Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez

En esta práctica vamos a analizar el uso de los algoritmos "voraces" o "greedy", algoritmos que seleccionan en cada momento lo mejor de entre un conjunto de candidatos, sin tener en cuenta lo ya hecho, para obtener una solución "rápida" al problema.

Vamos a tener dos problemas a los cuales vamos a aplicar esta manera de resolverlos y mediremos su eficiencia teórica.

Una vez diseñado el algoritmo, veremos los resultados de la ejecución y los compararemos con los resultados "óptimos", generados tras resolver el problema de la menor manera posible.

Recordemos que los algoritmos greedy no aseguran generar soluciones optimales siempre; esta desventaja es una ventaja en problemas en los que es muy difícil alcanzar la solución óptima, apliquemos el algoritmo que apliquemos, como el problema que se propone a continuación. No obstante, veremos que los resultados, a pesar de no ser los óptimos, son bastante eficientes, así como el tiempo de ejecución del algoritmo.

1. Problema común (Viajante de comercio)

Como hemos comentado anteriormente, aplicar un algoritmo que nos dé el resultado más óptimo para este problema es bastante complicado y su tiempo de ejecución se incrementaría bastante.

Es por eso por lo que el enfoque Greedy es una manera eficiente de solucionar este problema, generando un resultado que no es el óptimo pero se acerca a ello.

El problema se resume en encontrar un circuito hamiltoniano para una serie de puntos, en este caso ciudades, de manera que se recorran todas ellas sin volver a pasar por ninguna, de manera que la distancia total entre estas ciudades, es decir, del circuito, sea la mínima (y así minimizamos el recorrido).

1.1. Algoritmo basado en cercanía

En primer lugar, hemos desarrollado una estrategia basada en encontrar el "vecino más cercano": tomamos una ciudad inicial inicial de manera arbitraria, y buscamos en el vector de ciudades que no se han visitado la ciudad más cercana a esta. Una vez encontrada, se procede a hacer un borrado lógico de la ciudad en el vector, y se procede a encontrar la ciudad más cercana a esta última visitada.

El procedimiento se repite hasta que todas las ciudades se hayan visitado, obteniendo el camino.

Hemos creado también una clase matriz que hemos usado de forma auxiliar para simplificar la parte del código del algoritmo que se detalla a continuación.

```
1 #include <iostream>
```

Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez


```cpp
#include <fstream>
#include <string>
#include <vector>
#include <utility>
#include <cmath>
#include <limits>
#include "matriz.h"

using namespace std;
const double INF = numeric_limits<double>::max();


/*  FUNCION
*  Esta funcion recibe como parametros un vector, que son las distintas distancias
* a las distintas ciudades desde una misma ciudad y el vector candidatos para comprobar
* si ya se han visitado las respectivas ciudades.
*
* La funcion encuentra el menor elemento del vector v y devuelve la posicion dentro
* del vector de dicho elemento.
*/
int BuscaMenor(vector<double> v, vector<int> & candidatos){
    int menor;
    double minimo = INF;
    for(int i = 0;i< v.size(); ++i){
        if( v[i] < minimo && candidatos[i] !=-1){
            minimo = v[i];
            menor = i;
        }
    }
    return(menor);
}


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

    ifstream input_file(argv[1]);
    string line;
    int dimension;

    int ciudad;
    double coor_x, coor_y;

    vector<pair<double, double> > v;
    pair<double, double> p;

```

Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez


```
47      //Movemos el offset a la linea en la que se dice la
           ↪dimension del grafo
48      for(int i=0; i<3; ++i){
49          getline(input_file, line);
50      }
51
52      getline(input_file, line);
53      line.erase(0, 11);
54      dimension = atoi(line.c_str());
55
56      //Reservamos espacio para el grafo
57      matriz <double> m(dimension, dimension, INF);
58
59      //Movemos el offset al comienzo de los datos
60      for(int i=0; i<3; ++i){
61          getline(input_file, line);
62      }
63
64      //Tomamos los datos
65      for (int i=0; i<dimension; ++i){
66          input_file >> ciudad;
67          input_file >> coor_x;
68          input_file >> coor_y;
69          p.first = coor_x;
70          p.second = coor_y;
71          v.push_back(p);
72      }
73
74      //Calculamos las distancias y las metemos en la matriz
           ↪ que representa
75      //el grafo
76      for(int i=0; i<dimension; ++i){
77          for (int j=i+1; j<dimension; ++j){
78              m[i][j]=sqrt(pow(v[j].first - v[i].first, 2) +
                   ↪ pow(v[j].second - v[i].second, 2));
79              m[j][i]=m[i][j];
80          }
81      }
82
83      //Algoritmo basado en cercanias. La idea es ir a la
           ↪ciudad mas cercana
84      //sin formar un ciclo. Exceptuando la ultima.
85
86      //Declaramos los vectores que albergaran los conjuntos
           ↪ Candidato y Solucion
87      vector<int> solucion;
88      vector<int> candidatos;
89
90      // Inicializamos el conjunto de candidatos, el rango
```

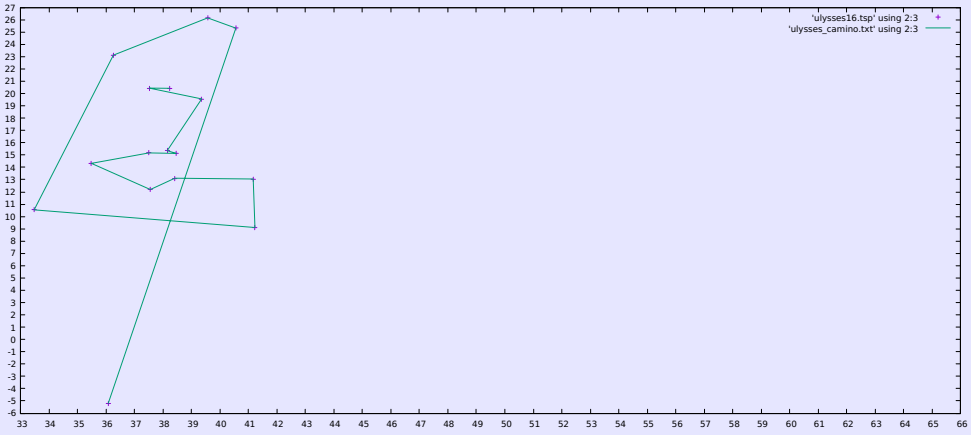
Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez


```
        ↪sera [0,15].
91      for(int i = 0; i<dimension;++i){
92          candidatos.push_back(i);
93      }
94
95      // Abergamos la primera ciudad en el conjunto solucion
            ↪.
96      int i = 0;
97      solucion.push_back(0);
98      candidatos[0] = -1;
99
100     // variable donde guardaremos el indice, es decir, la
            ↪ciudad a donde nos dirigimos.
101     int menor;
102
103     /*CUERPO DEL ALGORITMO:
104     * La idea es encontrar la ciudad mas cercana haciendo
            ↪uso de la matriz de
105     * distancias. Una vez encontrada la ciudad (indice) al
            ↪ que nos dirigimos,
106     * la posicion candidatos[indice] lo hacemos -1 para
            ↪mostrar que esa ciudad
107     * ya la hemos visitado y introducimos el indice en el
            ↪vector de soluciones.
108     *
109     * Para concluir, asignamos el valor del indice a la
            ↪variable i para empezar
110     * de nuevo todo el proceso
111     */
112     while(solucion.size()< dimension   ){
113         vector <double> c;
114         m.get_Fila(i,c);
115         menor =BuscaMenor(c, candidatos);
116         solucion.push_back(menor);
117         candidatos[menor] = -1;
118         i = menor;
119     }
120
121     // Imprimimos el vector solucion teniendo en cuenta
            ↪que para la implementacion
122     // La ciudad numero 1 ha sido el indice numero 0, por
            ↪lo tanto tenemos que sumar
123     // 1 a los valores del vector solucion.
124
125     for(int i = 0; i< solucion.size();i++){
126         cout << solucion [i]  + 1<< " --> ";
127     }
128     // Aniadimos la ciudad inicial para indicar que
            ↪completamos un ciclo.
```

Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez


```
129     cout << " 1 " << " FIN.";
130
131
132     // Volcamos la salida a un fichero para su
            ↪visualizacion con la ayuda de
133     // GNUPLOT.
134
135     ofstream ficherosalida("data/ulysses_camino.txt");
136     for (int i = 0; i < dimension; ++i){
137         int c = solucion[i];
138         ficherosalida << c+1 << " " << v[c].first << " "
                ↪<< v[c].second << endl;
139     }
140
141
142     //FIN.
143     return (0);
144 }
```

1.1.1. Visualización



1.1.2. Eficiencia teórica

La eficiencia teórica O(n) depende del número de ciudades que hay. Tomamos, por tanto, $TAM = dimension = n$.

La eficiencia del algoritmo, en el peor de los casos, es

$$T(n) = \sum_{i=1}^{n}\sum_{j=1}^{n}$$

$$T(n) = n * n$$

$$T(n) \in O(n^2)$$

Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez

Esto es debido a que la función *BuscaMenor* es de tiempo n, y se ejecuta también n veces en el bucle *while* de la línea 112.

Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez

1.2. Algoritmo basado en inserción

Este algoritmo greedy para resolver el problema del viajante de comercio consiste en partir de un circuito inicial. En nuestro caso las ciudades elegidas para el circuito inicial son la más al Este, la más al Oeste y la más al Norte. Una vez escogido el recorrido inicial comienza el algoritmo. Nuestro algoritmo de inserción se basa en insertar en cada iteración la ciudad que menos aumenta el tamaño de este.

```
1  #include <iostream>
2  #include <fstream>
3  #include <string>
4  #include <vector>
5  #include <utility>
6  #include <cmath>
7  #include <limits>
8  #include "matriz.h"
9
10 using namespace std;
11
12
13 /**
14 Este programa busca un ciclo que recorra todas las
   ↪ciudades de un mapa mediante un algoritmo de tipo
   ↪greedy. En esta version del algoritmo, dado un
   ↪recorrido inicial que contiene tres nodos en cada
   ↪paso se busca el nodo mas cercano al conjunto
   ↪solucion que se encuentre en el conjunto de
   ↪candidatos. El nodo mas cercano es insertado en la
   ↪posicion del vector que menos aumente el recorrido.
15 */
16 int main(int argc, char **argv){
17     const double INF = numeric_limits<double>::max();
18
19     if(argc < 2){
20         cout << "ERROR. Faltan argumentos [archivo de
              ↪datos]" << endl;
21         exit(-1);
22     }
23
24     ifstream input_file(argv[1]);
25     string line;
26     int num_ciudades;
27
28     int ciudad;
29     double coor_x, coor_y;
30
31     vector<pair<double, double> > v_coordenadas;
32     pair<double, double> coordenadas;
33
34     vector<int> candidatos;
```

Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez


```
35      vector<int> solucion;
36
37      //Movemos el offset a la linea en la que se dice la
           ↪dimension del grafo
38      for(int i=0; i<3; ++i){
39          getline(input_file, line);
40      }
41
42      getline(input_file, line);
43      line.erase(0, 11);
44      num_ciudades = atoi(line.c_str());
45
46      //Reservamos espacio para el grafo
47      matriz <double> distancias(num_ciudades, num_ciudades,
           ↪ 0);
48
49      //Movemos el offset al comienzo de los datos
50      for(int i=0; i<3; ++i){
51          getline(input_file, line);
52      }
53
54      //Tomamos los datos
55      for (int i=0; i<num_ciudades; ++i){
56          input_file >> ciudad;
57          input_file >> coor_x;
58          input_file >> coor_y;
59          coordenadas.first = coor_x;
60          coordenadas.second = coor_y;
61          v_coordenadas.push_back(coordenadas);
62      }
63
64      //Calculamos las distancias y las metemos en la matriz
           ↪ que representa
65      //el grafo
66      for(int i=0; i<num_ciudades; ++i){
67          for (int j=i+1; j<num_ciudades; ++j){
68              distancias[i][j]=sqrt(pow(v_coordenadas[j].
                   ↪first - v_coordenadas[i].first, 2) +
69              pow(v_coordenadas[j].second - v_coordenadas[i
                   ↪].second, 2));
70              distancias[j][i]=distancias[i][j];
71          }
72      }
73
74      distancias.draw();
75
76      //Generamos vector de candidatos
77      for(int i=0; i<num_ciudades; ++i)
78      candidatos.push_back(i);
```

Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez


```
79
80      //Elegimos el recorrido inicial
81      int E = 0, O = 0, N = 0;
82      double mas_al_E = v_coordenadas[0].first;
83      double mas_al_O = v_coordenadas[0].first;
84      double mas_al_N = v_coordenadas[0].second;
85
86      for(int i=1; i<num_ciudades; ++i){
87          if(v_coordenadas[i].second > mas_al_N){
88              mas_al_N = v_coordenadas[i].second;
89              N = i;
90          }
91          if(v_coordenadas[i].first > mas_al_E){
92              mas_al_E = v_coordenadas[i].first;
93              E = i;
94          }
95          if(v_coordenadas[i].first < mas_al_O){
96              mas_al_O = v_coordenadas[i].first;
97              O = i;
98          }
99      }
100
101     solucion.push_back(O); candidatos[O] = -1;
102     solucion.push_back(N); candidatos[N] = -1;
103     solucion.push_back(E); candidatos[E] = -1;
104
105     int tam_solucion = solucion.size(); //Tamanio del
        ↪conjunto solucion
106
107     //Comienzo del algoritmo
108     vector<int>::iterator sol_it, cand_it;  //Iteradores
        ↪de los vectores de candidatos y solucion
109     vector<int>::iterator ciudad_origen_it; //Iterador que
        ↪ almacenara la posicion de la ciudad del
110     //conjunto solucion, que tiene mas cerca a una ciudad
111     //del conjuno candidatos
112
113     while(tam_solucion < num_ciudades){ //Mientras que no
        ↪hayamos recorrido todas las ciudades
114         //Buscamos la ciudad mas cercana al conjunto
            ↪solucion
115         int ciudad_mas_cercana = 0;
116         double distancia_mas_cercana = INF;
117
118         for(sol_it=solucion.begin(); sol_it!=solucion.end
            ↪(); ++sol_it){
119             for (cand_it=candidatos.begin(); cand_it!=
                ↪candidatos.end(); ++cand_it){
120                 if ((distancias[*sol_it][*cand_it] <
```

Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez


```
                    ↪distancia_mas_cercana) && (*cand_it
                    ↪!= -1)){
121                   ciudad_origen_it = sol_it;
122                   ciudad_mas_cercana = *cand_it;
123                   distancia_mas_cercana = distancias[*
                         ↪sol_it][*cand_it];
124                 }
125               }
126             }
127
128             //Una vez encontrada vemos en que posicion del
                   ↪conjunto solucion insertarla para minimizar
                   ↪el trayecto
129             vector<int>::iterator ciudad_siguiente_it =
                   ↪ciudad_origen_it;
130             vector<int>::iterator ciudad_anterior_it =
                   ↪ciudad_origen_it;
131             vector<int>::iterator final_it = solucion.end();
132             final_it--;
133
134             //Puesto que el recorrido es un ciclo (cerrado)
                   ↪hay que contemplar el caso de que la
135             //ciudad a insertar sea adyacente al primer o
                   ↪ultimo elemento del conjunto solucion
136             if(ciudad_origen_it == solucion.begin()){
137               ++ciudad_siguiente_it;
138               ciudad_anterior_it = final_it;
139             }
140             else if(ciudad_origen_it == final_it){
141               ciudad_siguiente_it = solucion.begin();
142               --ciudad_anterior_it;
143             }
144             else{
145               ++ciudad_siguiente_it;
146               --ciudad_anterior_it;
147             }
148
149
150             if(distancias[ciudad_mas_cercana][*
                   ↪ciudad_anterior_it] < distancias[
                   ↪ciudad_mas_cercana][*ciudad_siguiente_it]){
151               solucion.insert(ciudad_origen_it,
                     ↪ciudad_mas_cercana);
152             }
153             else{
154               solucion.insert(ciudad_siguiente_it,
                     ↪ciudad_mas_cercana);
155             }
156
```

Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez


```
157             candidatos[ciudad_mas_cercana] = -1;
158             tam_solucion++;
159         }
160 
161         //Insertamos de nuevo el primer elemento del conjunto
               ↪solucion
162         // ya que es un caamino cerrado
163         solucion.push_back(*solucion.begin());
164 
165         //Mostramos la solucion
166         cout << "Solucion: " << endl;
167 
168         for(int i=0; i<tam_solucion; ++i){
169             cout << solucion[i]+1 << " ";
170         }
171         cout << endl;
172 
173         //Calculo de la distancia total recorrida
174         double distancia_recorrida = 0.00;
175         for(int i=0; i<tam_solucion-1; ++i){
176             distancia_recorrida += distancias[i][i+1];
177         }
178 
179         cout << "Distancia total recorrida: " <<
               ↪distancia_recorrida << endl;
180 
181         //Salida de la solucion a fichero
182         ofstream output_file("data/ulysses16_insercion.txt");
183         for(int i=0; i<num_ciudades; ++i){
184             int c = solucion[i];
185             output_file << c+1 << " " << v_coordenadas[c].
                   ↪first << " " << v_coordenadas[c].second <<
                   ↪endl;
186         }
187 
188         return (0);
189 }
```

1.2.1. Pseudocódigo

El algoritmo por inserción en pseudocódigo es el siguiente:

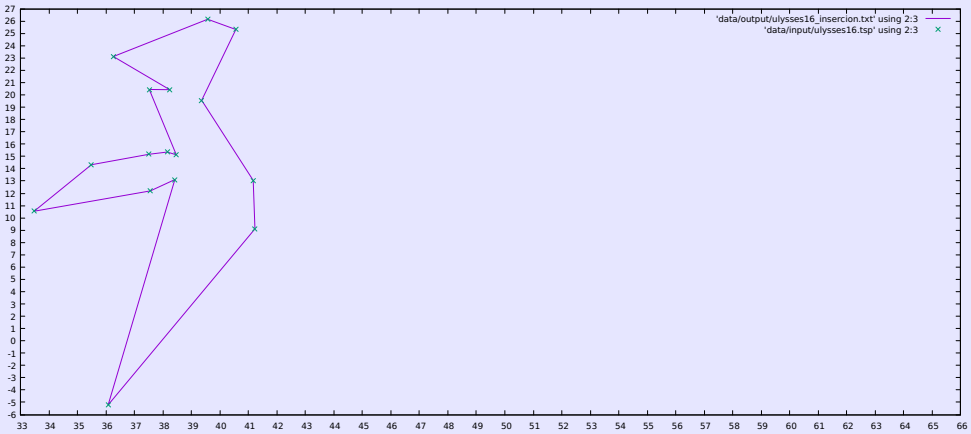
```
1 N = |V|
2 S = {ciudad mas al Norte,
3     ciudad mas al Este,
4     ciudad mas al Oeste}
5 Repetir
6 U = Buscar ciudad del conjunto V que menos aumenta la
     ↪distancia de S
```

Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez


```
Eliminar U de V
Insertar U en S
Hasta que |T| = N
Devolver T
```

Donde la búsqueda de la ciudad del conjunto V que menos aumenta la distancia de S es de orden $O(n^2)$ por lo que estaríamos hablando de un algoritmo de orden $O(n^3)$.

1.2.2. Visualización



Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez

1.3. Algoritmo con otra estrategia

1.3.1. Visualización

1.3.2. Eficiencia teórica

Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez

1.4. Comparación de algoritmos

Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez

2. Problema específico

2.1. Ahorro de gasolina

El problema trata de partir de una ciudad y llegar a otra con un vehículo con cierta autonomía pasando por el menor número de gasolineras posibles.

Para entender el algoritmo lo podemos imaginar gráficamente. La autonomía del coche va a ser el radio de la circunferencia de centro la primera ciudad o gasolinera en donde nos encontremos.

Dentro de esa circunferencia se encontrarán las gasolineras a las que podemos llegar con la autonomía del vehículo. Solo nos queda elegir a cual de ellas. Muy facil, nos vamos a la gasolinera que este más cerca de la ciudad objetivo.

Así nos vamos moviendo de gasolinera en gasolinera hasta que dentro de nuestra circunferencia se encuentre a la ciudad objetivo.

En el desarrollo de este algoritmo nos encontramos un error en tiempo de ejecución de violación de segmento. Esto se debía a que no hacíamos un clear del vector que contenía las distancias desde la posición actual hasta el resto de gasolineras.

En el código, más concretamente en la funcion “BuscarGasolinera“,recopilamos las ciudades que están dentro de la circunferencia en el vector de “indices_posibles_gasolineras” en donde guardamos los índices de las gasolineras.

Después de recopilarlas, buscamos en él, el índice de la gasolinera que minimiza la distancia a la ciudad objetivo y guardamos el índice de aquella que cumple el criterio de optimalidad.

La función devuelve el índice que será añadido al vector solución y borrado de manera lógica del vector de candidatos.

En el caso de que el índice devuelto de la función sea -1 significa que hemos llegado a un punto en el que desde la gasolinera que nos encontramos no podemos ir a ninguna otra con la autonomía dada. Para indicarlo se muestra un mensaje de error y finaliza el algoritmo.

Si el algoritmo finaliza sin mensaje de error significa que se ha conseguido llegar a la ciudad objetivo.

```
1  #include <iostream>
2  #include <fstream>
3  #include <string>
4  #include <vector>
5  #include <utility>
6  #include <cmath>
7  #include <limits>
8  #include "matriz.h"
9  #include <iomanip>
10
```

Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez


```
11 using namespace std;
12 const double INF = numeric_limits<double>::max();
13
14
15 double Distancia(const pair<double,double> & posicion_A,
   ↪const pair<double,double> &posicion_B){
16
17     double distancia = sqrt(pow(posicion_A.first -
       ↪posicion_B.first,2) + pow(posicion_A.second-
       ↪posicion_B.second,2));
18
19     return distancia;
20 }
21
22 //TODO
23 int BuscarGasolinera(const int autonomia,const int
   ↪pos_actual, const int ciudad_destino, matriz<double>
   ↪& grafo,  vector<int> &candidatos){
24
25     int parada = -1;
26     vector<int> indices_posibles_gasolineras;
27     double minimo = INF;
28
29     for (int i = 0 ;i <candidatos.size();++i){
30         double distancia_actual_candidato = grafo[
           ↪pos_actual][i];
31         double distancia_candidato_destino = grafo[i][
           ↪ciudad_destino];
32         if((distancia_actual_candidato <= autonomia) && (
           ↪candidatos[i] != -1) && (
           ↪distancia_candidato_destino < minimo)){
33             parada = i;
34             minimo = distancia_candidato_destino;
35         }
36     }
37
38     return(parada);
39 }
40
41 int main(int argc, char **argv){
42
43     string line;
44     int num_ciudades;
45
46     if(argc<5){
47         cout << "Faltan argumentos:\n" "./gasolineras [
           ↪archivo] [autonomia] [ciudad origen] [ciudad
           ↪destino]" << endl;
48     }
```

Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez


```
49
50      ifstream input_file(argv[1]);
51      double autonomia = atoi(argv[2]);
52      int ciudad_origen = atoi(argv[3]) - 1;
53      int ciudad_destino = atoi(argv[4]) - 1;
54      int pos_actual = ciudad_origen;
55
56      //Variables que usaremos para crear la matriz de
           ↪distancias (grafo)
57      int ciudad;
58      double coor_x, coor_y;
59      vector<pair<double, double> > v_coordenadas;
60      pair<double, double> p;
61
62      //Movemos el offset a la linea en la que se dice la
           ↪dimension del grafo
63      for(int i=0; i<3; ++i){
64          getline(input_file, line);
65      }
66
67      getline(input_file, line);
68      line.erase(0, 11);
69      num_ciudades = atoi(line.c_str());
70
71      //Reservamos espacio para el grafo
72      matriz<double> distancias(num_ciudades, num_ciudades,
           ↪0);
73
74
75      //Movemos el offset al comienzo de los datos
76      for(int i=0; i<3; ++i){
77          getline(input_file, line);
78      }
79
80      //Tomamos los datos
81      for (int i=0; i<num_ciudades; ++i){
82          input_file >> ciudad;
83          input_file >> coor_x;
84          input_file >> coor_y;
85          p.first = coor_x;
86          p.second = coor_y;
87          v_coordenadas.push_back(p);
88      }
89
90      //Calculamos las distancias y las metemos en la matriz
           ↪ que representa
91      //el grafo
92      for(int i=0; i<num_ciudades; ++i){
93          for (int j=i+1; j<num_ciudades; ++j){
```

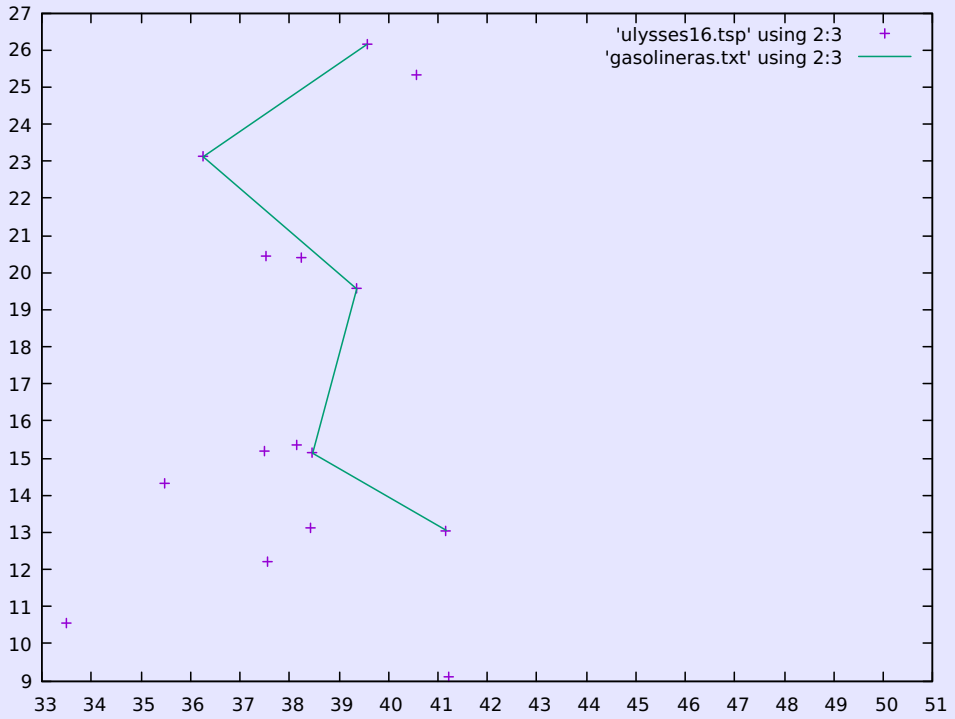
Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez


```
94              distancias[i][j]=Distancia(v_coordenadas[i],
                    ↪v_coordenadas[j]);
95              distancias[j][i]=distancias[i][j];
96          }
97      }
98
99      //Comienzo del algoritmo
100     vector<int> candidatos;
101     vector<int> solucion;
102     solucion.push_back(ciudad_origen);
103
104     for(int i = 0; i< num_ciudades; i++)
105         candidatos.push_back(i);
106
107     candidatos[ciudad_origen] = -1;
108     bool fin = false;
109     pos_actual = ciudad_origen;
110
111
112     while(fin == false){
113         if(autonomia >= distancias[pos_actual][
                ↪ciudad_destino]){
114             cout << "FIN = DESTINO" <<  endl;
115             solucion.push_back(ciudad_destino);
116             fin = true;
117         }
118
119         else{
120             pos_actual = BuscarGasolinera(autonomia,
                    ↪pos_actual, ciudad_destino, distancias,
                    ↪candidatos);
121
122             if(pos_actual != -1){
123                 candidatos[pos_actual] = -1;
124                 solucion.push_back(pos_actual);
125             }
126             else{
127                 cout << "No podemos llegar a ninguna otra
                        ↪gasolinera" << endl;
128                 fin = true;
129             }
130         }
131     }
132
133     for (int i = 0; i<solucion.size(); ++i){
134         cout << solucion[i]+1 << " --> ";
135     }
136
137     cout << "FIN" << endl;
```

Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez


```
138
139     ofstream ficherosalida("data/gasolineras.txt");
140     for (int i = 0; i < solucion.size(); ++i){
141         int c = solucion[i];
142         ficherosalida << c+1 << " " << v_coordenadas[c].
            ↪first << " " << v_coordenadas[c].second <<
            ↪endl;
143     }
144
145     return(0);
146 }
```

2.1.1. Visualización



2.1.2. Eficiencia teórica

La eficiencia teórica O(n) depende del número de ciudades que hay. Tomamos, por tanto, $TAM = num\_ciudades = n$.

La eficiencia del algoritmo, en el peor de los casos, es

$$T(n) = \sum_{i=1}^{n}\sum_{j=i}^{n}$$

$$T(n) = n * (n - i)$$

$$T(n) \in O(n^2)$$

Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez

Para hallar esto nos debemos de fijar en el bucle que comienza en la línea 112 y analizarlo. Nos damos cuenta que, en el peor de los casos, el algoritmo revisa todas las ciudades y escoge la última, y luego vuelve a revisarlas todas y escoger la última, y así consecutivamente.