Algorítmica

Practica 3

parte II

Problema del Viajante de Comercio

Grupo 1

Fc.Javier Azpeitia Muñoz
Christian Andrades Molina
Guillermo Muriel Sánchez lafuente
Miguel Keane Cañizares
Mercedes Alba Moyano

❑ Introducción.

Esta práctica consiste en un estudio de diferentes heurísticas para la resolución del Problema del Viajante de Comercio. Dejando de lado la búsqueda de una solución óptima, el proyecto se centra en las implementaciones de diversas heurísticas que obtienen soluciones aproximadas.

Dichas heurísticas permiten reducir el tiempo de ejecución desde tiempo exponencial a tiempo polinómico. Las heurísticas implementadas son las siguientes:vecino mas cercano, inserción y una propia**.**

En este trabajo se contrastan las complejidades de las mencionadas heurísticas, y la solución que construyen, con las mejores soluciones conocidas de los problemas reales, accesibles a través de internet.

❑ Descripción del Problema.

El problema del viajante, al que a partir de ahora nos referiremos como TSP (Travelling Salesman Problem), se plantea de la siguiente manera: dados un número finito de ciudades (nodos) que tiene que visitar nuestro viajante o vendedor, así como la distancia o el coste entre cada par de ciudades, tenemos que encontrar la manera de que nuestro viajante visite todas las ciudades una sola vez con el menor coste posible y de manera que el último viaje sea a la ciudad de partida para cerrar el recorrido.

Este ha sido un problema ampliamente tratado desde distintos campos tales como por ejemplo la Biología, la Inteligencia Artificial y las Matemáticas, ya que el planteamiento tan sencillo del TSP surge en muchos ámbitos científicos y su interés se extiende más allá de la simple resolución teórica del problema en un tiempo razonable. Tiene aplicaciones industriales en las que una solución óptima ahorraría mucho tiempo y dinero. Otras aplicaciones prácticas del TSP se presentan en el análisis de la estructura de cristales, recogida de pedidos en almacenes, identificación de conglomerados en un vector de datos, elaboración de rutas escolares, etc...

La existencia de un algoritmo que resuelva el problema en tiempo exponencial es evidente. También es conocido que no existe ningún algoritmo eficiente que resuelva el problema, así como que su problema de decisión asociado es NP-Completo.

Todo esto unido a la amplitud de áreas de aplicación ha provocado la aparición de una gran cantidad de algoritmos para resolver casos particulares, consiguiendo a lo largo del siglo pasado soluciones prácticas para un número de ciudades cada vez mayor, llegándose en 1993 a resolver un caso particular con 4461 ciudades.

Este progreso se debe sólo parcialmente a las vertiginosas mejoras en el hardware de las computadoras; habiendo sido ante todo posible gracias al desarrollo de teorías matemáticas (en particular combinatoria poliédrica) y de algoritmos eficientes.

No obstante los algoritmos desarrollados no son estables en el sentido de que los tiempos de ejecución varían fuertemente para problemas diferentes con el mismo número de ciudades y a la no existencia de una función en el número de ciudades que dé una pequeña idea del tiempo necesario para resolver un problema particular.

Además los problemas que se presentan en la práctica pueden tener un número de ciudades que hacen imposible la aplicación de un algoritmo que produzca una solución óptima, ya sea por razones de tiempo real, tiempo de CPU, etc...

Todo esto ha provocado que se haya concentrado mucho esfuerzo investigador en el desarrollo de algoritmos heurísticos eficientes cuyo propósito es encontrar una solución aproximada de la solución óptima.

De hecho se abre un gran abanico de posibilidades en cuanto a si se mejora la calidad del camino obtenido o bien se quiere conseguir una solución suficientemente buena en menos tiempo.

Esto último ha sido el objetivo de nuestra práctica, el estudio de algunas de estas aproximaciones, que a partir de ahora llamaremos heurísticas, y el análisis de los resultados de las mismas, comparando unas con otras tanto en tiempo de ejecución como en lo óptimo del resultado obtenido.

Para un tratamiento más sistemático y más cercano al lenguaje matemático, a partir de ahora tomaremos las ciudades como nodos de un grafo, cuyas aristas están valoradas por la distancia entre las ciudades que unen. Así podemos abordar el problema como un problema de teoría de grafos y aplicar algunos resultados de la misma. Por tanto, la solución del TSP se presenta como un recorrido del grafo en el que se visitan todos sus nodos, no se repite ningún nodo y en el que el nodo final es el mismo que el inicial (comenzamos y terminamos en la misma ciudad, pudiendo ser ésta cualquiera de ellas).

❑ Heurísticas.

En esta parte pasamos a considerar la parte computacional del TSP, por medio de las heurísticas, que son procedimientos que determinan un tour de acuerdo a algunas reglas de construcción.

Las heurísticas que vamos a utilizar son:

- Vecino más cercano
- Inserción
- La propia

A partir de aquí, asumiremos que nos es dado el grafo completo no-dirigido G_n con pesos en las aristas C_{uv} entre cada par de nodos u y v. Denotaremos por V al conjunto de todos los nodos del grafo y asumiremos que, V = {1,2,...,n}. El objetivo de las heurísticas será hallar buenos ciclos Hamiltonianos en este grafo.

❑ Vecino más cercano.

Esta es la aproximación o Heurística mas intuitiva cuando de metodologías greedy se trata, es decir, la descripción de un algoritmo voraz encaja a la perfección con esta heurística,

La definición formal podría desarrollarse como:
Dado un conjunto de Nodos $G_n / n \in N \{0,1,2,3,...n-1\}$ y sea un conjunto de aristas E que unen 2 ciudades u, v con un coste C_{uv} existe un $S \subseteq G$ tal que para cada nodo de S se pueden visitar todos los demás nodos de S sin repetir ninguno de ellos y con un coste total C lo más bajo posible (No tiene porque ser el óptimo.). Para incluir un nodo G_n en S se debe cumplir que C_{uv} sea el menor para cada uno de los posibles C_{uvi}, asegurando así la búsqueda del óptimo local, pero no pudiendo garantizar la optimalidad global.

En nuestro caso computacional en concreto, el conjunto de nodos es representado como números enteros desde el 0 hasta n-1 nodos, y las distancias son representadas en una matriz, donde cada $M_{i,j}$ representa el coste que tiene ir del nodo i al nodo j siendo $i \neq j$ .La matriz es simétrica, ya que $C_{uv} = C_{vu}$.

El proceso de localización e inclusión de un nodo en el conjunto solución sigue los siguientes pasos:

-1º Localizar de entre todos los caminos posibles para un nodo base u aquel con el menor coste C_{uv} .

-2º Una vez localizado el C_{uv} menor, incluir v en S.

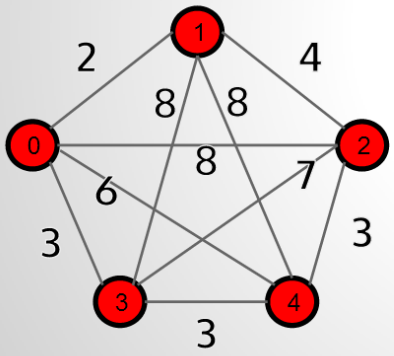
-3º Si u no es el primer nodos de S se deben eliminar todos los caminos en los que esté incluido u .

-4º Volver al paso 1 tomando ahora como nodo base v.

Esta heurística ha demostrado dar muy buenos resultados en las pruebas realizadas sobre un programa que implemente esta aproximación, lo cual nos ha sido un poco sorprendente debido a la simplicidad de la aproximación.

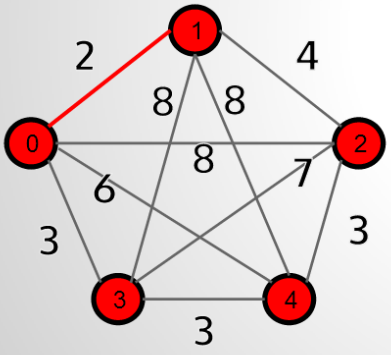
Un ejemplo seria:

- Ejemplo del vecino más cercano :



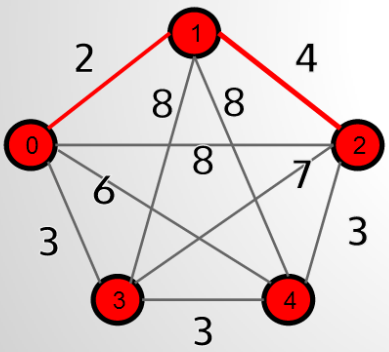
	0	1	2	3	4
0	**0**	**2**	**8**	**3**	**6**
1	2	0	4	8	8
2	8	4	0	7	3
3	3	8	7	0	3
4	6	8	3	3	0

- Ejemplo del vecino más cercano :



	0	1	2	3	4
0	0	0	8	3	6
1	**0**	**0**	**4**	**8**	**8**
2	0	0	0	7	3
3	0	0	7	0	3
4	0	0	3	3	0

- Ejemplo del vecino más cercano :



	0	1	2	3	4
0	0	0	0	3	6
1	0	0	0	8	8
2	**0**	**0**	**0**	**7**	3
3	0	0	0	0	3
4	0	0	0	3	0

Así sucesivamente hasta que se termina el ciclo completo.

Las estructuras de datos usadas han sido una matriz cuadrada bidimensional de tamaño n y una lista con funcionalidad de Cola FIFO. (Conjunto solución, es decir, el recorrido)

❑ Inserción.

La idea de éste tipo de heurísticas es la de empezar con un pequeño conjunto de nodos usado como tour inicial, para después ir extendiendo este tour inicial insertando los restantes nodos.

La estructura general de éste tipo de heurísticas es la siguiente:

Procedimiento inserción

-1. Seleccionar un tour inicial de k nodos (k>1) de entre los nodos existentes

-2. Mientras existan nodos no insertados al tour hacer:

a. Seleccionar un nodo de entre los nodos no pertenecientes al tour

provisional de acuerdo con algún criterio.

b. Insertar el nodo seleccionado en el tour provisional.

Fin procedimiento

De esta manera se irán incluyendo en el tour todos los nodos, quedando al final un tour Hamiltoniano compuesto por todos los nodos.

La principal diferencia entre las heurísticas de inserción es el criterio que tenemos para elegir el nodo a insertar, y el criterio que usamos a la hora de insertar el nodo elegido.

El tour inicial normalmente es un tour de dos o tres nodos.

En nuestra práctica el criterio a seguir es el siguiente: de entre todos los nodos no insertados, se elige un nodo cuya inserción cause el mínimo incremento en la longitud del tour provisional, insertándose de hecho en la posición donde menos se incremente la longitud del tour provisional.

Más particularmente, en nuestro caso para decidir el tour inicial hemos usado 3 nodos con valores de C_{uv} alto, C_{vy} también alto y que cumpla que C_{uy} también sea alto. Con esto intentamos representar que los 3 nodos del tour inicial “envuelven” el grafo.

Los resultados obtenidos con esta aproximación han sido bastante poco consistentes, es decir, en algunos casos los tiempos de ejecución del algoritmo eran muy buenos, y en otros casos, con otros grafos, eran bastante malos. Manteniéndonos en el orden polinómico en todo momento $O(n^2)$ lo cual sigue siendo mucho mejor que el orden exponencial que nos daría la solución óptima y que es a día de hoy no válido.

Para el método de inserción existen otros criterios también válidos, que no hemos podido probar por falta de tiempo, pero si que merece al pena nombrarlos:

-De entre todos los nodos que están conectados al tour por el subgrafo de candidatos, insertar el nodo cuya mínima distancia al tour sea máxima. Si no hay ningún nodo del subgrafo de candidatos conectado al tour, entonces insertar un nodo aleatorio.

Este sería otro criterio válido para resolver el problema.

❑ Nuestra Heurística.

Nuestra Heurística está basada en la selección aleatoria de una arista con un coste C_{uv} sin usar ningun tipo de criterio para la busqueda de mínimos locales ni nada parecido. Es extremadamente simple, pero los resultados obtenidos no difieren en gran medida de otras Heurísticas supuestamente mejores.

Los pasos a seguir para esta aproximación son los siguientes:

- ➔ 1º Dado un nodo base del que partimos u seleccionamos de entre todos los posibles $n-1$ caminos o aristas la primera.
- ➔ 2º Introducimos este nuevo nodo v al conjunto solución.
  - a) SI el nodo u es el primer nodo del circuito, se pasa al paso 3
  - b) Si el nodo u no es el primer nodo del circuito, eliminamos todos los caminos en los que esta incluido.
- ➔ 3º Volvemos al paso 1 tomando ahora v como nodo base.

Somos conscientes de que existen muchas otras heurísticas que podrían dar mejores resultados, pero esta es la que hemos elegido…

❑ Códigos Fuente

Vecino mas cercano:

```
#include <iostream>
using namespace std;
#include <cstdlib>
#include <cassert>
#include <errno.h>
#include <list>

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

/*--------------------------*/
// ENTRADA INICIAL DE DATOS

if (argc != 2)
    {
      cerr << "Formato " << argv[0] << " <tamaño_grafo>" << endl;
      return -1;
    }

int n = atoi(argv[1]);
/*--------------------------*/

//INICIALIZACION DE VARIABLES

int u=0; //etiqueta del nodo
int matriz[n][n];
int matriz2[n][n];

list <int> vertice;//Subconjunto solución

int minimo;
double px,py;
int datos[n*2];
int distancias[n];
int contadore,contadord,contadordis;

/*--------------------------*/
// Rellenamos matriz a ceros

for(int i=0; i<n; i++){
    for(int j=0; j<n; j++){
              matriz[i][j] = 0;
    }
}

/*--------------------------*/
// Entrada de datos (Guarda en vector todas las coordenadas)

/* contadore = 0;
contadord = 0;
while (contador<n){

    cin >> px >> py;

datos[contadord] = px;
datos[contadord+1] = py;
```

```
cout << "contador: " << contadore << endl;
contadord = contadord + 2;
contadore++;
}

for (int i=0; i<n*2; i++)
    cout << datos[i] << " ";

    cout << "\n\n";
*/

/*-------------------------------------------------*/
//Matriz rellenada a distancias (no funciona)

/*contadordis = 0;

for (int i=0; i<n; i++){
    for (int j=2; j<n; j++){

    distancias[contadordis] = sqrt(pow(datos[j]-datos[i],2) +
pow(datos[j+1]-datos[i+1],2));
    contadordis++;

    }
}

for (int i=0; i<n*2; i++)
    cout << distancias[i] << " ";

*/

// EJEMPLO PARA EL CODIGO
/*----------------*/
matriz[0][1]=2;
matriz[0][2]=8;
matriz[0][3]=3;
matriz[0][4]=6;
/*----------------*/
matriz[1][0]=2;
matriz[1][2]=4;
matriz[1][3]=8;
matriz[1][4]=8;
/*----------------*/
matriz[2][0]=8;
matriz[2][1]=4;
matriz[2][3]=7;
matriz[2][4]=3;
/*----------------*/
matriz[3][0]=3;
matriz[3][1]=8;
matriz[3][2]=7;
matriz[3][4]=3;
/*----------------*/
matriz[4][0]=6;
matriz[4][1]=8;
matriz[4][2]=3;
matriz[4][3]=3;
/*----------------*/


for(int i=0; i<n; i++){
```

```
    for(int j=0; j<n; j++){
                matriz2[i][j] = 0;
    }
}

matriz2[0][1]=2;
matriz2[0][2]=8;
matriz2[0][3]=3;
matriz2[0][4]=6;

/*-----------------*/
// Matriz rellenada

cout << "\nMATRIZ RELLENADA: \n";
for (int a=0;a<n;a++){
        for (int b=0;b<n;b++){
            cout << " " << matriz[a][b] << " ";
        }
        cout << "\n";
    }
    cout << "\n";

/*-----------------*/

bool aumento;         //Limitar a una ejecución por fila hasta encontrar primer minimo
int contador;         //contador1 = posición del primer elemento distinto de cero
int contador2;        //contador2 = en ultima iteracción, ponemos a cero y sumamos
distancia minima
int val1,val2;
int contador_bucle = 0;
int suma_distancia = 0;      //Distancia minimal


for(int i=0; contador_bucle < n-1;){

    contador_bucle++;
    aumento = true;
    contador2 = 0;

    for(int j=0; j<n; j++){
        if (aumento){
            contador = 0;

            while (matriz[i][contador]==0){               //Calculamos primer elemento !=0
que será el primer minimo
                contador++;
            }

            minimo = matriz[i][contador];
            val1 = i;
            val2 = contador;
            aumento = false;
        }

        contador2++;
        if(matriz[i][j]!=0){                        //Calculamos minimo real de la fila
distinto de cero
            if(matriz[i][j] < minimo){
                minimo = matriz[i][j];
                val1 = i;
                val2 = j;
```

```
            }
        }

        if (contador2 == n){                        //Sumar distancias y poner a cero matriz
            suma_distancia = suma_distancia + minimo;
            i = val2;
            vertice.push_back(val1);

             for(int j=0; j<n; j++){

                            matriz[j][val1]=0;
                            matriz[j][val2]=0;
            }

        }

        if (contador_bucle==(n-1) and j==n-1){            //Si es ultimo nodo, añadir
distancia hacia el primer nodo
            vertice.push_back(val2);
            suma_distancia = suma_distancia + matriz2[0][val2];

        }
    }
}

cout << "\nMATRIZ MODIFICADA: \n";

for (int a=0;a<n;a++){
        for (int b=0;b<n;b++){
            cout << " " << matriz[a][b] << " ";
        }
        cout << "\n";
    }
    cout << "\n\n";

cout << "- Resultado: [";

while (!vertice.empty()){
    u=vertice.front();
        cout << u << " ";
        vertice.pop_front();
}

cout << "]\n";
cout << "- DISTANCIA: " << suma_distancia << "\n\n";

return 0;

}
```

Insercion:

```
#include <iostream>
using namespace std;
#include <cstdlib>
#include <cassert>
#include <errno.h>
#include <list>
#include <map>
```

```
#include <vector>
#include <fstream>
#include <sstream>
#include <set>
#include <cmath>

void leer_orden(string & nombre, vector<int> & V){
    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;
    V.reserve(n);
    int i=0;
    while (i<n) {
        datos >> act;
    V.push_back(act);
    i++;
      }

    datos.close();
    }
   else {
      cout << "Error de Lectura en " << nombre << endl;
    }
 }

void leer_puntos(string & nombre, map<int, pair<double, double> > & M, int &n){
    ifstream datos;
    string s;
    pair<double,double> p;
    int 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;
    }
 }

class CLParser
{
public:

    CLParser(int argc_, char * argv_[],bool switches_on_=false);
```

```
    ~CLParser(){}

    string get_arg(int i);
    string get_arg(string s);

private:

    int argc;
    vector<string> argv;

    bool switches_on;
    map<string,string> switch_map;
};

CLParser::CLParser(int argc_, char * argv_[],bool switches_on_)
{
    argc=argc_;
    argv.resize(argc);
    copy(argv_,argv_+argc,argv.begin());
    switches_on=switches_on_;

    //map the switches to the actual
    //arguments if necessary
    if (switches_on)
    {
        vector<string>::iterator it1,it2;
        it1=argv.begin();
        it2=it1+1;

        while (true)
        {
            if (it1==argv.end()) break;
            if (it2==argv.end()) break;

            if ((*it1)[0]=='-')
                switch_map[*it1]=*(it2);

            it1++;
            it2++;
        }
    }
}

string CLParser::get_arg(int i)
{
    if (i>=0&&i<argc)
        return argv[i];

    return "";
}

string CLParser::get_arg(string s)
{
    if (!switches_on) return "";

    if (switch_map.find(s)!=switch_map.end())
        return switch_map[s];

    return "";
}
```

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

  map<int, pair<double, double> >  m;
  vector<int> V;
  string fp,fok;
    int n;
   if (argc == 1) {
     cout << "Error Formato:  tsp puntos orden_correcto" << endl;
     exit(1);
   }

    CLParser cmd_line(argc,argv,false);

    fp = cmd_line.get_arg(1);
    leer_puntos(fp,m,n);
    fok = cmd_line.get_arg(2);

    leer_orden(fok,V);

    double distancia[n][n],recorrido[n][n];
    int v;
    map<int, pair<double, double> >::iterator it1, it2;
    int i,j;
    for(it1=m.begin(),i=0;it1!=m.end();++it1,i++){
        for(it2=it1, j=0; it2!=m.end();++it2,j++){
            if(i!=j){

v=sqrt(pow((((*it2).second).first-((*it1.second).first),2.0)+pow((((*it2).second).second-
((*it1.second).second),2.0));
                distancia[i][j]=v;
                distancia[j][i]=v;
            }
            else{
                distancia[i][j]=0;
            }
        }
    }

    set<int> ciudades;
    int cnorte=0,ceste=0,coeste=0;
    int vnorte=-1000,veste=-1000,voeste=1000;
    int distancia_total=0,dist_local,pos,aux;
    vector<int> camino_final,v_local;
    vector<int>::iterator it_v,it_aux;

    for(it1=m.begin();it1!=m.end(); ++it1){
        if(((*it1).second).second>vnorte ){
            vnorte = ((*it1).second).second;
            cnorte = (*it1).first;
        }
    }
    ciudades.insert(cnorte); camino_final.push_back(cnorte);
    for (it1=m.begin();it1!=m.end();++it1){
        if(((*it1).second).first>veste && (*it1).first != cnorte){
            veste = ((*it1).second).first;
            ceste = (*it1).first;
        }
    }
    ciudades.insert(ceste);
    camino_final.push_back(ceste);
```

```
    for (it1=m.begin();it1!=m.end();++it1){
        if(((*it1).second).first<voeste && (*it1).first != cnorte && (*it1).first !=
ceste){
            voeste = ((*it1).second).first;
            coeste = (*it1).first;
        }
    }
    ciudades.insert(coeste);
    camino_final.push_back(coeste);

    for (it1=m.begin();it1!=m.end();++it1){
        if(((*it1).second).first<voeste && (*it1).first != cnorte && (*it1).first !=
ceste){
            voeste = ((*it1).second).first;
            coeste = (*it1).first;
        }
    }
    ciudades.insert(coeste);
    camino_final.push_back(coeste);

    int tamloc=n-3,ciudad;
    while(tamloc>0){
        for (it1=m.begin();it1!=m.end();++it1){
            dist_local=1000;
            v_local.clear();
            v_local.insert(v_local.begin(),camino_final.begin(),camino_final.end());

            for(j=0;j<camino_final.size();j++){
                if(ciudades.find((*it1).first)==ciudades.end()){
                    for(i=0,it_v=v_local.begin();i<=j;i++,++it_v){}
                        v_local.insert(it_v,(*it1).first);
                            aux= distancia_recorrida(n,distancia,v_local);
                            if(aux<dist_local){
                                ciudad=(*it1).first;
                                dist_local=aux;
                                pos = j;
                            }
                }

                v_local.clear();
                v_local.insert(v_local.begin(),camino_final.begin(),camino_final.end());

            }
        }
        for(i=0,it_aux=camino_final.begin();i<=pos;i++,++it_aux){}

        ciudades.insert(ciudad);
        camino_final.insert(it_aux,ciudad);

        tamloc--;
    }
    for(i=0;i<camino_final.size();i++){
        cout<<camino_final[i]<<" ";
    }
    cout << endl;
    cout << distancia_recorrida(n,distancia,camino_final);
    cout << endl;



}
```

Heuristica de grupo (Propia):

```
#include <iostream>
using namespace std;
#include <ctime>
#include <cstdlib>
//#include <chrono>

//using namespace std::chrono;

//high_resolution_clock::time_point tantes, tdespues;
//duration<double> transcurrido;

int main(){

int n;
cin >> n;

int matriz[n][n]; //Matriz
int v1, v2;

int vertice[n]; //Subconjunto solución
int t = 0; //Contador que aumentaremos en 2 para meter los datos en vértice

/*-------------------------------------------------*/
//Rellenamos matriz a ceros

for(int i=0; i<n; i++){
    for(int j=0; j<n; j++){
              matriz[i][j] = 0;
    }
}

/*while ((v1!=-1 and v2!=-1) and (v1<n or v2<n)){

      cin >> v1 >> v2;
      matriz[v1][v2]=1;
}*/

//EJEMPLO PARA RELLENAR LA MATRIZ
/*----------------*/
matriz[0][0]=0;
matriz[0][1]=2;
matriz[0][2]=8;
matriz[0][3]=3;
matriz[0][4]=6;
/*----------------*/
matriz[1][0]=2;
matriz[1][1]=0;
matriz[1][2]=4;
matriz[1][3]=8;
matriz[1][4]=8;
/*----------------*/
matriz[2][0]=8;
matriz[2][1]=4;
matriz[2][2]=0;
matriz[2][3]=7;
matriz[2][4]=3;
/*----------------*/
matriz[3][0]=3;
```

```
matriz[3][1]=8;
matriz[3][2]=7;
matriz[3][3]=0;
matriz[3][4]=3;
/*----------------*/
matriz[4][0]=6;
matriz[4][1]=8;
matriz[4][2]=3;
matriz[4][3]=3;
matriz[4][4]=0;
/*----------------*/
/*-------------------------------------------------*/

//Matriz rellenada

cout << "\nMATRIZ RELLENADA: \n";
for (int a=0;a<n;a++){
        for (int b=0;b<n;b++){
            cout << " " << matriz[a][b] << " ";
        }
        cout << "\n";
    }
    cout << "\n\n";

/*-------------------------------------------------*/

/* Cogemos un arista y sus dos vértices. Posteriormente los quitaremos y volveremos a
elegir una arista de manera aleatoria */

//tantes = high_resolution_clock::now();

int ultima_ciudad = n-1;//Metemos la última ciudad que introduciremos al final
int distancia, suma_distancia;

for(int x=0; x<n; x++){
    for(int y=0; y<n; y++){
        if(matriz[x][y]!=0){ //Si encuentra la ciudad con una distancia a otra distinta
de 0

        //Mete los dos vértices subconjunto solución
        vertice[t]=x;
            vertice[t+1]=y;
        t = t + 2;

        //SUMA DE LAS DISTANCIAS
            distancia = matriz[x][y];
        suma_distancia = suma_distancia + distancia;

        for(int d=0; d<n; d++){ //Borrar las conexiones con esos vértices(FILAS)
                    matriz[x][d]=0;
                    matriz[y][d]=0;
            }

            //Borramos las Columnas
            for(int d=0; d<n; d++){
                    matriz[d][x]=0;
                    matriz[d][y]=0;
            }
        }
    }
}
```

```
vertice[t]= ultima_ciudad; //ULTIMA CIUDAD AÑADIDA

//tdespues = high_resolution_clock::now();

// Matriz resultante

/*cout << "MATRIZ MODIFICADA: \n";

for (int a=0;a<n;a++){
        for (int b=0;b<n;b++){
            cout << " " << matriz[a][b] << " ";
        }
        cout << "\n";
    }
    cout << "\n\n";
*/

cout << "Resultado: [";

for(int g=0; g<=t; g++)
    cout << vertice[g] << " ";

cout << "]\n";

cout <<" Distancia: "<< suma_distancia<< endl;



//transcurrido = duration_cast<duration<double>>(tdespues - tantes);
//cout << n << "      " << transcurrido.count() << endl;

return 0;

}
```