

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 óptimas 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 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.

#### 1.1.1. Código del programa



```
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 const double INF = numeric_limits<double>::max();
12
13
14 /*  FUNCION
15  *   Esta funcion recibe como parametros un vector, que son
16  *   ↪ las distintas distancias
17  *   a las distintas ciudades desde una misma ciudad y el
18  *   ↪ vector candidatos para comprobar
19  *   si ya se han visitado las respectivas ciudades.
20  *
21  *   La funcion encuentra el menor elemento del vector v y
22  *   ↪ devuelve la posicion dentro
23  *   del vector de dicho elemento.
24  */
25 int BuscaMenor(vector<double> v, vector<int> & candidatos)
26     ↪{
27     int menor;
28     double minimo = INF;
29     for(int i = 0; i < v.size(); ++i){
30         if( v[i] < minimo && candidatos[i] != -1){
31             minimo = v[i];
32             menor = i;
33         }
34     }
35     return(menor);
36 }
37
38
39 int main(int argc, char **argv){
40
41     ifstream input_file(argv[1]);
42     string line;
43     int dimension;
44
45     int ciudad;
46     double coor_x, coor_y;
47
48     vector<pair<double, double> > v;
```



```
45 pair<double, double> p;
46
47 //Movemos el offset a la linea en la que se dice la
48   ↳ dimension del grafo
49 for(int i=0; i<3; ++i){
50     getline(input_file, line);
51 }
52
53 getline(input_file, line);
54 line.erase(0, 11);
55 dimension = atoi(line.c_str());
56
57 //Reservamos espacio para el grafo
58 matriz <double> m(dimension, dimension, INF);
59
60 //Movemos el offset al comienzo de los datos
61 for(int i=0; i<3; ++i){
62     getline(input_file, line);
63 }
64
65 //Tomamos los datos
66 for (int i=0; i<dimension; ++i){
67     input_file >> ciudad;
68     input_file >> coor_x;
69     input_file >> coor_y;
70     p.first = coor_x;
71     p.second = coor_y;
72     v.push_back(p);
73 }
74
75 //Calculamos las distancias y las metemos en la matriz
76   ↳ que representa
77 //el grafo
78 for(int i=0; i<dimension; ++i){
79     for (int j=i+1; j<dimension; ++j){
80         m[i][j]=sqrt(pow(v[j].first - v[i].first, 2) +
81           ↳ pow(v[j].second - v[i].second, 2));
82         m[j][i]=m[i][j];
83     }
84 }
85
86 //Algoritmo basado en cercanías. La idea es ir a la
87   ↳ ciudad mas cercana
88 //sin formar un ciclo. Exceptuando la ultima.
89
90 //Declaramos los vectores que albergaran los conjuntos
91   ↳ Candidato y Solucion
92 vector<int> solucion;
93 vector<int> candidatos;
```



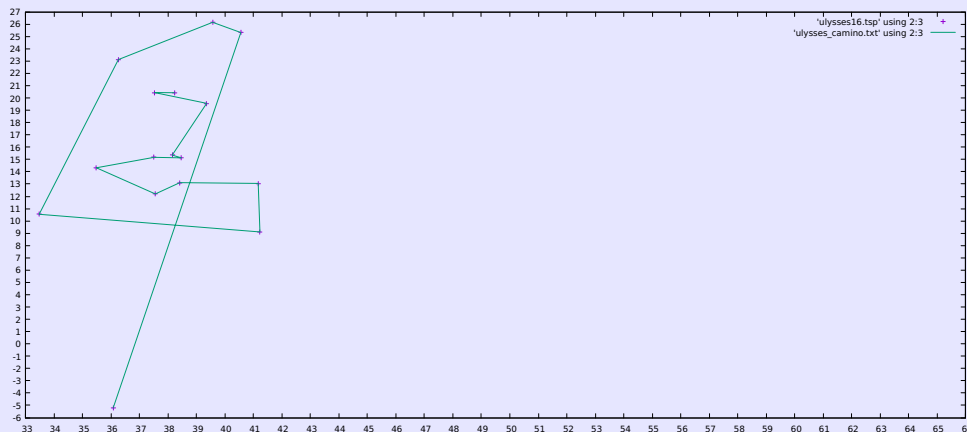
```
89
90 // Inicializamos el conjunto de candidatos, el rango
    ↳ sera [0,15].
91 for(int i = 0; i<dimension;++i){
92     candidatos.push_back(i);
93 }
94
95 // Abergamos la primera ciudad en el conjunto solucion
    ↳ .
96 int i = 0;
97 solucion.push_back(0);
98 candidatos[0] = -1;
99
100 // variable 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.
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 }

```

### 1.1.2. Visualización



### 1.1.3. 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

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

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

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



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.

## 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.

### 1.2.1. Código del programa

```
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
15  ↪ciudades de un mapa mediante un algoritmo de tipo
16  ↪greedy. En esta version del algoritmo, dado un
17  ↪recorrido inicial que contiene tres nodos en cada
18  ↪paso se busca el nodo mas cercano al conjunto
19  ↪solucion que se encuentre en el conjunto de
20  ↪candidatos. El nodo mas cercano es insertado en la
21  ↪posicion del vector que menos aumente el recorrido.
22  */
23
24 int main(int argc, char **argv){
25     const double INF = numeric_limits<double>::max();
26
27     if(argc < 2){
28         cout << "ERROR. Faltan argumentos [archivo de
29         ↪datos]" << endl;
30         exit(-1);
31     }
32
33     ifstream input_file(argv[1]);
34     string line;
35     int num_ciudades;
36
37     int ciudad;
38     double coor_x, coor_y;
39
40     vector<pair<double, double> > v_coordenadas;
```



```
32 pair<double, double> coordenadas;
33
34 vector<int> candidatos;
35 vector<int> solucion;
36
37 //Movemos el offset a la linea en la que se dice la
38   ↳ dimension del grafo
39 for(int i=0; i<3; ++i){
40     getline(input_file, line);
41 }
42
43 getline(input_file, line);
44 line.erase(0, 11);
45 num_ciudades = atoi(line.c_str());
46
47 //Reservamos espacio para el grafo
48 matriz <double> distancias(num_ciudades, num_ciudades,
49   ↳ 0);
50
51 //Movemos el offset al comienzo de los datos
52 for(int i=0; i<3; ++i){
53     getline(input_file, line);
54 }
55
56 //Tomamos los datos
57 for (int i=0; i<num_ciudades; ++i){
58     input_file >> ciudad;
59     input_file >> coor_x;
60     input_file >> coor_y;
61     coordenadas.first = coor_x;
62     coordenadas.second = coor_y;
63     v_coordenadas.push_back(coordenadas);
64 }
65
66 //Calculamos las distancias y las metemos en la matriz
67   ↳ que representa
68 //el grafo
69 for(int i=0; i<num_ciudades; ++i){
70     for (int j=i+1; j<num_ciudades; ++j){
71         distancias[i][j]=sqrt(pow(v_coordenadas[j].
72   ↳ first - v_coordenadas[i].first, 2) +
73         pow(v_coordenadas[j].second - v_coordenadas[i]
74   ↳ .second, 2));
75         distancias[j][i]=distancias[i][j];
76     }
77 }
78
79 distancias.draw();
```





```
76 //Generamos vector de candidatos
77 for(int i=0; i<num_ciudades; ++i)
78     candidatos.push_back(i);
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(0); candidatos[0] = -1;
102 solucion.push_back(N); candidatos[N] = -1;
103 solucion.push_back(E); candidatos[E] = -1;
104
105 int tam_solucion = solucion.size(); //Tamaño 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 conjunto 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!=
120             ↪candidatos.end(); ++cand_it){
121             if ((distancias[*sol_it][*cand_it] <
122                 ↪distancia_mas_cercana) && (*cand_it
123                 ↪!= -1)){
124                 ciudad_origen_it = sol_it;
125                 ciudad_mas_cercana = *cand_it;
126                 distancia_mas_cercana = distancias[*
127                     ↪sol_it][*cand_it];
128             }
129         }
130
131         //Una vez encontrada vemos en que posicion del
132         ↪conjunto solucion insertarla para minimizar
133         ↪el trayecto
134         vector<int>::iterator ciudad_siguiente_it =
135         ↪ciudad_origen_it;
136         vector<int>::iterator ciudad_anterior_it =
137         ↪ciudad_origen_it;
138         vector<int>::iterator final_it = solucion.end();
139         final_it--;
140
141         //Puesto que el recorrido es un ciclo (cerrado)
142         ↪hay que contemplar el caso de que la
143         ↪ciudad a insertar sea adyacente al primer o
144         ↪ultimo elemento del conjunto solucion
145         if(ciudad_origen_it == solucion.begin()){
146             ++ciudad_siguiente_it;
147             ciudad_anterior_it = final_it;
148         }
149         else if(ciudad_origen_it == final_it){
150             ciudad_siguiente_it = solucion.begin();
151             --ciudad_anterior_it;
152         }
153         else{
154             ++ciudad_siguiente_it;
155             --ciudad_anterior_it;
156         }
157
158         if(distancias[ciudad_mas_cercana][*
159             ↪ciudad_anterior_it] < distancias[
160             ↪ciudad_mas_cercana][*ciudad_siguiente_it]){
161             solucion.insert(ciudad_origen_it,
162                 ↪ciudad_mas_cercana);
163         }
164         else{
165             solucion.insert(ciudad_origen_it,
166                 ↪ciudad_mas_cercana);
167         }
168     }
```

```
154         solucion.insert(ciudad_siguiente_it,
155             ↪ciudad_mas_cercana);
156     }
157     candidatos[ciudad_mas_cercana] = -1;
158     tam_solucion++;
159 }
160
161 //Insertamos de nuevo el primer elemento del conjunto
162 ↪solucion
163 // ya que es un caamino cerrado
164 solucion.push_back(*solucion.begin());
165
166 //Mostramos la solucion
167 cout << "Solucion: " << endl;
168
169 for(int i=0; i<tam_solucion; ++i){
170     cout << solucion[i]+1 << " ";
171 }
172 cout << endl;
173
174 //Calculo de la distancia total recorrida
175 double distancia_recorrida = 0.00;
176 for(int i=0; i<tam_solucion-1; ++i){
177     distancia_recorrida += distancias[i][i+1];
178 }
179
180 cout << "Distancia total recorrida: " <<
181     ↪distancia_recorrida << endl;
182
183 //Salida de la solucion a fichero
184 ofstream output_file("data/ulysses16_insercion.txt");
185 for(int i=0; i<num_ciudades; ++i){
186     int c = solucion[i];
187     output_file << c+1 << " " << v_coordenadas[c].
188         ↪first << " " << v_coordenadas[c].second <<
189         ↪endl;
190 }
191
192 return (0);
193 }
```

### 1.2.2. Pseudocódigo

El algoritmo por inserción en pseudocódigo es el siguiente:

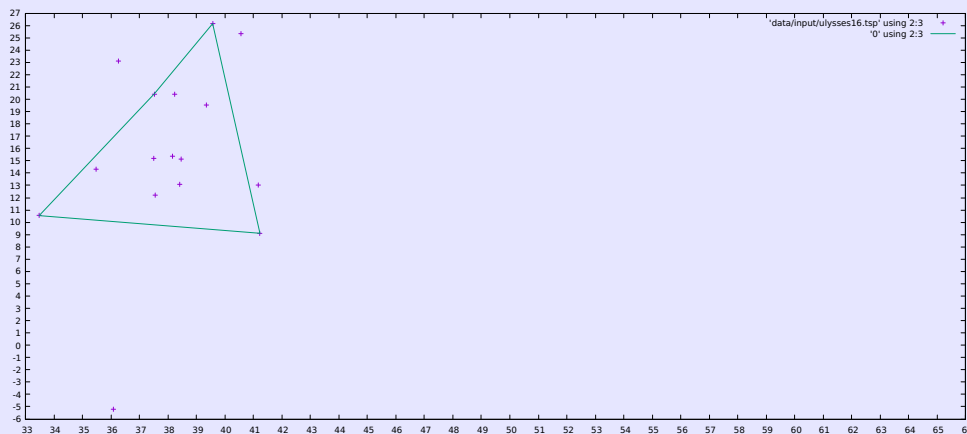
```
1  N = |V|
2  S = {ciudad mas al Norte,
3      ciudad mas al Este,
```

```
4 ciudad mas al Oeste}
5 Repetir
6 U = Buscar ciudad del conjunto V que menos aumenta la
  ↪ distancia de S
7 Eliminar U de V
8 Insertar U en S
9 Hasta que |T| = N
10 Devolver T
```

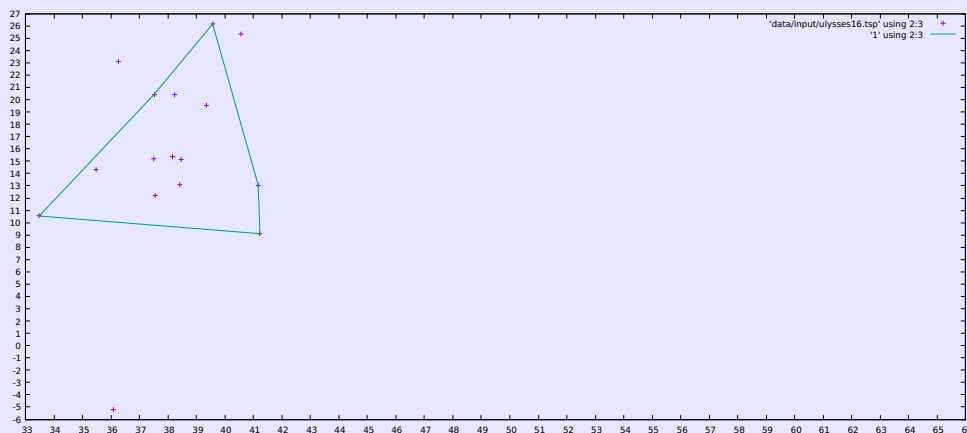
Donde la búsqueda de la ciudad del conjunto  $V$  que menos aumenta la distancia de  $S$  es de orden  $O(n^2)$  por lo que estaríamos hablando de un algoritmo de orden  $O(n^3)$ .

### 1.2.3. Visualización

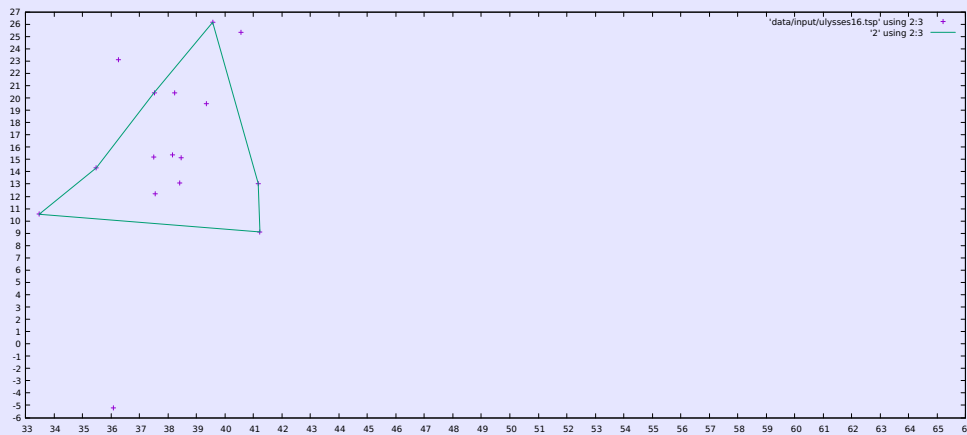
Empezamos seleccionando 3 ciudades distanciadas.



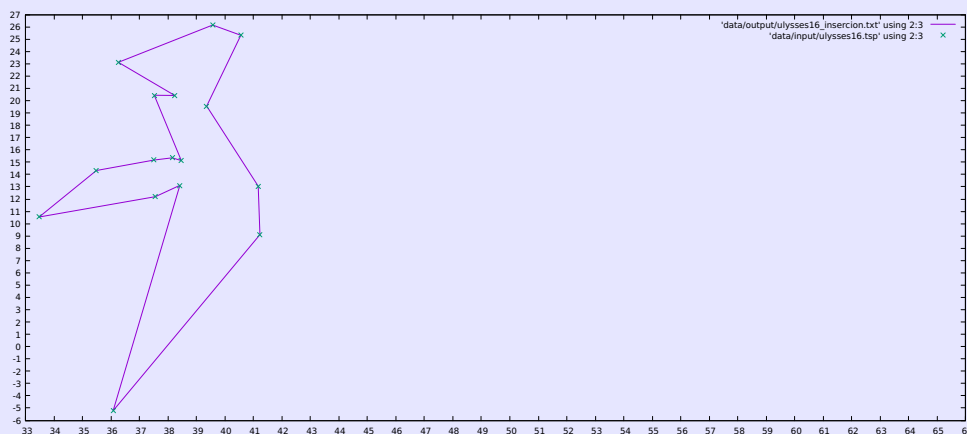
Continuamos añadiendo aquella ciudad que aumente en menor medida el recorrido total.



Como podemos comprobar, se añaden las ciudades en el lugar que hagan que el recorrido total sea menor.



Tras terminar obtenemos el recorrido final.



**1.3. Algoritmo con otra estrategia****1.3.1. Visualización****1.3.2. Eficiencia teórica**



#### 1.4. Comparación de algoritmos

## 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 en cada momento.

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 fácil, 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 función “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.

#### 2.1.1. Código del programa

```
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
11 using namespace std;
12 const double INF = numeric_limits<double>::max();
13
14
15 double Distancia(const pair<double,double> & posicion_A,
16     ↪ const pair<double,double> &posicion_B){
17
18     double distancia = sqrt(pow(posicion_A.first -
19     ↪ posicion_B.first,2) + pow(posicion_A.second-
20     ↪ posicion_B.second,2));
21
22     return distancia;
23 }
24
25 //TODO
26 int BuscarGasolinera(const int autonomia,const int
27     ↪ pos_actual, const int ciudad_destino, matriz<double>
28     ↪ & grafo, vector<int> &candidatos){
29
30     int parada = -1;
31     vector<int> indices_posibles_gasolineras;
32     double minimo = INF;
33
34     for (int i = 0 ;i <candidatos.size();++i){
35         double distancia_actual_candidato = grafo[
36     ↪ pos_actual][i];
37         double distancia_candidato_destino = grafo[i][
38     ↪ ciudad_destino];
39         if((distancia_actual_candidato <= autonomía) && (
40     ↪ candidatos[i] != -1) && (
41     ↪ distancia_candidato_destino < minimo)){
42             parada = i;
43             minimo = distancia_candidato_destino;
44         }
45     }
46
47     return(parada);
48 }
49
50 int main(int argc, char **argv){
51
52     string line;
53     int num_ciudades;
54
55     if(argc<5){
```



```
47     cout << "Faltan argumentos:\n" " ./gasolineras [
         ↪archivo] [autonomia] [ciudad origen] [ciudad
         ↪destino]" << endl;
48 }
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){
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 while(fin == false){
112     if(autonomia >= distancias[pos_actual][
    ↳ ciudad_destino]){
113         cout << "FIN = DESTINO" << endl;
114         solucion.push_back(ciudad_destino);
115         fin = true;
116     }
117
118     else{
119         pos_actual = BuscarGasolinera(autonomia,
    ↳ pos_actual, ciudad_destino, distancias,
    ↳ candidatos);
120
121         if(pos_actual != -1){
122             candidatos[pos_actual] = -1;
123             solucion.push_back(pos_actual);
124         }
125         else{
126             cout << "No podemos llegar a ninguna otra
    ↳ gasolinera" << endl;
127             fin = true;
128         }
129     }
130 }
131 }
132 }
```

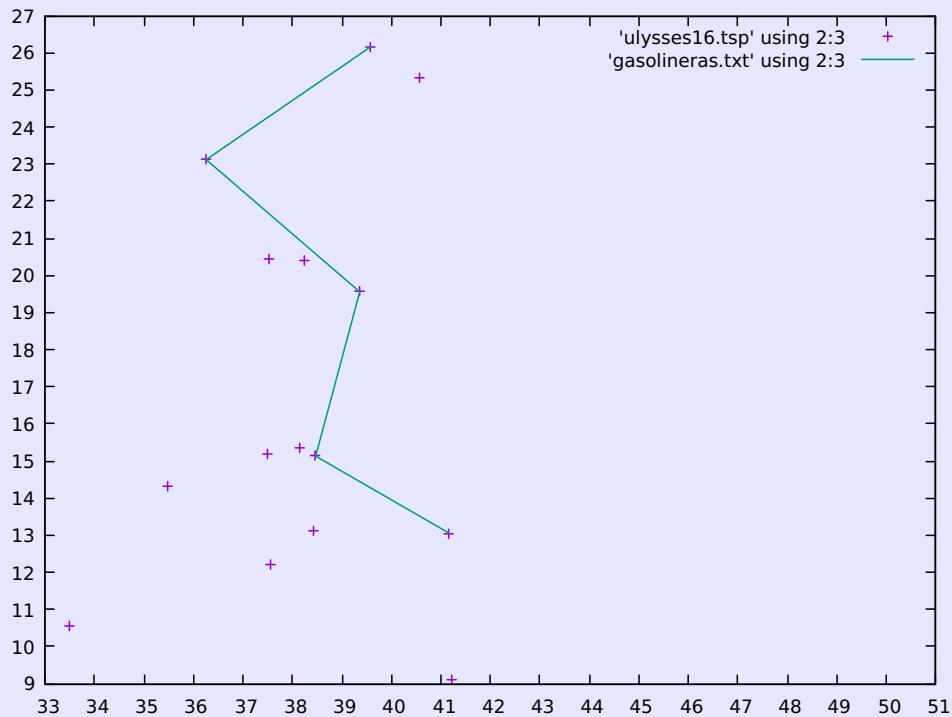


```
133     for (int i = 0; i<solucion.size(); ++i){
134         cout << solucion[i]+1 << " --> ";
135     }
136
137     cout << "FIN" << endl;
138
139     ofstream ficherosalida("data/gasolineras.txt");
140     for (int i = 0; i < solucion.size(); ++i){
141         int c = solucion[i];
142         ficherosalida << c+1 << " " << v_coordenadas[c].
            ↳first << " " << v_coordenadas[c].second <<
            ↳endl;
143     }
144
145     return(0);
146 }
```

### 2.1.2. Pseudocódigo

```
1  Mientras no se llegue al destino o a un punto sin salida:
2      Encontrar gasolineras / ciudades posibles con la
        ↳autonomia;
3      Si podemos ir a gasolineras o a la ciudad objetivo:
4          Si podemos ir a la ciudad objetivo:
5              FIN;
6          Si podemos ir a una o varias gasolineras:
7              Elegir la mas cercana a la ciudad objetivo;
8              Anadir al vector solucion;
9              Posicionarnos en la nueva gasolinera;
10     Si no podemir a ningun lado:
11         FIN;
```

## 2.1.3. Visualización



## 2.1.4. 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

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

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

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

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.