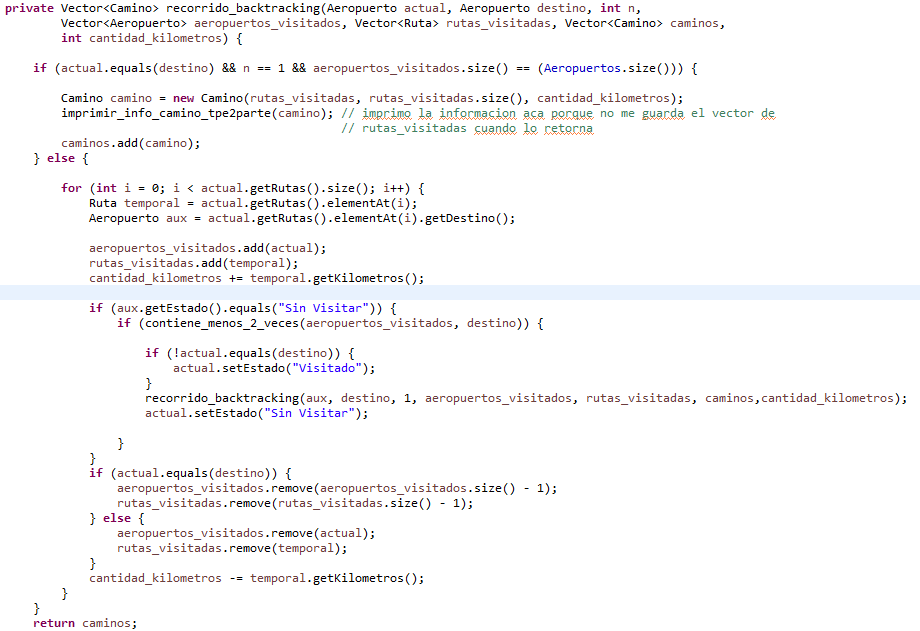
camino\_corto\_backtracking: Mediante este método se pide que se ingrese como parámetro el aeropuerto deseado, y agrega al vector “caminos\_totales” todos los caminos que retorno el metodo recorrido\_backtracking ( ) .

recorrido\_backtracking: Este método se utiliza para realizar el backtracking. La lógica del mismo es que a medida que recorre los aeropuertos los setea como “Visitados” y los agrega a un vector, el cual se verificara si es solución.

contiene\_menos\_2\_veces: La finalidad de este método es corroborar que el vector de aeropuerto visitados, no contenga más de dos veces a el aeropuerto destino, ya que este debe aparecer tanto al principio como al final del vector.

mejor\_camino: Finalmente este método recibe un vector de vectores de la clase camino y busca cual es el que tiene la menor distancia recorrida.

**Seguimiento del algoritmo recorrido\_backtracking**



if (actual.equals(destino) && n == 1 && aeropuertos\_visitados.size() == (Aeropuertos.size())) {

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

imprimir\_info\_camino\_tpe2parte(camino);

caminos.add(camino);

Verifica que el nodo en el que se encuentra sea el de destino, que n sea igual a 1 (esto quiere decir que entro al menos una vez al else), y verifica que el vector de aeropuerto visitados sea de igual tamaño que la cantidad de aeropuertos totales.

En caso de que todo esto se cumpla se crea un nuevo camino y se agrega al vector caminos\_totales.

---------------------------------------------------------------------------------------------

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

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

Aeropuerto aux = actual.getRutas().elementAt(i).getDestino();

aeropuertos\_visitados.add(actual);

rutas\_visitadas.add(temporal);

cantidad\_kilometros += temporal.getKilometros();

Agrego el aeropuerto y la ruta a sus respectivos vectores y cálculo la cantidad de kilómetros actuales.

---------------------------------------------------------------------------------------------

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

if (contiene\_menos\_2\_veces(aeropuertos\_visitados, destino)) {

if (!actual.equals(destino)) {

actual.setEstado("Visitado");

}

recorrido\_backtracking(aux, destino, 1, aeropuertos\_visitados, rutas\_visitadas, caminos,cantidad\_kilometros);

actual.setEstado("Sin Visitar");

Si no visite el nodo siguiente al actual y si mi vector de aeropuertos no contiene más de 2 veces el aeropuerto destino, ingresa al if, y en caso de que el aeropuerto actual no sea el destino lo setea como Visitado, luego hace la recursión y deshace el seteo de estado.

---------------------------------------------------------------------------------------------

if (actual.equals(destino)) {

aeropuertos\_visitados.remove(aeropuertos\_visitados.size() - 1);

rutas\_visitadas.remove(rutas\_visitadas.size() - 1);

} else {

aeropuertos\_visitados.remove(actual);

rutas\_visitadas.remove(temporal);

}

cantidad\_kilometros -= temporal.getKilometros();

}

}

return caminos;

Se deshacen los cambios según el aeropuerto donde se encuentra, debido que como el aeropuerto destino debe aparecer dos veces en el vector (al principio y al final), al utilizar el método “.remove” este me eliminaba el primer aeropuerto seteado en el arreglo y no el ultimo.

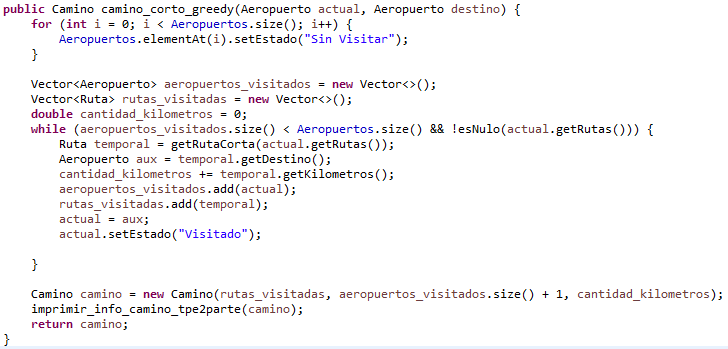
**Implementación algoritmo Greedy**

Para realizar el servicio que consiste en obtener el camino más corto mediante Greedy realice 2 métodos:

camino\_corto\_greedy: Este método se utiliza para realizar el Greedy, a partir del aeropuerto donde se encuentra, busca la ruta más corta y marca el destino de esa ruta como visitado.

getRutaCorta: se pasa como parámetro un vector de rutas y selecciona la de menor distancia.

**Seguimiento del algoritmo camino\_corto\_greedy**



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

Aeropuertos.elementAt(i).setEstado("Sin Visitar");

}

Le seteo a todos los aeropuertos el estado “Sin visitar”.

---------------------------------------------------------------------------------------------

Vector<Aeropuerto> aeropuertos\_visitados = new Vector<>();

Vector<Ruta> rutas\_visitadas = new Vector<>();

double cantidad\_kilometros = 0;

Inicializo variables y vectores.

---------------------------------------------------------------------------------------------

while (aeropuertos\_visitados.size() < Aeropuertos.size() && !esNulo(actual.getRutas())) {

Ruta temporal = getRutaCorta(actual.getRutas());

Aeropuerto aux = temporal.getDestino();

cantidad\_kilometros += temporal.getKilometros();

Mientras que el tamaño del vector de los aeropuertos visitados sea menor que la cantidad de aeropuertos y que el método getRutaCorta() devuelva una ruta, se establecen variables y se aumenta la cantidad de kilómetros actuales.

------------------------------------------------

aeropuertos\_visitados.add(actual);

rutas\_visitadas.add(temporal);

actual = aux;

actual.setEstado("Visitado");

Se agrega el aeropuerto y ruta seleccionada a sus respectivos vectores, se remplaza el valor de la variable actual con el del destino de la ruta seleccionada y se marca como visitado el nuevo actual.

---------------------------------------------------------------------------------------------

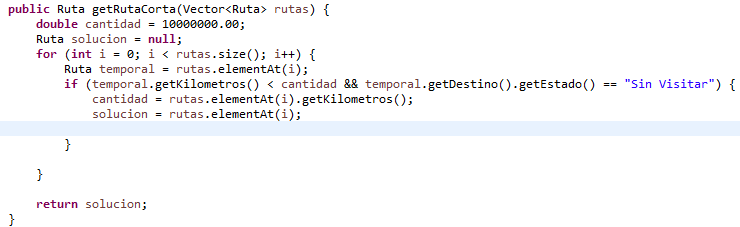
Camino camino = new Camino(rutas\_visitadas, aeropuertos\_visitados.size() + 1, cantidad\_kilometros);

imprimir\_info\_camino\_tpe2parte(camino);

return camino;

Finalmente se crea un nuevo camino con los vectores que se cargaron dentro del “while” y se retorna.

**Seguimiento del algoritmo getRutaCorta**



double cantidad = 10000000.00;

Ruta solucion = null;

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

Se crean variables y se comienza a iterar sobre el vector de rutas pasado como parametro

---------------------------------------------------------------------------------------------

if (temporal.getKilometros() < cantidad && temporal.getDestino().getEstado() == "Sin Visitar"{

cantidad = rutas.elementAt(i).getKilometros();

solucion = rutas.elementAt(i);

Se busca la ruta con menor cantidad de kilómetros y que no haya sido visitada, finalmente se retorna la solucion