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.

```cpp
using Ciudad = int;
using Distancia = int;
using Coordenadas = pair<int, int>;
using MapaCiudades = vector<Coordenadas>;
using MatrizDistancias = vector<vector<Distancia>>;
using Trayectoria = vector<Ciudad>;

#define X 0
#define Y 1


MapaCiudades leerArchivo(char* nombre) {
    // Esta función sólo lee los datos del archivo de entrada.
    ifstream archivo(nombre);;
    archivo.ignore(10);

    int N = 0;
    archivo >> N;


    if (!archivo.is_open() || N == 0){
        cerr << "Error de lectura de archivo";
        exit (-1);
    }

    MapaCiudades salida;
    salida.reserve(N);
    double x, y;
    for (int i = 0; i < N; ++i) {
        archivo >> x;
        archivo >> x;
        archivo >> y;
        salida.emplace_back(x, y);
    }

    return salida;
}
```

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.

```cpp
Distancia distanciaCiudad(Ciudad c1, Ciudad c2, const MapaCiudades& v) {
    // Distancia Euclídea
    return round(sqrt(pow(get<X>(v[c1])-get<X>(v[c2]),2) +
                      pow(get<Y>(v[c1])-get<Y>(v[c2]),2)));
}


MatrizDistancias calcularMatrizDistancias(const MapaCiudades& v) {
    // Matriz con todas las distancias, distancias[i][j] == distancias[j][i]
    MatrizDistancias salida;
    vector<Distancia> fila;
    for (unsigned i = 0; i < v.size(); ++i) {
        fila.clear();
        for (unsigned j = 0; j < v.size(); ++j)
            if (i != j)
              fila.push_back(distanciaCiudad(i,j,v));
            else
              fila.push_back(-1);
        salida.push_back(fila);
    }
    return salida;
}
vector<Ciudad> seleccionarTriangulo(const MapaCiudades& loc) {
    // Buscamos la más al norte, la más al este y la mas al oste y devuelve un vector con las mismas;
    Coordenadas norte, este, oeste;
    Ciudad cnorte = 0, ceste = 0, coeste = 0;
    norte = este = oeste = loc.front();
    for (unsigned city = 0; city < loc.size(); ++city) {
        if (get<Y>(loc[city]) >= get<Y>(norte)) { norte = loc[city]; cnorte = city ;}
        if (get<X>(loc[city]) >= get<X>( este)) {  este = loc[city]; ceste  = city ;}
        if (get<X>(loc[city]) <= get<Y>(oeste)) { oeste = loc[city]; coeste = city ;}
    }
    Trayectoria salida;
    salida.push_back(cnorte);
    salida.push_back(ceste);
    salida.push_back(coeste);
    salida.push_back(cnorte);
    return salida;
}
```

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.

```
Distancia agregarCiudad(Trayectoria trayectoria, Ciudad city, unsigned posicion,
                        const MatrizDistancias& distancias) {
    // Trayectoria se pasa por copia puesto que no se quiere modificar la trayectoria inicial,
    // sólo calcular la supuesta distancia del camino.
    Distancia dist = 0;
    trayectoria.insert(trayectoria.begin()+posicion,city);
    for (unsigned i = 1; i < trayectoria.size(); ++i)
        dist += distancias[trayectoria[i-1]][trayectoria[i]];
    return dist;
}

Distancia heuristicaInsercion(const MapaCiudades& ciudades, const MatrizDistancias& distancias,
                              Trayectoria& trayectoria) {
    // Primero seleccionamos las ciudades más al norte, este y oeste
    trayectoria = seleccionarTriangulo(ciudades);
    // Recorremos las ciudades en orden numérico
    for (unsigned city = 0; city < ciudades.size(); ++city) {
        // Si no se encuentran en el camino todavía
        if (find(trayectoria.begin(),trayectoria.end(),city) == trayectoria.end()) {
            unsigned pos = 1;
            Distancia dist_min = agregarCiudad(trayectoria, city, pos, distancias);
            // Comprobamos que ocurriría si la metiésemos entre cada una de las aristas válidas
            for (unsigned j = 2; j < trayectoria.size(); ++j) {
                Distancia dist = agregarCiudad(trayectoria, city, j, distancias);
                if (dist < dist_min) { // Cogemos la de menor distancia
                    dist_min = dist;
                    pos = j;
                }
            }
            trayectoria.insert(trayectoria.begin()+pos,city); // y la añadimos
        }
    }
    Distancia dist = 0;
    for (unsigned i = 1; i < trayectoria.size(); ++i)
        dist += distancias[trayectoria[i-1]][trayectoria[i]];
    return dist;
}

void mostrarRecorrido(const MapaCiudades& mapa, const vector<Ciudad>& trayectoria, Distancia d) {
    //Imprime el recorrio y la distancia final
    for (auto ciudad : trayectoria)
        cout << ciudad + 1 << " " << mapa[ciudad].first << " " << mapa[ciudad].second << endl;

    cout << "La distancia es de " << d << endl;
}
```

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.

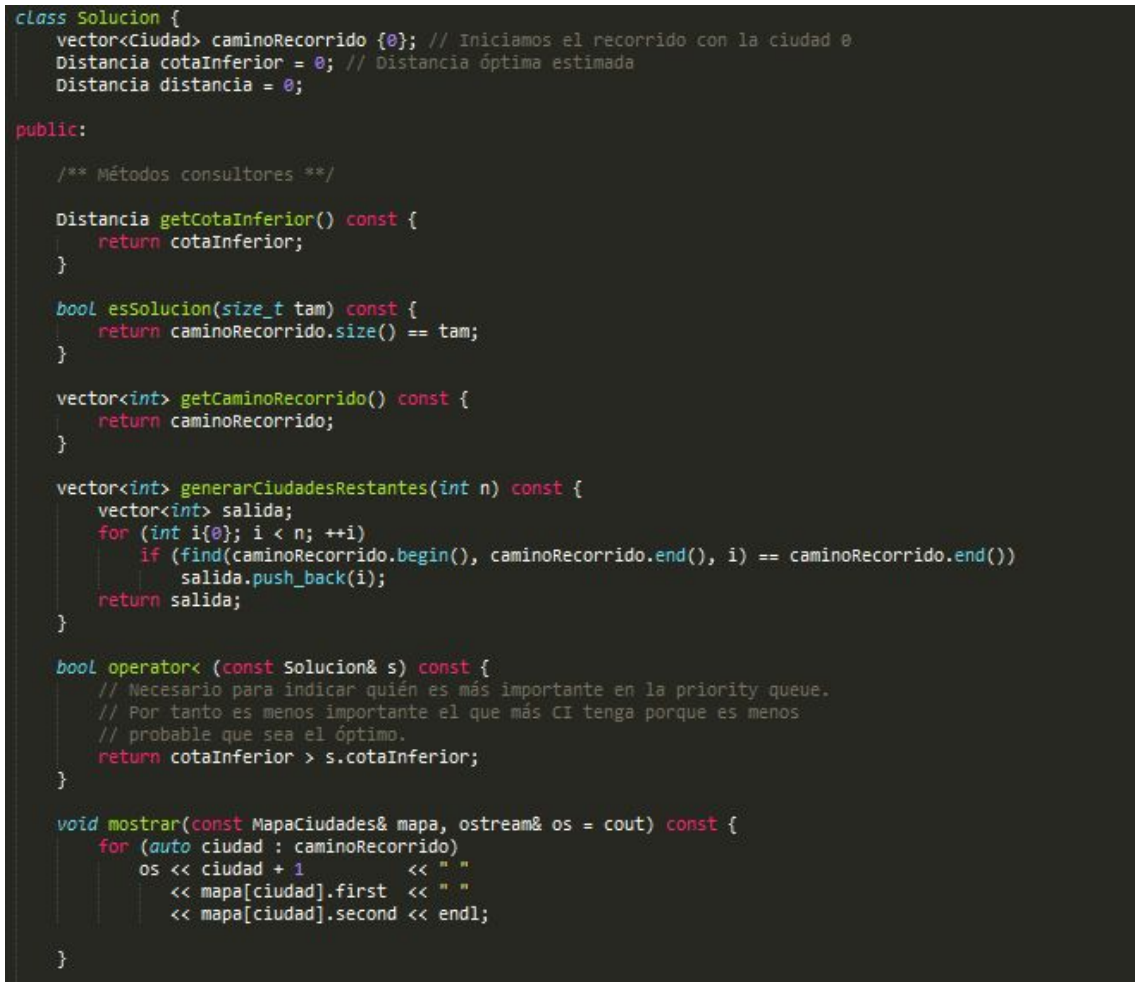
❖ Parte de Algoritmos Branch&Bound

```
Distancia calcularDistancia(const Trayectoria& trayectoria,
                            const MatrizDistancias& distancias) {
    Distancia dist = 0;
    for (unsigned i = 1; i < trayectoria.size(); ++i)
        dist += distancias[trayectoria[i-1]][trayectoria[i]];

    if (trayectoria.size() == distancias.size())
        dist += distancias[trayectoria[0]][trayectoria[trayectoria.size()-1]];
    return dist;
}

map<Ciudad, multimap<Distancia, Ciudad>> calcularArcos(const MatrizDistancias& distancias) {
    map<Ciudad, multimap<Distancia, Ciudad>> salida;
    for (size_t origen = 0; origen < distancias.size(); ++origen)
        for (size_t destino = 0; destino < distancias.size(); ++destino)
            if (origen != destino)
                salida[origen].insert( { distancias[origen][destino], destino});

    return salida;
}
```

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.



```
class Solucion {
    vector<Ciudad> caminoRecorrido {0}; // Iniciamos el recorrido con la ciudad 0
    Distancia cotaInferior = 0; // Distancia óptima estimada
    Distancia distancia = 0;

public:

    /** Métodos consultores **/

    Distancia getCotaInferior() const {
        return cotaInferior;
    }

    bool esSolucion(size_t tam) const {
        return caminoRecorrido.size() == tam;
    }

    vector<int> getCaminoRecorrido() const {
        return caminoRecorrido;
    }

    vector<int> generarCiudadesRestantes(int n) const {
        vector<int> salida;
        for (int i{0}; i < n; ++i)
            if (find(caminoRecorrido.begin(), caminoRecorrido.end(), i) == caminoRecorrido.end())
                salida.push_back(i);
        return salida;
    }

    bool operator< (const Solucion& s) const {
        // Necesario para indicar quién es más importante en la priority queue.
        // Por tanto es menos importante el que más CI tenga porque es menos
        // probable que sea el óptimo.
        return cotaInferior > s.cotaInferior;
    }

    void mostrar(const MapaCiudades& mapa, ostream& os = cout) const {
        for (auto ciudad : caminoRecorrido)
            os << ciudad + 1          << " "
               << mapa[ciudad].first  << " "
               << mapa[ciudad].second << endl;

    }
```

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.

```
    void mostrarPantalla(const MapaCiudades& mapa) const {
        mostrar(mapa);
        cout << "La distancia es de " << distancia << endl;
    }
    /** Métodos modificadores **/

    Distancia buscarAristaMenor(Ciudad ciudad, const map<Ciudad, multimap<Distancia, Ciudad>>& arcos) {
        auto it = arcos.find(ciudad)->second.begin();
        while (it != arcos.find(ciudad)->second.end() &&
            find(caminoRecorrido.begin(), caminoRecorrido.end() - 1, it->second) != caminoRecorrido.end() - 1)
            ++it;
        return it->first;
    }

    Distancia calcularCotaInferior(const map<Ciudad, multimap<Distancia, Ciudad>>& arcos) {
        // La cota inferior será la distancia
        cotaInferior = distancia;
        // más [para todo vértice no presente en el recorrido] su arista menor
        for (auto ciudad : generarCiudadesRestantes(arcos.size()))
            cotaInferior += buscarAristaMenor(ciudad, arcos);
        return cotaInferior += buscarAristaMenor(0, arcos);
    }

    void nuevaCiudad() {
        caminoRecorrido.push_back(-1);
    }

    void cambiarUltimaCiudad(Ciudad ciudad, const MatrizDistancias& distancias) {
        if (caminoRecorrido.back() != -1 && caminoRecorrido.size() > 1) {
            distancia -= distancias[caminoRecorrido.back()][caminoRecorrido [caminoRecorrido.size() - 2]];

            if (caminoRecorrido.size() == distancias.size())
                distancia -= distancias[caminoRecorrido.back()][0];
        }
        caminoRecorrido.back() = ciudad;
        if (caminoRecorrido.size() > 1)
        distancia += distancias[caminoRecorrido.back()][caminoRecorrido [caminoRecorrido.size() - 2]];
    }

    /** Métodos consultores / modificadores **/
    Distancia getDistanciaRecorridoCompleto(const MatrizDistancias& distancias) {
        // Añadimos el coste de volver a la ciudad inicial y lo devolvemos
        return distancia += distancias[caminoRecorrido.back()][0];
    }
};
```

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.

```
void branchAndBound(const map<Ciudad, multimap<Distancia, Ciudad>>& arcos, const MatrizDistancias& distancias,
                    Trayectoria& mejorCamino, Distancia& mejorDistancia) {
    priority_queue<Solucion, vector<Solucion>> cola;
    size_t n = arcos.size();
    Solucion enodo;
    Distancia distanciaActual;
    vector<int> ciudadesRestantes;
    cola.push(enodo);
    // mejorDistancia actúa como cota superior
    while (!cola.empty() && (cola.top().getCotaInferior() < mejorDistancia)) {
        enodo = cola.top(); // Seleccionamos de LNV
        cola.pop();
        ciudadesRestantes = enodo.generarCiudadesRestantes(n);
        enodo.nuevaCiudad();
        for (size_t i = 0; i < ciudadesRestantes.size(); ++i) { //Para cada ciudad restante
            enodo.cambiarUltimaCiudad(ciudadesRestantes[i], distancias);
            if (enodo.esSolucion(n)) {
                distanciaActual = enodo.getDistanciaRecorridoCompleto(distancias);
                if (distanciaActual < mejorDistancia) { // Comprobamos si es la mejor solución
                    mejorDistancia = distanciaActual;
                    mejorCamino = enodo.getCaminoRecorrido();
                }
            }
            else if (enodo.calcularCotaInferior(arcos)< mejorDistancia) //Podamos
                cola.push(enodo);
        }


    mejorCamino.push_back(mejorCamino.front());
}
```

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.

```
int main(int argc, char* argv[]) {
    // Leemos del archivo
    if (argc != 2) {
        cerr << "Debe indicar como único argumento el archivo" << endl;
        return -1;
    }

    MapaCiudades mapa = leerArchivo(argv[1]);
    auto distancias = calcularMatrizDistancias(mapa);
    vector<Ciudad> trayectoria;
    // Calculamos la cota superior inicial
    Distancia distancia = heuristicaInsercion(mapa, distancias, trayectoria);


    mostrarRecorrido(mapa, trayectoria, distancia);
    auto arcos = calcularArcos(distancias);

    auto tAntes = high_resolution_clock::now();
    branchAndBound(arcos, distancias, trayectoria, distancia);
    auto tDespues = high_resolution_clock::now();
    mostrarRecorrido(mapa, trayectoria, distancia);
    duration<double> tiempo = duration_cast<duration<double>>(tDespues - tAntes);
    cout << "El tiempo de ejecucion es de " << tiempo.count() << endl;

}
```

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**

```
2 39 26
3 40 25
16 39 19
10 41 13
9 41 9
11 36 -5
7 38 13
6 37 12
5 33 10
15 35 14
14 37 15
13 38 15
12 38 15
1 38 20
8 37 20
4 36 23
2 39 26
La distancia es de 75
1 38 20
16 39 19
12 38 15
13 38 15
14 37 15
7 38 13
6 37 12
10 41 13
9 41 9
11 36 -5
5 33 10
15 35 14
8 37 20
4 36 23
2 39 26
3 40 25
1 38 20
La distancia es de 70
El tiempo de ejecucion es de 212.017
```

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