Práctica 4. Parte 2: El Problema del Viajante de Comercio

Álvaro Fernández García
Alexandra-Diana Dumitru
Ana María Romero Delgado
Germán Castro Montes
Marino Fajardo Torres

Índice

Descripción del Problema

El ejercicio propuesto es el siguiente:

> Dado un conjunto de ciudades y una matriz con todas las distancias entre ellas, un viajante debe recorrer todas las ciudades exactamente una vez, regresando al punto de partida, de forma tal que la distancia sea mínima. Más formalmente, dado un grafo G, conexo y ponderado, se trata de hallar el ciclo hamiltoniano de mínimo peso de dicho grafo.

Este problema ya se ha visto anteriormente con algoritmos voraces, donde se estudiaron métodos de este tipo para encontrar soluciones razonables a este problema, aunque no óptimos necesariamente. Por tanto, el objetivo es encontrar una solución óptima utilizando métodos más potentes y más costosos, como es el caso de la vuelta atras y ramificación y poda, que exploran el espacio de posibles soluciones de forma más exhaustiva.

Algoritmo Branch&Bound

Características principales

- Todos los nodos que se van generando y cumplen con las restricciones del Branch and Bound, se almacenan en una cola con prioridad cuyo criterio para seleccionar el siguiente nodo que hay que expandir del árbol de búsqueda es el Least Cost o “más prometedor”. En este caso consideraremos como nodo más prometedor aquel que presente el menor valor de cota local.
- Para calcular la cota global hemos hecho uso del algoritmo Greedy para el Viajante de Comercio basado en la técnica de los vecinos más cercanos.
- Aparte del tiempo de ejecución, para analizar la eficiencia, calculamos el número de nodos expandidos, el número de podas realizadas y el tamaño máximo de la cola de nodos vivos.

Idea General

Se trata de partir de la primera ciudad, que coincide con la raiz del árbol de exploración, y empezar el proceso de expansión.
Para todas las ciudades restantes no visitadas se genera un hijo que incluye el número de la ciudad, el recorrido realizado hasta el momento, el conjunto de las ciudades que faltan por seleccionar y, lo más importante, la cota local, a la cual nos remitiremos posteriormente.
Si hemos recorrido todas las ciudades, es decir, el nivel en el que estamos corresponde con el numero de ciudades que se quieren explorar (-1), calculamos la distancia acumulada. Si dicha distancia es mejor que la cota global, actualizamos dicho valor y guardamos la solución. En el caso de aun queden ciudades por visitar, se comprueba la condicion de poda y ramificación. Para este proceso, comparamos la cota local del nodo actual con la cota global, mejor

costo encontrado hasta ese momento, y, en el caso de que sea superior realizamos la poda, si no, guardamos el nodo en la cola con prioridad de nodos por explorar.
Mientras no se haya alcanzado la solución, vamos sacando el primer nodo de la cola y repetimos el proceso de expansión.

Elementos del algoritmo

Para la solución al problema del viajante de comercio del algoritmo B&B se ha desarrollado una clase que cuenta con los siguientes atributos:

- **int k:** representa el nivel por el que va el nodo en el árbol;
- **double cota_local:** representa la cota local del nodo. Para calcular la cota local de cada nodo del arbol:

  > – Primero, calculamos la distancia al salir de cada una de las ciudades restantes y nos quedamos con la más pequeña
  > – Finalmente, establecemos el recorrido del nodo en el que estamos añadiendo éste mínimo recorrido para salir de la ciudad elegida a la distancia que llevabamos acumulada hasta ahora.

- **vector < int > recorrido:** representa el circuito solución;
- **vector < int > ciudadesNoElegidas:** representa el conjunto de ciudades aun no visitadas.

A continuación se presentan los métodos necesarios para la realización del algoritmo Branch and Bound.

- **double calculaDistancia(vector Solucion, double * * MD):** calcula el coste total de la solución;
- **void calculaCotaLocal(Solucion & sol, double * * MD, int N):** calcula la cota local de un nodo, mediante el proceso visto anteriormente;
- **Solucion Raiz(int N, double * * MD):** crea la raíz del árbol estableciendo como punto de partida la primera ciudad;
- **Solucion generaHijo(const Solucion & padre, int ciudad, int N, double * * MD):** genera un sucesor de un nodo asociandole la ciudad pasada como parámetro y quitandola del conjunto de ciudades no visitadas. El nivel del hijo es uno mas que el del padre. También calcula la cota local del hijo creado;
- **void ViajanteComercioBandB(int N, double * * MD):** se trata de la función branch and bound para el problema del viajante de comercio, explicada anteriormente en la idea general.

Pseudocódigo

A continuación se desarrolla, en pseudocódigo, el algoritmo principal Branch and Bound:
ViajanteComercioBandB(int N, double * * MD)

```
//Viajante de Comercio B&B
Require: Matriz de Distancias (MD), Cantidad de ciudades (N)

Q {Cola de prioridad}
CG ← VecinoMasCercano(candidatos, MD) {Cota Global}
e_node ← Raiz(N, MD)
Q.push(e_node)

while Q no vacío and Q.top().cota_local < CG do
    e_node ← Q.top()
    Q.pop()
    {incrementar nodos expandidos}
    for all ciudad in e_node.ciudadesNoVisitadas do
        hijo ← generaHijo(e_node, ciudad, N, MD)
        if {hijo tiene recorrido completa} then
            distancia ← calculaDistancia(hijo, MD)
            if distancia < CG then
                CG ← distancia
                mejorSolucion ← hijo
            end if
```

```
        else if hijo.cota_local < CG then
            Q.push(hijo)
        else
            {incrementar número de podas}
        end if
        {Comprobar el tamaño de la cola}
    end for
end while

print mejorSolucion, numero_podas, numero_nodos_expandidos, tamaño_cola, CG
```

Código

Aquí se presenta el código del algoritmo:

```
#include <iostream>
#include <fstream>
#include <map>
#include <cmath>
#include <vector>
#include <queue>
#include <limits>
#include <chrono>

using namespace std;
using namespace std::chrono;

////////////////////////////////////////////////////////////////////////////////
///////////////
//Representa una solución al problema del viajante de comercio:
struct Solucion{
  int k;                            //Representa el nivel por el que va el nodo e
n el árbol.
  double cota_local;                //Representa la cota local del nodo.
  vector<int> recorrido;            //Representa el circuito solución.
  vector<int> ciudadesNoElegidas;   //Representa las ciudades que aún faltan por
elegir.
};

////////////////////////////////////////////////////////////////////////////////
///////////////
//Functor empleado para la priority_queue:
class Comparator{
public:
  bool operator()(const Solucion & s1, const Solucion & s2){
    return s1.cota_local > s2.cota_local;
  }
};

////////////////////////////////////////////////////////////////////////////////
///////////////
//Función para leer los puntos del archivo:
void leer_puntos(string & nombre, map<int,pair<double,double>> & M){
  ifstream datos;
  string s;
  pair<double,double> p;
  int n,act;
```

```
  datos.open(nombre.c_str());
  if(datos.is_open()) {
      datos >> s; // Leo "dimension:"
        datos >> n;
    int i=0;

      while(i<n){
       datos >> act >> p.first >> p.second;
           M[act] = p;
           i++;
    }
      datos.close();
 }
 else{
   cout << "Error de Lectura en " << nombre << endl;
   exit(-1);
 }
}

////////////////////////////////////////////////////////////////////////////////
///////////////
//Función para calcular la distancia euclídea entre dos puntos:
double Distancia(pair<double,double> p1, pair<double,double> p2){
    return sqrt(pow(p2.first-p1.first,2) + pow(p2.second-p1.second,2));
}

////////////////////////////////////////////////////////////////////////////////
///////////////
//Función para calcular la matriz de distancias:
void CalculaMD(double** M, int n, map<int,pair<double,double>> & mapa){
    for(int i = 1; i<n; ++i)
        for(int j = 1; j<n; ++j)
            M[i][j] = Distancia(mapa[i], mapa[j]);
}

////////////////////////////////////////////////////////////////////////////////
///////////////
//Método para calcular el coste de una solución:
double calculaDistancia(vector<int> Solucion, double** MD){
  double coste = 0;
  for(unsigned int i = 0; i < Solucion.size()-1; ++i)
    coste += MD[Solucion[i]][Solucion[i+1]];

  coste += MD[Solucion.front()][Solucion.back()];

  return coste;
}

////////////////////////////////////////////////////////////////////////////////
///////////////
//Sirve para calcular la cota global inicial mediante el vecino más cercano:
double VecinoMasCercano(vector<int> & candidatos, double** MD){

    //Variables para el calculo de la distancia;
    vector<int> S, Stemp, candidatos_tmp;
    double MejorDist = -1.0, distTemp, min_dist;
    int c_actual, c_siguiente;

    //Calcular la distancia partiendo de distintas ciudades:
```

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

        distTemp = 0.0;
        candidatos_tmp = candidatos;
        c_actual = candidatos_tmp[i];
        Stemp.push_back(c_actual); //Añadir a la lista de seleccionados.
        candidatos_tmp.erase(candidatos_tmp.begin()+i); //Borrar de entre los candi
datos la ciudad.

        while(!candidatos_tmp.empty()){

            //Función Selección. El vecino más cercano:
            min_dist = MD[c_actual][candidatos_tmp[0]];
            c_siguiente = 0;
            for(int j = 1; j < candidatos_tmp.size(); ++j)
                if(MD[c_actual][candidatos_tmp[j]] < min_dist){
                    c_siguiente = j;
                    min_dist = MD[c_actual][candidatos_tmp[j]];
                }
            //Las ciudades elegidas son factibles, ya que las ya elegidas no figura
n en candidatos.
            distTemp += min_dist;
            Stemp.push_back(candidatos_tmp[c_siguiente]);
            c_actual = candidatos_tmp[c_siguiente];
            candidatos_tmp.erase(candidatos_tmp.begin()+c_siguiente);
        }

        //Añadir a distTemp la distancia entre la primera y la última ciudad:
        distTemp += MD[Stemp.front()][Stemp.back()];

        //Comprobar si es mejor que la que ya había:
        if(MejorDist == -1.0){
            MejorDist = distTemp;
            S = Stemp;
        }
        else if(distTemp < MejorDist){
            MejorDist = distTemp;
            S = Stemp;
        }
        Stemp.clear(); // Vaciar la solución temporal.
    }

    return MejorDist;
}

////////////////////////////////////////////////////////////////////////////////
///////////////
//Función que calcula la cota local de un nodo:
void calculaCotaLocal(Solucion & sol, double **MD, int N){
  double cota = 0.0;
  //Calcular la distancia que lleva el recorrido:
  for(int i = 0; i<sol.k; ++i)
    cota += MD[sol.recorrido[i]][sol.recorrido[i+1]];

  //Calcular el mínimo recorrido para salir de cada una de las ciudades restantes:
  double min;
  for(int i = 0; i<sol.ciudadesNoElegidas.size(); ++i){
    min = numeric_limits<double>::max();
    for(int j = 1; j<N+1; ++j)
```

```
      if(sol.ciudadesNoElegidas[i] != j && MD[sol.ciudadesNoElegidas[i]][j] < min)
        min = MD[sol.ciudadesNoElegidas[i]][j];
    cota += min;
  }

  //Añadir el mínimo recorrido para salir de la última ciudad elegida:
  min = numeric_limits<double>::max();
  for(int i = 1; i<N+1; ++i)
    if(i != sol.recorrido[sol.k] && MD[i][sol.recorrido[sol.k]] < min)
      min = MD[i][sol.recorrido[sol.k]];
  cota += min;

  sol.cota_local = cota;
}

////////////////////////////////////////////////////////////////////////////////
///////////////
//Función que crea la raiz del árbol: (Actúa como constructor)
Solucion Raiz(int N, double** MD){
  Solucion sal;
  sal.k = 0;
  sal.recorrido.resize(N);
  sal.recorrido[sal.k] = 1; //Empezamos desde la primera ciudad.
  for(int i = 2; i<=N; ++i)
    sal.ciudadesNoElegidas.push_back(i);
  calculaCotaLocal(sal, MD, N);
  return sal;
}

////////////////////////////////////////////////////////////////////////////////
///////////////
//Función que crea el sucesor de un nodo:
Solucion generaHijo(const Solucion & padre, int ciudad, int N, double** MD){
  Solucion hijo = padre;
  hijo.k = padre.k + 1;
  hijo.recorrido[hijo.k] = ciudad;
  bool enc = false;
  for(auto it = hijo.ciudadesNoElegidas.begin(); it != hijo.ciudadesNoElegidas.end(
) && !enc; ++it)
    if(*it == ciudad){
      enc = true;
      hijo.ciudadesNoElegidas.erase(it);
    }
  calculaCotaLocal(hijo, MD, N);
  return hijo;
}

////////////////////////////////////////////////////////////////////////////////
///////////////
//Función B&B para el Viajante de Comercio:
void ViajanteComercioBandB(int N, double** MD, vector<int> & candidatos){
  int ciudad, nodos_exp = 0, num_podas = 0;
  int tam_max = numeric_limits<int>::min();
  double distancia = 0.0;
  Solucion e_node, hijo, mejorSolucion;
  priority_queue<Solucion, vector<Solucion>, Comparator> Q;
  double CG = VecinoMasCercano(candidatos, MD);
  e_node = Raiz(N, MD);
  Q.push(e_node);
```

```
  while(!Q.empty() && (Q.top().cota_local < CG)){
    e_node = Q.top();
    Q.pop();
    nodos_exp++;
    for(int i = 0; i<e_node.ciudadesNoElegidas.size(); ++i){
      ciudad = e_node.ciudadesNoElegidas[i];
      hijo = generaHijo(e_node, ciudad, N, MD);
      if(hijo.k == N-1){
        distancia = calculaDistancia(hijo.recorrido, MD);
        if(distancia < CG){
          CG = distancia;
          mejorSolucion = hijo;
        }
      }
      else if(hijo.cota_local < CG)
        Q.push(hijo);
      else
        num_podas++;
      if((int) Q.size() > tam_max) tam_max = (int) Q.size();
    }
  }
  //Imprimir resultados:
  cout << "Recorrido: ";
  for(int i = 0; i<mejorSolucion.recorrido.size(); ++i)
    cout << "[" << mejorSolucion.recorrido[i] << "]";
  cout << endl;
  cout << "Distancia: " << CG << endl;
  cout << "Nodos expandidos: " << nodos_exp << endl;
  cout << "Podas realizadas: " << num_podas << endl;
  cout << "Tamaño máximo de la cola: " << tam_max << endl;
}
```

MAIN

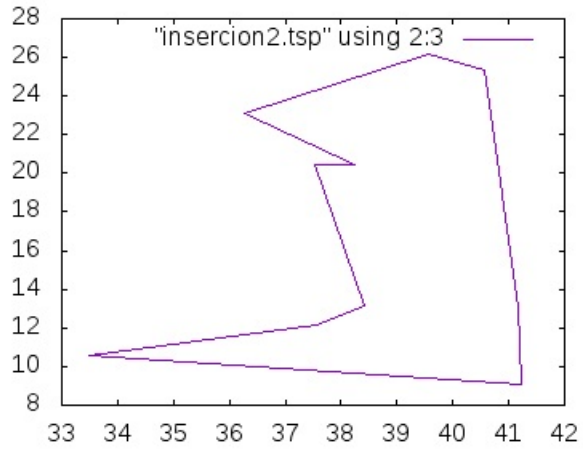
```
int main(int argc, char* argv[]){

    if(argc < 2){
        cout << "Modo de empleo: " << argv[0] << " <archivo.tsp>" << endl;
        exit(-1);
    }

  high_resolution_clock::time_point tantes, tdespues;
  duration<double> duracion;
    string archivo(argv[1]);
    map<int,pair<double,double>> puntos;

    //Leer los puntos:
    leer_puntos(archivo, puntos);
    int N = puntos.size();

    //Reservar matriz de distancia:
    double** D = new double*[N+1];
    for(int i = 0; i<N+1; ++i)
        D[i] = new double[N+1];

    //Calcular matriz de Distancias
```

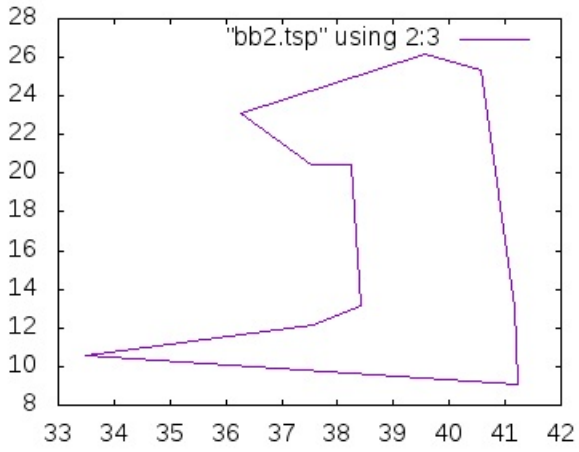
```
    CalculaMD(D, N+1, puntos);

  //Crear el vector de candidatos para el greedy:
    vector<int> cand;
    for(int i = 1; i<=N; ++i)
        cand.push_back(i);

  tantes = high_resolution_clock::now();
  ViajanteComercioBandB(N, D, cand);
  tdespues = high_resolution_clock::now();

  duracion = duration_cast<duration<double>>(tdespues - tantes);
  cout << "Tiempo transcurrido: " << duracion.count() << " segundos" << endl;

  for(int i = 0; i<N+1; ++i)
    delete[] D[i];
  delete[] D;

  exit(0);
}
```

Soluciónes encontradas

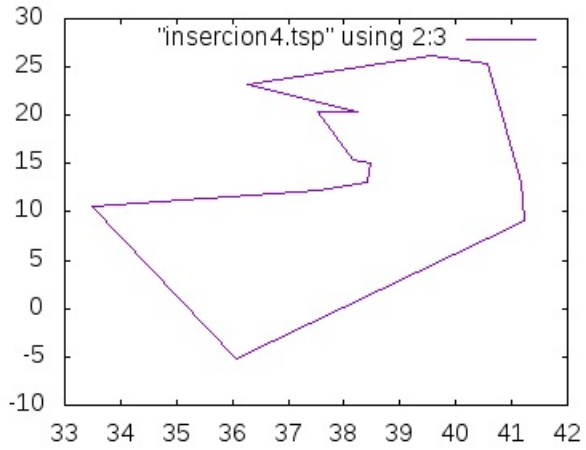
A continuación se muestran las soluciones encontradas por el algoritmo Branch and Bound para circuitos pequeños, así como la comparación de éstas con las soluciones encontradas por el algoritmo Greedy basado en la técnica de inserción.

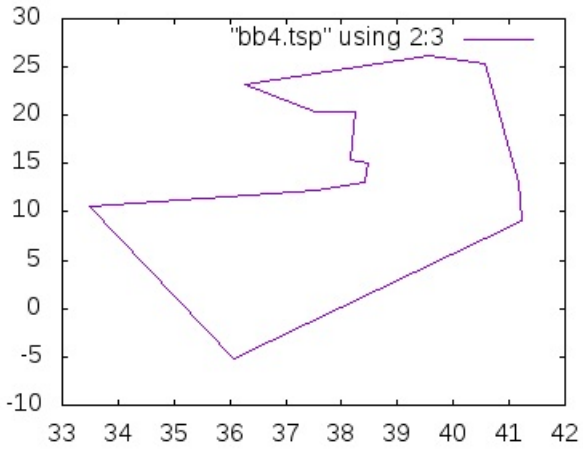
- Circuito de 10 ciudades:





- Circuito de 13 ciudades:





- Circuito de 15 ciudades: