**PRÁCTICA 4 – PARTE 2**

**VIAJANTE DE COMERCIO**

**BRANCH & BOUND**

**Alejandro Campoy Nieves**

**David Criado Ramón**

**Nour Eddine El Alaoui**

**Luis Gallego Quero**

**I - Definición del Problema Y enfoque B&B**

Dado un conjunto de ciudades y una matriz con las distancias entre todas ellas, un viajante debe recorrer todas las ciudades exactamente una vez, regresando al punto de partida, de forma tal que la distancia recorrida sea mínima.

Concretamente se trata de hallar el ciclo Hamiltoniano de mínimo peso de ese grafo.

Para emplear un algoritmo de ramificación y poda es necesario utilizar una cota inferior: un valor menor o igual que el verdadero coste de la mejor solución (la de menor coste) que se puede obtener a partir de la solución parcial en la que nos encontremos.

Para realizar la poda, guardamos en todo momento en una variable C el costo de la mejor solución obtenida hasta ahora (que se utiliza como cota superior global: la solución óptima debe tener un coste menor a esta cota). Esa variable puede inicializarse con el costo de la solución obtenida utilizando un algoritmo voraz (como los utilizados en la práctica 3). Si para una solución parcial, su cota inferior es mayor o igual que la cota global (superior) entonces se puede realizar la poda.

Como criterio para seleccionar el siguiente nodo que hay que expandir del árbol de búsqueda (la solución parcial que tratamos de expandir), se emplear el criterio LC o “más prometedor”.

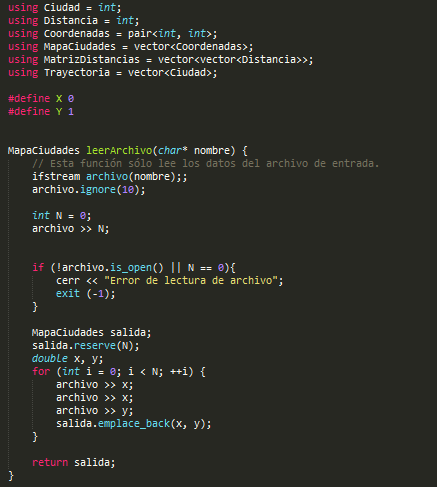
En este caso consideraremos como nodo más prometedor aquel que presente el menor valor de cota inferior.

Para ello se debe de utilizar una cola con prioridad que almacene los nodos ya generados (nodos vivos).

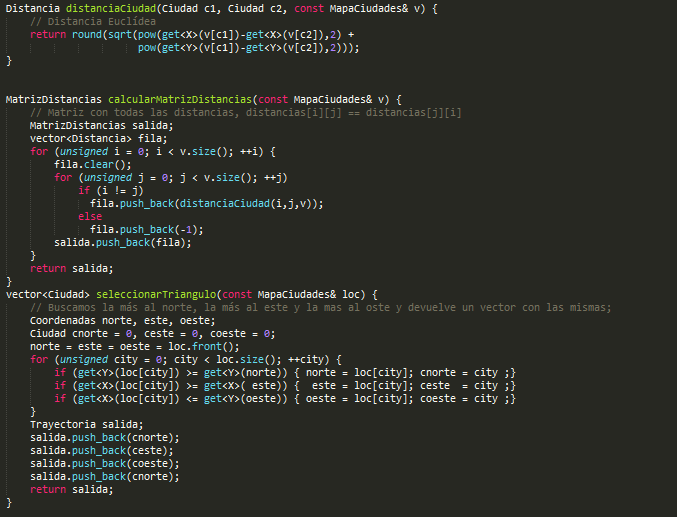
**II - Código**

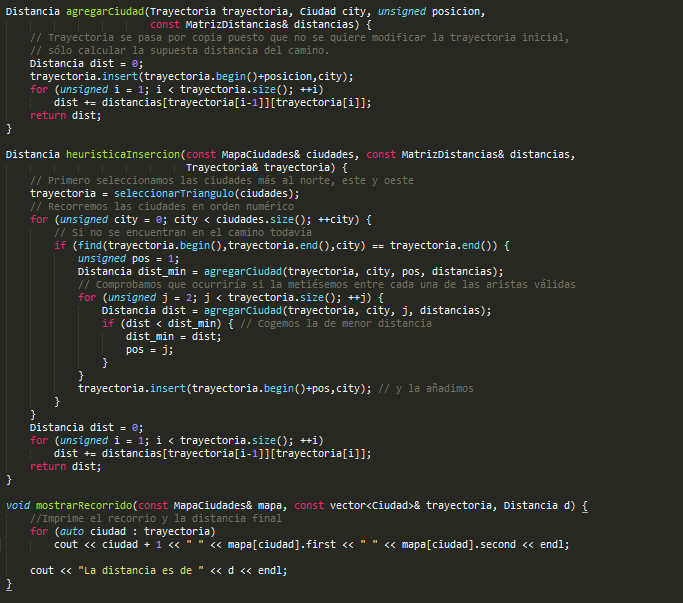
* **Parte de Algoritmos Voraces**

La primera parte del código hace referencia al código utilizado en la práctica 3 (algoritmos voraces) para la resolución del mismo problema utilizando una heurística de inserción a partir de un recorrido parcial.



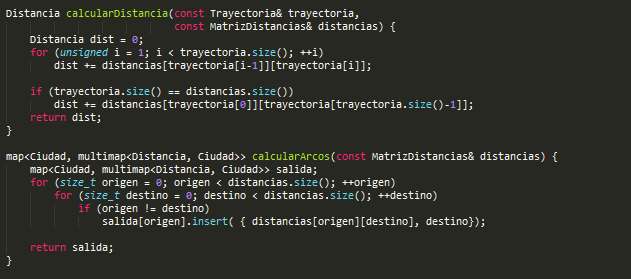
La primera función sirve para leer el archivo recibido como parámetro y convertirlo en un vector de pares de enteros (MapaCiudades) que contendrá para cada ciudad su respectivo par de coordenadas.

Este fragmento de código genera la matriz de distancias utilizada para la resolución del problema dejando a -1 la distancia entre una ciudad consigo misma y contiene la función que genera el recorrido parcial inicial formado por la ciudad más al norte, al este y al oeste.



Este fragmento de código es el que realiza la heurística de inserción, al recorrido parcial obtenido añade la siguiente ciudad en orden ascendente no presente al camino que genera el menor aumento de distancia posible y una función con la que mostrar el recorrido y la distancia por pantalla.

* **Parte de Algoritmos Branch&Bound**



La función **calcularDistancia** dada una trayectoria y la matriz de las distancias entre todas las ciudades devuelve la distancia recorrida en la trayectoria dada.

Si el recorrido no estuviese completo se calcula la distancia entre cada una de esas ciudades. Si está completo se añade la distancia entre la ciudad inicial y la final.

Por otro lado, la función **calculaArcos** generará un multimap ordenado por distancia y cuya descripción sea la ciudad de destino para cada ciudad origen. Esto será fundamental para el cálculo de nuestra cota inferior.



Para la realización del algoritmo Branch&Bound utilizamos una clase solución que contendrá un vector con el camino recorrido actualmente, su correspondiente distancia y su cota inferior.

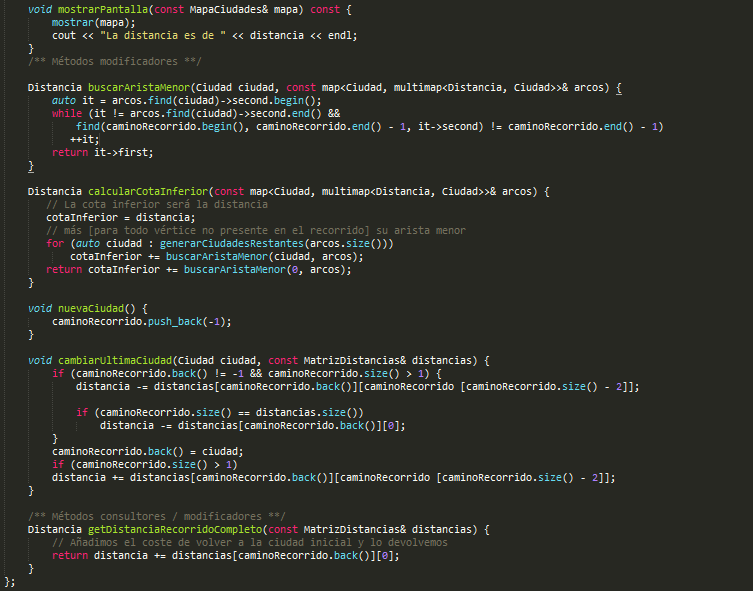
Los métodos **getCotaInferior** y **getCaminoRecorrido** son puros consultores para las variables de mismo nombre.

El método **esSolucion** nos indica si el tamaño del camino recorrido es igual al tamaño dado como parámetro (número de ciudades únicas) indicándonos que hemos encontrado una solución (no tiene por qué ser óptima).

El método **generarCiudadesRestantes** calcula las ciudades por las que todavía no se ha pasado buscando si no se encuentran en el vector caminoRecorrido.

La **sobrecarga del operador <** es utilizada para indicar la prioridad a la hora de sacar los elementos de la cola con prioridad. Es menos prioritario aquel elemento que tenga mayor cotaInferior, ya que es más improbable que dé lugar al recorrido óptimo.

El método **mostrar** permite mostrar el recorrido actual tanto por pantalla como por cualquier otro flujo de salida. El método **mostrarPantalla** además muestra la distancia de dicho recorrido.

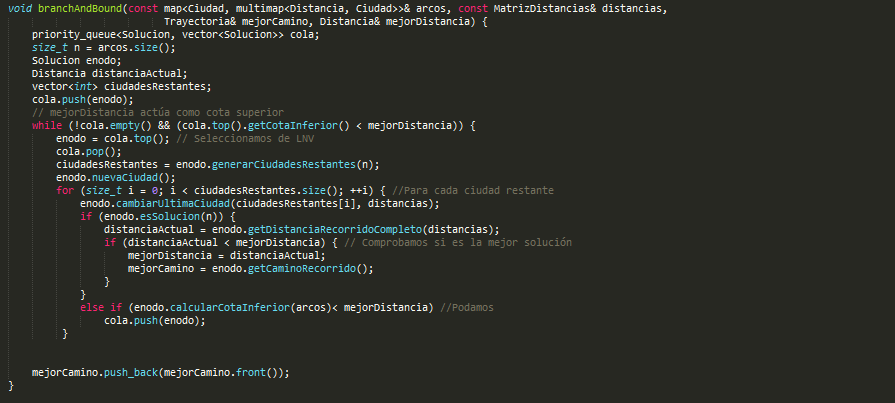


El método **calcularCotaInferior** actualiza el valor de cotaInferior con el valor de la distancia del recorrido actual, más para cada una de las ciudades no presentes en el camino, la arista más chica que parta de dicha ciudad y no vaya a una ciudad del camino que ya esté relacionada con 2 ciudades. Buscar dicha arista es realizado por el método **buscarAristaMenor**.

El método **nuevaCiudad** añade la ciudad nula al final del camino actual.

El método **cambiarUltimaCiudad** cambia la última ciudad por la dada como parámetro actualizando el valor de la distancia en consecuencia a ello. Si sólo hay una ciudad la distancia se considera cero, si hay más de una ciudad se quita la distancia de la última arista del circuito, se modifica la última ciudad y se añade la distancia de la nueva arista. Si estamos con una posible solución también se tiene en cuenta la arista que va de vuelta al primer elemento desde el último para eliminarla antes de modificarla.

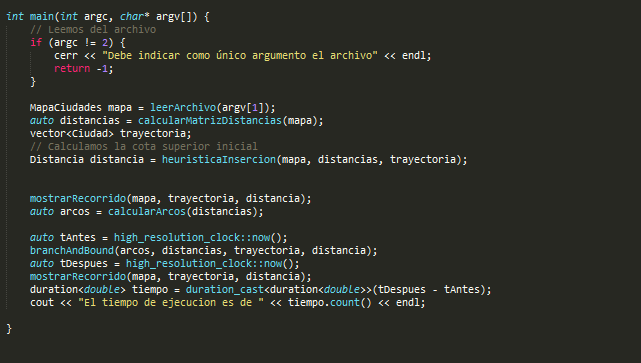
El método **getDistanciaRecorridoCompleto** es únicamente llamado una vez si el recorrido dado es solución y añade la distancia de la arista de la última ciudad a la primera para poder dar la distancia de todo el ciclo.



Este fragmento de código es la parte correspondiente al **branch&bound LC**

Con una cola de prioridad de objetos de la clase Solución, y empezando por el primer nodo vamos sacando elementos de la cola hasta que la cota inferior de dicho elemento sea menor que la cota superior (la cota superior es la mejor distancia obtenida hasta el momento y es inicializado con el valor obtenido por la heurística greedy de inserción). Si sacamos un elemento que está dentro de los límites establecidos por las cotas, generamos un nuevo nivel y calculamos las ciudades que restan por formar parte del camino, una vez eso ocurre iteramos sobre dichas ciudades para cambiar la última ciudad, de tal manera que si son solución, comprobamos si es mejor que la “mejor solución actual” actualizando dichos valores o si no, si la nueva cota inferior tras los cambios sigue siendo menor que la cota superior lo volvemos a meter en la cola con prioridad.

Una vez el proceso ha finalizado insertamos al final del camino la primera ciudad para completar el ciclo.



El main se encarga de ir llamando a las funciones básicas de la práctica tales como leer la trayectoria, llamar a la función greedy y llamar la función branchAndBound midiendo el tiempo de esta última.

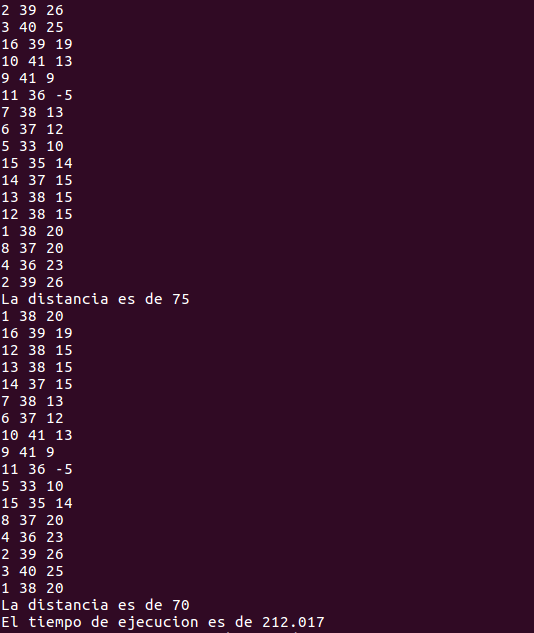
**III - Ejemplo de ejecución y gráfica obtenida**

Para el ejemplo propuesto y como se puede ver en el siguiente ejemplo de ejecución, habiéndose ejecutado con el siguiente PC :

Intel Core i3-3217U 1.80 GHz 4 GB RAM

Ubuntu 16.04 LTS con g++ -O3 -std=c++11

* El algoritmo B&B tarda 217.07 segundos para **ulysses16**



* La gráfica obtenida para el recorrido óptimo es la siguiente:

