Algorítmica (2018-2019)
Doble Grado en Ingeniería Informática Y Matemáticas
Universidad de Granada

Memoria Práctica 3 Algoritmos voraces (Greedy) Grupo Azul

Irene Huertas González,
Javier Alcántara García,
César Muñoz Reinoso

February 13, 2020

Contents

1 Problema común. TSP

1.1 Enunciado

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 departida, de forma tal que la distancia recorrida sea mínima. Mas formalmente, dado un grafo G, conexo y ponderado, se trata de hallar el ciclo hamiltoniano de mínimo peso de ese grafo.
Nos piden abordar la solución para TSP, la cual consiste en indicar el orden en que se deben recorrer las ciudades sin olvidar cerrar el ciclo para calcular la longitud del ciclo.

A continuación vamos a desarrollar 3 tipos de algoritmos voraces con distintos enfoques.

1.2 Algoritmos

1.2.1 Basado en cercanía

Estrategia basada en el vecino más cercano. Dada una ciudad inicial, se agrega como siguiente aquella que aun no se haya visitado que esté más cercana a esta. Esto se repite hasta visitarlas todas. La distancia entre una ciudad y otra se calcula con la fórmula de la distancia euclídea.

Implementación: Leemos el archivo con las coordenadas de las ciudades y las metemos en una matriz que tiene dos columnas (coordenada x, coordenada y) y tantas filas como ciudades haya.

A continuación creamos otra matriz que contiene las distancias redondeadas al entero más próximo de una ciudad a otra. Se interpretaría de la siguiente forma: el elemento de la posicion (1,3) es la distancia que hay desde la ciudad 1 a la 3, que es igual al elemento que encontramos en la posición (3,1). La diagonal principal de la matriz está compuesta de ceros ya que contiene la distancia entre una ciudad consigo misma.

```
int distancias[dimension][dimension] = {0};
  for (int i = 0; i < dimension; i++){
        for(int j = 0; j < dimension; j++){
          distancias[i][j]=distancia_euclidea(matriz[j][0],matriz[i][0],
          matriz[j][1], matriz[i][1]);
        }
      }
```

Función que calcula la distancia euclídea:

```
int distancia_euclidea(int x2, int x1, int y2, int y1){
  double d = sqrt(((x2-x1)*(x2-x1)) + ((y2-y1)*(y2-y1)));
  return round(d);
```

}

Creamos a parte también un vector que contiene el orden de las ciudades y además nos sirve para saber cuales hemos visitado ya y cuales no.

Comenzamos seleccionando como ciudad inicial la primera que hayamos leído. Nos vamos a la fila, de la matriz de distancias, que se corresponde con el número de la ciudad (por ejemplo, si es la ciudad 1 nos vamos a la primera fila). Recorremos la fila, seleccionamos el menor elemento que indica la distancia más pequeña a otra ciudad y nos fijamos en que columna estamos, ya que nos indica la siguiente ciudad. Miramos que no hayamos estado ya allí, la metemos en el vector de orden y saltamos a la fila cuyo número se corresponde con el de la ciudad seleccionada y volvemos a realizar los mismos pasos hasta recorrerlas todas.
Cuando haya introducido de la lista todas las ciudades sin repeticiones, añadimos a la distancia final el camino entre la última ciudad y la primera, e insertamos de nuevo la ciudad 1 (única que se repite).

```
for(int i=1; i < dimension; i++){
  min=9000;
  for(int j = 0; j < dimension; j++){
    if((distancias[ciudad-1][j] != 0) && (distancias[ciudad-1][j]<min)){
          for(int k=0; k < dimension; k++){
            if (j+1 == orden[k])
              repe = true;
      }
          if(!repe){
        min = distancias[ciudad-1][j];
        ciudad_min = j;
      }
      else
        repe = false;
    }
  }
  orden[i] = ciudad_min+1;
  ciudad = orden[i] -1;
}

orden[dimension]=1; //Cerramos el circulo
```

1.2.2 Basado en mejor inserción

Primero creamos una matriz con la distancias entre las ciudades, utilizando el algoritmo de Euclides. Comenzamos seleccionando la ciudad inicial como la 1. Recorremos la fila 1 y seleccionamos la ciudad a menor distancia (columnas). Cuando hacemos esto saltamos a la fila de la ciudad seleccionada y volvemos a realizar este procedimiento. Tenemos en cuenta que no podemos introducir ciudades repetidas. Cuando haya introducido de la lista todas las ciudades sin repeticiones, añadimos a la distancia final el camino entre la última ciudad y la primera, e insertamos de nuevo la ciudad 1 (única que se repite).

```
list<int> TSP(const vector<vector<int> > &matriz, const map<int,
pair<double, double> > &m, int &distancia_final) {
        int n=1;
        int s=1;
        int w=1;

        list<int> salida;
        map<int,pair<double,double> >::const_iterator it = m.begin();
        list<int> ciudades_restantes;
        ciudades_restantes.push_back(1);

        pair<double,double> pn=(*it).second,
        ps=(*it).second, pw=(*it).second;
        it++;


        for(it; it != m.end(); it++) {
                ciudades_restantes.push_back((*it).first);
                pair<double,double> p=(*it).second;
                int ciudad_actual=(*it).first;

                if(p.first<pw.first) { //Elige la de mas al oeste
                        pw=p;
                        w=ciudad_actual;
                }
                else if(p.second>pn.second || (n==w && n==1)) {
                        pn=p;
                        n=ciudad_actual;
                }
                else if(p.second<ps.second || (s==w && s==1)) {
                        ps=p;
                        s=ciudad_actual;
                }
        }
```

```
        salida.push_back(n);
        salida.push_back(s);
        salida.push_back(w);

        ciudades_restantes.remove(n);
        ciudades_restantes.remove(s);
        ciudades_restantes.remove(w);

        while(ciudades_restantes.size() > 0) {
                pair<int,list<int>::iterator> pres;
                pres = ElegirCiudad(salida, matriz, ciudades_restantes);
                InsertarCiudad(salida, pres.first, pres.second);
        }

        salida.push_back(*(salida.begin()));
        distancia_final = DistanciaTotal(salida, matriz);

        return salida;
}
```

1.2.3 Basado en mejor arista

Creamos un mapa formado por la distancia entre dos ciudades y un par con esas dos ciudades. Vamos seleccionando las distancias menores entre dos ciudades y añadiendo estas al conjunto de seleccionados. Para poder insertar una nueva ciudad, esta no puede tener más de 2 aristas ni crear un ciclo. Cómo el camino resultante de este algoritmo no es cerrado, tenemos que recorrer el camino y añadir la distancia entre las 2 ciudades que únicamente tienen una arista.

```
vector<int> TSP(multimap<int, pair<int,int>>
& aristas_par_ciudades, int & distancia) {
        vector<int> salida;
        list<pair<int,int>> aux;
        multimap<int, pair<int,int>>::iterator it =
        aristas_par_ciudades.begin();
        aux.push_back((*it).second);
        distancia+=(*it).first;
        it++;
        while(it != aristas_par_ciudades.end()) {
                if(CiudadFactible(aux, (*it).second) && !ExistenCiclos(aux,
                (*it).second)) {
                        aux.push_back((*it).second);
                        distancia += (*it).first;
                }
                it++;
        }
        CerrarCiclo(aux, aristas_par_ciudades, distancia);
        salida = TransformaACamino(aux);

        return salida;
}
```

1.3 Comparación de las tres estrategias

1.3.1 Visualización recorridos

Con gnuplot hemos representado el recorrido de las coordenadas del fichero *att48.tsp*. Las ciudades de este fichero están situadas de la siguiente forma:

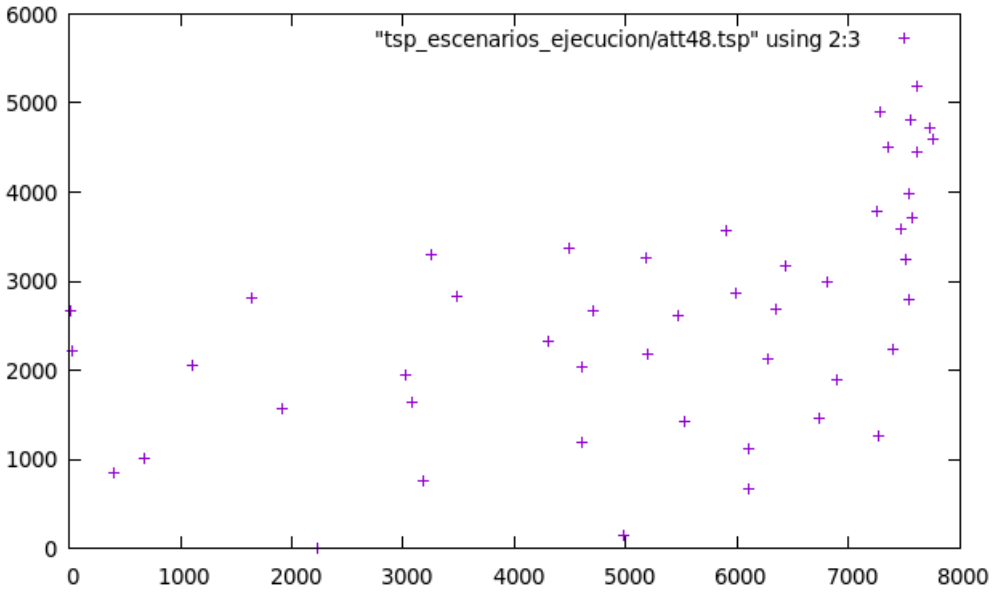

Figure 1.1: Disposición de las ciudades en el plano

Si se recorren las ciudades con el enfoque cercanía el recorrido se queda así habiendo comenzado por la ciudad cuyas coordenadas son(6734,1453) y finaliza en la ciudad (1916,1569) y ya vuelve a la inicial:

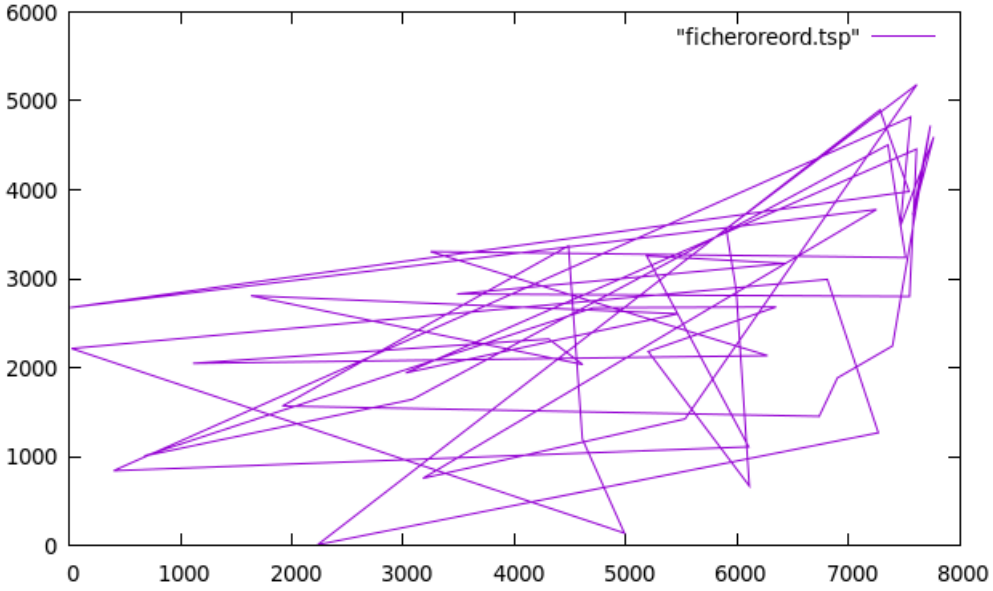

Figure 1.2: Recorrido con el enfoque cercanía

Si se recorren las ciudades con el enfoque de inserción el recorrido se queda así habiendo comenzado por la ciudad cuyas coordenadas son(1633,2809) y finaliza en la ciudad (10, 2676) y ya vuelve a la inicial:

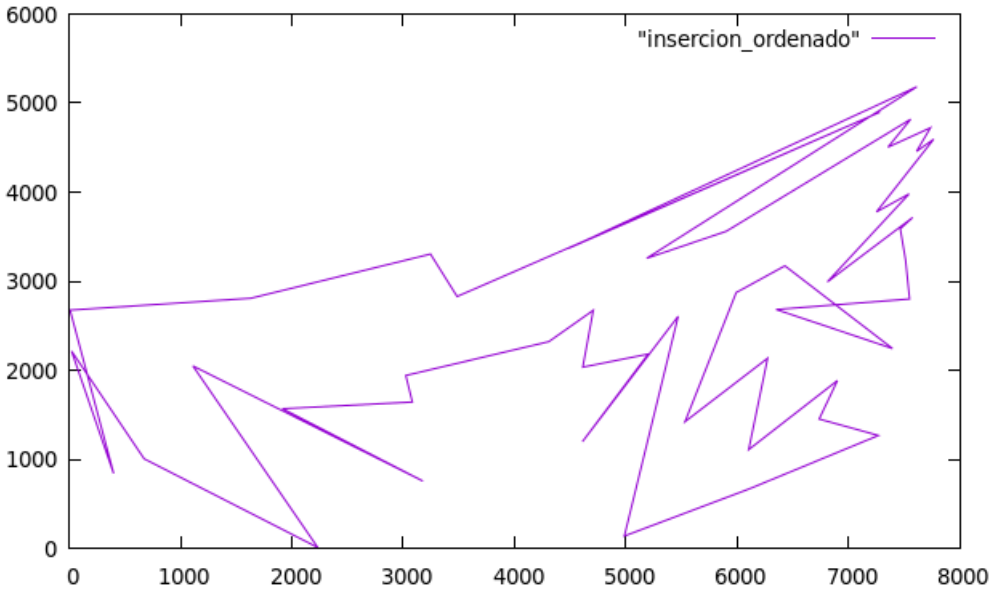

Figure 1.3: Recorrido con el enfoque inserción

Para acabar, si se recorren las ciudades con el método arista que hemos implementado, el recorrido se queda así habiendo comenzado por la ciudad cuyas coordenadas son(7732,4723) y finaliza en la ciudad (7555,4819) y ya vuelve a la inicial:

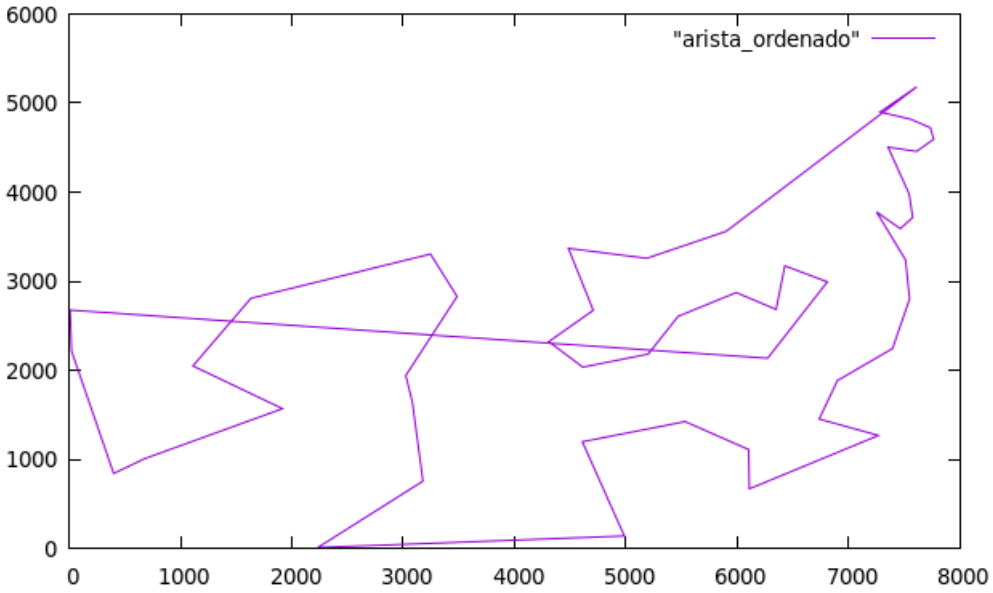

Figure 1.4: Recorrido con el agoritmo arista

1.3.2 Longitud ciclos

Hemos calculado la longitud de los ciclos previos con las distancias redondeadas al entero más próximo y nos han salido los siguientes valores:

Cercanía	161208
Inserción	64134
Arista	48229

Por lo tanto podemos deducir que el mejor enfoque en este caso es el de Arista porque nos sale una distancia recorrida más pequeña.

2 Problema Asignado

2.1 Enunciado

Tenemos que completar un conjunto de n tareas con plazos limite. Cada una de las tareas consume la misma cantidad de tiempo (una unidad) y, en un instante determinado, podemos realizar únicamente una tarea. La tarea i tiene como plazo limite d i y produce un beneficio gi (g i $>$ 0) solo si la tarea se realiza en un instante de tiempo t $<=$ d i . Diseñe un algoritmo voraz que nos permita seleccionar el conjunto de tareas que nos asegure el mayor beneficio posible.

2.2 Algoritmo

```
#include<iostream>
#include<stdlib.h>
#include<time.h>
#include <math.h>
#include <string>

using namespace std;

bool func_sol(int **v, int n){
        bool sol = true;
        for(int j = 0 ; j < n ; j++){
                if(v[j][0] != -1)
                        sol = false;
        }
        return sol;
}

int func_seleccion(int **v, int *sol, int &asig, int n){
        int objetivo = 0;
        int menor,mayor;
        menor = v[0][1];
        mayor = v[0][2];

        for(int i = 1 ; i < n ;i++){
//      Igual duracion pero con mayor beneficio
                if (v[i][1] == menor && v[i][1] > asig){
                        if (v[i][2] > mayor){
                                if(menor == asig)
                                        v[objetivo][0] = -1;
                                menor = v[i][1];
                                objetivo= i;
                                mayor = v[i][2];
                        }
                        else if (v[i][1] <= asig){
                                v[i][0] = -1;
                        }
                }// Menor duracion y dentro cronologicamente
                else if (v[i][1] < menor && v[i][1] > asig){
                        menor = v[i][1];
                        objetivo= i;
                        mayor = v[i][2];
                }// Anterior usado y usamos este
                else if (v[objetivo][0] == -1 && v[i][0] != -1 && v[i][1] > as
                        menor = v[i][1];
```

```
                        mayor = v[i][2];
                        objetivo = i;
                }
        }

        sol[asig] = v[objetivo][2];
        asig++;

        v[objetivo][0] = -1;


        return objetivo;
}

int func_objetivo(int *v, int n){
        int beneficio=0;
        for(int i=0; i<n;i++){
                beneficio += v[i];
        }
        return beneficio;
}

bool func_fact(int *sol, int n){
        if (func_objetivo(sol, n) > 0)
                return true;
        else{
                return false;
        }
}



int main(int argc, char *argv[]){
        int asig = 0, beneficio;
        int n;
        int * sol;
        int ** v;

        cout << 'Cuantas tareas vas a asignar?' << endl;
        cin >> n;

        sol = new int [n];
        v = new int *[n];
```

```
        for (int i = 0 ; i < n ; i++){
                v[i] = new int [3];
                for(int j = 0 ; j < 3 ; j ++){
                        cout << "Fila: " << i << " columna " << j << endl;
                        cin >> v[i][j];
                }
        }

cout << endl;

//      IMPRIMIR
for (int i=0 ; i< n;i++){
        for(int j=0; j <3; j ++){
                cout << v[i][j] << " ";
        }
        cout << endl;
}
//      ALGORITMO
while(!func_sol(v, n)){
        int obj = func_seleccion(v, sol, asig, n);
        if (!func_fact(sol,n)){
                sol[asig-1] = 0;
                asig--;
        }
}

beneficio = func_objetivo(sol,n);
cout << "En un tiempo de " << asig << " ,se ha obtenido un beneficio de "
 << beneficio << endl;

//      IMPRIMIR
        for (int i=0 ; i< n;i++){
                for(int j=0; j <3; j ++){
                        cout << v[i][j] << " ";
                }
                cout << endl;
        }
        cout << endl;
        for (int i=0 ; i< n;i++){
        cout << sol[i] << " ";
        }cout << endl;
        return 0;
}
```

2.3 Ejecución

```
cesar@cesar-TM1701:~/Escritorio/2018-2019_UGR/2º CUATRIMESTRE/ALGORITMICA/PRACTI
CA/ALG19/Practica 3/material_prac3_doble$ ./asignatareas
¿Cuantas tareas vas a asignar?
3
Fila: 0 columna 0
0
Fila: 0 columna 1
1
Fila: 0 columna 2
2
Fila: 1 columna 0
1
Fila: 1 columna 1
3
Fila: 1 columna 2
5
Fila: 2 columna 0
2
Fila: 2 columna 1
5
Fila: 2 columna 2
7

0 1 2
1 3 5
2 5 7
En un tiempo de 3 ,se ha obtenido un beneficio de 14
-1 1 2
-1 3 5
-1 5 7

2 5 7
```

Figure 2.1: Asignación de tareas

El algoritmo va ejecutando el bucle:

```
while(!func_sol(v, n)){
        int obj = func_seleccion(v, sol, asig, n);
        if (!func_fact(sol,n)){
                sol[asig-1] = 0;
                asig--;
        }
}
```

donde mira si el vector es una solución y selecciona a la nueva tarea a ejecutar.
Se ejecutan las funciones solución:

```
bool func_sol(int **v, int n){
        bool sol = true;
```

```
        for(int j = 0 ; j < n ; j++){
                if(v[j][0] != -1)
                sol = false;
        }
        return sol;
}
```

y selección, donde se busca una tarea que tenga el menor plazo de ejecución y entre ellos el de mayor beneficio si hubiera alguna coincidencia.

```
int func_seleccion(int **v, int *sol, int &asig, int n){
        int objetivo = 0;
        int menor,mayor;
        menor = v[0][1];
        mayor = v[0][2];

for(int i = 1 ; i < n ;i++){
//      Igual duracion pero con mayor beneficio
if (v[i][1] == menor && v[i][1] > asig){
        if (v[i][2] > mayor){
if(menor == asig)
v[objetivo][0] = -1;
menor = v[i][1];
objetivo= i;
mayor = v[i][2];
}
else if (v[i][1] <= asig){
v[i][0] = -1;
}
}// Menor duracion y dentro cronologicamente
else if (v[i][1] < menor && v[i][1] > asig){
menor = v[i][1];
objetivo= i;
mayor = v[i][2];
}// Anterior usado y usamos este
else if (v[objetivo][0] == -1 && v[i][0] != -1 && v[i][1] > asig){
menor = v[i][1];
mayor = v[i][2];
objetivo = i;
}
}

sol[asig] = v[objetivo][2];
asig++;
```

```
v[objetivo][0] = -1;


return objetivo;
}
```

Mas adelante mira si el nuevo valor es factible y si no, lo elimina de la solución. Cuando acaba el bucle del algoritmo, llama a la función para obtener el beneficio total de todas las tareas y lo muestra por pantalla.

```
beneficio = func_objetivo(sol,n);
cout << "En un tiempo de " << asig << " ,se ha obtenido un beneficio de "
<< beneficio << endl;
```