UNIVERSIDAD DE GRANADA

E.T.S. DE INGENIERÍAS INFORMATICA y DE TELECOMUNICACIÓN

Departamento de Ciencias de la Computación e Inteligencia Artificial

Práctica 4: Problema de viajante de comercio

grupo b4

Alejandro Casado Quijada

Andrés Ortiz Corrales

Antonio Jiménez Martínez

Jesús Prieto López

Salvador Rueda Molina

Curso 2013-2014

Grado en Informática

1. Objetivo

El objetivo de esta práctica es construir un programa que utilice la técnica de ramificación y acotación para resolver el problema del viajante de comercio.

2. Viajante del comercio.

El problema del viajante de comercio ya se ha comentado y utilizado en la práctica sobre algoritmos voraces.
Si se desea encontrar una solución óptima es necesario utilizar métodos más potentes (y costosos), como la vuelta atrás y la ramificación y poda, que exploran el espacio de posibles soluciones de forma más exhaustiva. De esta manera utilizamos la técnica de Branch and Bound, en un problema de minimizar.

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 practica 2). 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).

3. Algoritmo

Para resolver el problema hemos utilizado la técnica de Branch and Bound que es una generalización de Backtracking con la diferencia en la forma de analizar el árbol, ya que B&B realiza un búsqueda en anchura.

B&B va incluyendo en su lista de nodos vivos aquellos cuyo estimador (cota inferior o local) sea menor que la cota global.

En nuestro algoritmo hemos escogido como cota:

-Cota local o inferior: el estimador que es la suma de la distancia hasta el momento, más las menores aristas de las ciudades que quedan y la menor arista de la ciudad 1(la ciudad a la que regresamos).

- cota global o superior: un greedy con el criterio de búsqueda de la ciudad más cercana que nos ayuda a encontrar una solución bastante ajustada a la final por lo que podremos podar más.

Se podará cuando la cota local se mayor o igual a la cota global ya que nos nos interesa analizar un camino que nos llevará a una solución mayor. Así guardaremos en la cola con prioridad un nodo si su estimador es menor que la cota superior.

Hay que comentar que comenzamos en la ciudad 1. Así esta será la ciudad de inicio y fin.

3.1. Código de los algoritmos

– Algoritmo de poda y ramificacion

```
Solucion Branch_and_Bound(const vector< vector<int> > & matriz,const vector<int> &
arista_menor,int tama)
{
    vector<int> aux;
    priority_queue<Solucion> Q;
    Solucion n_e(tama), mejor_solucion(tama) ; //nodoe en expansion
    mejor_solucion.greedy(matriz);
    int CG=mejor_solucion.getDistancia(); // Cota Global
    int distancia_actual=0;
    Q.push(n_e);
    while ( !Q.empty() && (Q.top().CotaLocal() < CG) ) {
        n_e = Q.top();
        Q.pop();
        aux=n_e.resto_ciudades();
        for(int i=0; i<aux.size(); i++) {
            n_e.anadirciudad(aux[i],matriz);//añadimos ciudad y le sumamos distancia
            n_e.quitarcidudadRestante(aux[i]);//quietamos la ciudad de ciudadrestante
            if ( n_e.EsSolucion(tama) ) {
                distancia_actual = n_e.Evalua(matriz);
                if (distancia_actual < CG) {
                    CG = distancia_actual;
                    mejor_solucion = n_e;
                }
            }
            else {
                n_e.calcularCotaLocal(arista_menor);
                if (n_e.CotaLocal()<CG ) {
                    Q.push( n_e );
                }
            }
            n_e.quitarciudad(matriz);//la ultima que se ha añadido
            n_e.anadircidudadRestante(aux[i]);//añadimos la ultma que se quito
        }
    }
    return mejor_solucion;
}
```

- Clase solución

```
class Solucion {
public:
    Solucion(int tama) {
        pos_e=distancia=estimador=0;
        x.push_back(1);//le pasamos la ciduad 1
        ciudadesRestantes.assign(tama-1,0);//-1 ya que la ciudad 1 no se la
metemos
        crearCiudades();
    }

    void crearCiudades() {
        int i=2;//empezamos en la ciudada 2 ya que el un lo metemos
inicialmente.
        for (list<int>::iterator it=ciudadesRestantes.begin();
it !=ciudadesRestantes.end(); ++it) {
            *it=i;
            i++;
        }
    }
```

```
    void greedy(const vector< vector<int> >  & matriz) {
        x=recorrido(matriz,distancia);
        ciudadesRestantes.clear();

    }
    int getDistancia() {
        return distancia;
    }
    void calcularCotaLocal(const vector<int> & arista_menor) { //en arista_menor
en la posicion 0, tenemos la ciudad 1.
        estimador=distancia;
        for (list<int>::iterator it=ciudadesRestantes.begin();
it !=ciudadesRestantes.end(); ++it) {
            estimador+=arista_menor[(*it)-1];//creo que es -1
        }
        //podemos tambn añadirle siempre la arista de la ciudad 1, ya que
tenemos que volver, ->estimador más preciso
        estimador+=arista_menor[0];
    }
    int CotaLocal() const {
        return estimador;
    }
    bool EsSolucion(int tama) {
        return x.size()==tama;
    }
    vector<int> resto_ciudades() {
        vector<int> aux;
        for (list<int>::iterator it=ciudadesRestantes.begin();
it !=ciudadesRestantes.end(); ++it) {
            aux.push_back(*it);
        }
        return aux;
    }
    void anadirciudad(int a,const vector< vector<int> >  &m) {
        //Le volvemos a restar uno ya que la ciduad n, en la matriz es la n-1
        distancia+=distanciaCiudades(x.back(),a,m);//añade la distancia para ir
a la ciudad
        x.push_back(a);
    }
    void quitarciudad(const vector< vector<int> > & m) {
        //le quitamos a la ultima ciduad la que habia
        //Le volvemos a restar uno ya que la ciduad n, en la matriz es la n-1
        distancia-=distanciaCiudades(x[x.size()-2],x.back(),m);//quita la
distancia para ir a la ciudad
        x.pop_back();//quitamos la ultima ciudad añadida
    }
    void quitarcidudadRestante(int a) {
        for (list<int>::iterator it=ciudadesRestantes.begin();
it !=ciudadesRestantes.end(); ++it) {
            if(*it==a) {
                ciudadesRestantes.erase(it);
                break;
            }
        }
    }
    void anadircidudadRestante(int a) {
        ciudadesRestantes.push_back(a);

    }
    int Evalua(const vector< vector<int> > & m) { //devuelve la distnacia y
añade la distancia para ir a la ciudad 1.
        distancia+=distanciaCiudades(x.back(),1,m);
        return distancia;
    }
```

```
    bool operator<( const Solucion & s) const { //ordenados de menos a mayor
        return estimador > s.estimador;
    }

    void mostrar() {
        cout<<" las ciudades son:"<<endl;
        for(int i=0; i<x.size(); i++) cout<<x[i]<<"   ";
        cout<<x[0]<<endl;
        cout<<"la distancia es: "<<distancia<<endl;
    }
    vector<int> devolverSolucion() {
        return x;
    }
private:
    vector<int> x; // Almacenamos la solucion
    int pos_e; // Posición de la última decisión en X
    int distancia;
    int estimador; // Valor del estimador para el nodo X
    list<int> ciudadesRestantes;
```

3. Cálculo del tiempo teórico

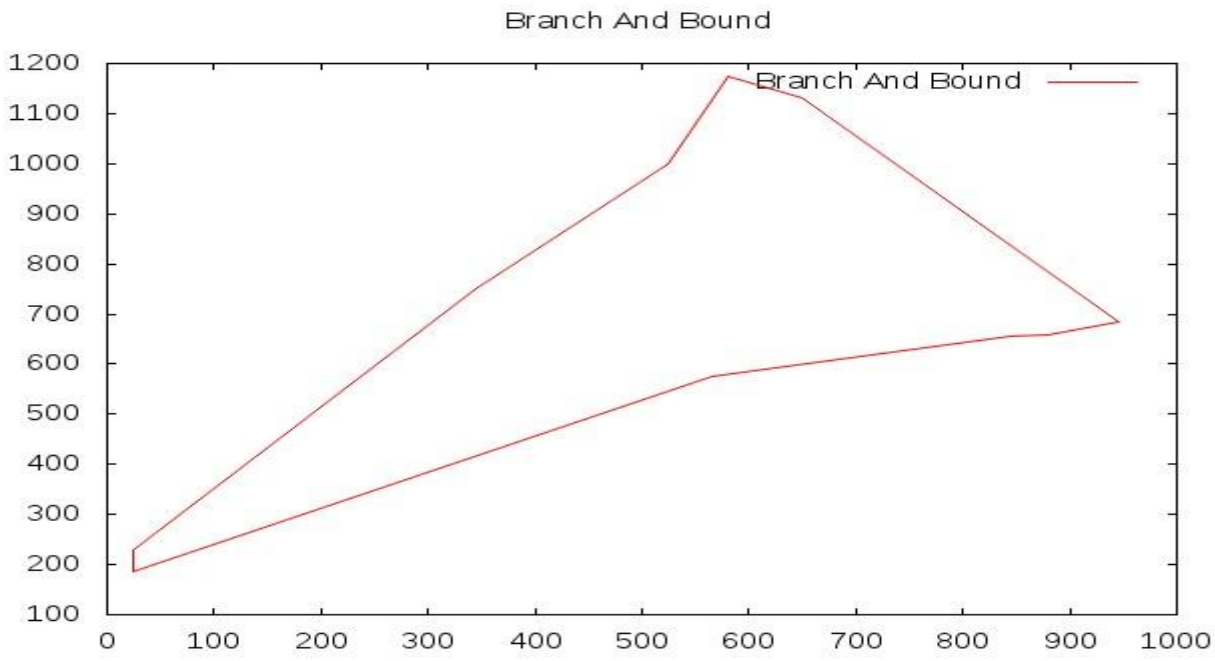
Como hemos comentado anteriormente la técnica Branch and Bound es un algoritmo exhaustivo que analiza todas los nodos que tienen posibilidades de generar un camino al a solución óptima. De esta manera podemos decir que nuestro algoritmo es de orden exponencial.

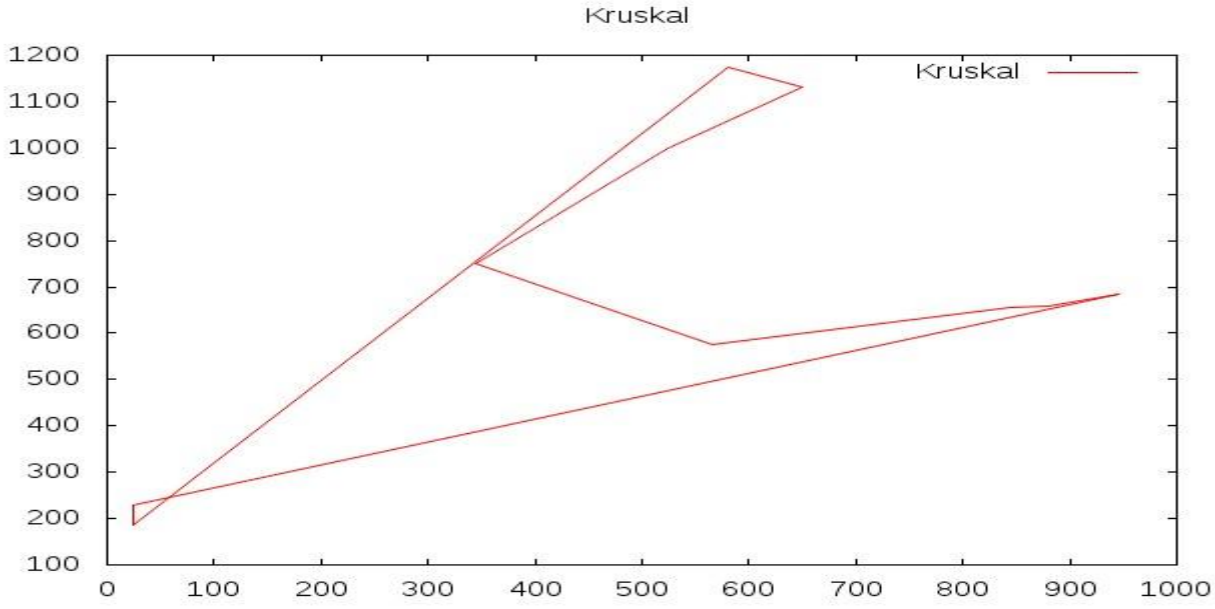
4. Comprobación del algoritmo en la ciudad Berlin52.tsp

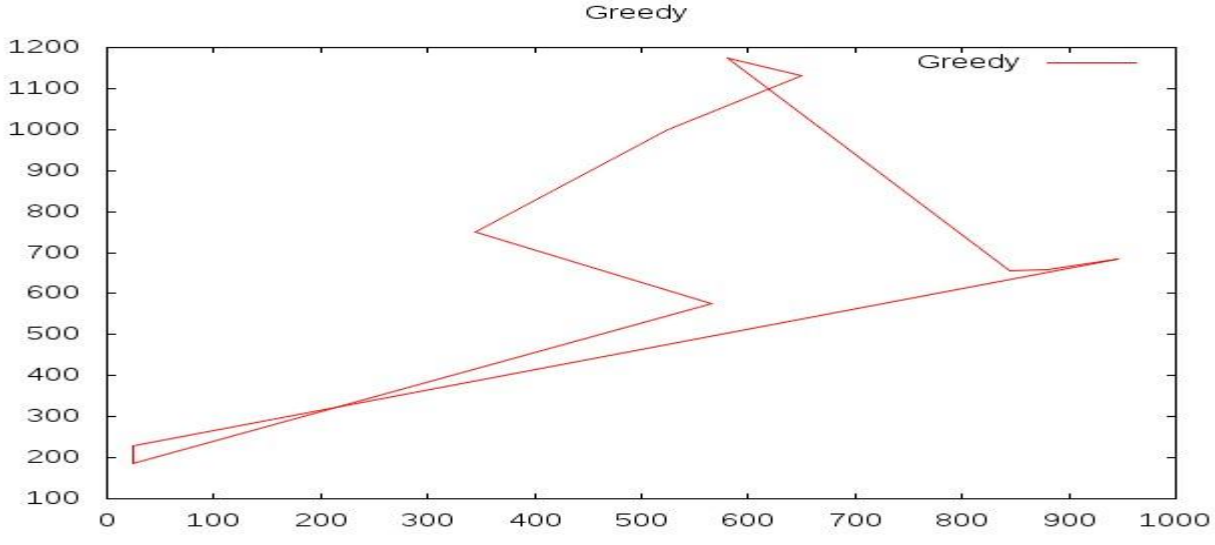
Vamos a trabajar en la ciudad Berlin52.tsp, pero teniendo en cuenta únicamente 10 ciudades, puesto que sino el algoritmo Branch and Bound, generaría desbordamiento de pila, ya que las cotas no son lo suficientes ajustadas. Podemos ver que la distancia más prometedora se encuentra en el algoritmo de ramificación y poda. Lo comparamos con un algoritmo voraz y con el algoritmo de Krukal.
Las distancias para el mismo ejemplo son las siguientes:

B&B: distancia: 2823
Greedy: distancia: 3276
Krukal: distancia: 3452

Los caminos son los siguientes:


Branch And Bound
Branch And Bound
1200
1100
1000
900
800
700
600
500
400
300
200
100
0
100
200
300
400
500
600
700
800
900
1000


Kruskal
Kruskal
1200
1100
1000
900
800
700
600
500
400
300
200
100
0
100
200
300
400
500
600
700
800
900
1000


Greedy
Greedy
1200
1100
1000
900
800
700
600
500
400
300
200
100
0
100
200
300
400
500
600
700
800
900
1000