

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 | #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;
49     pair<double, double> p;
```



```
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
```



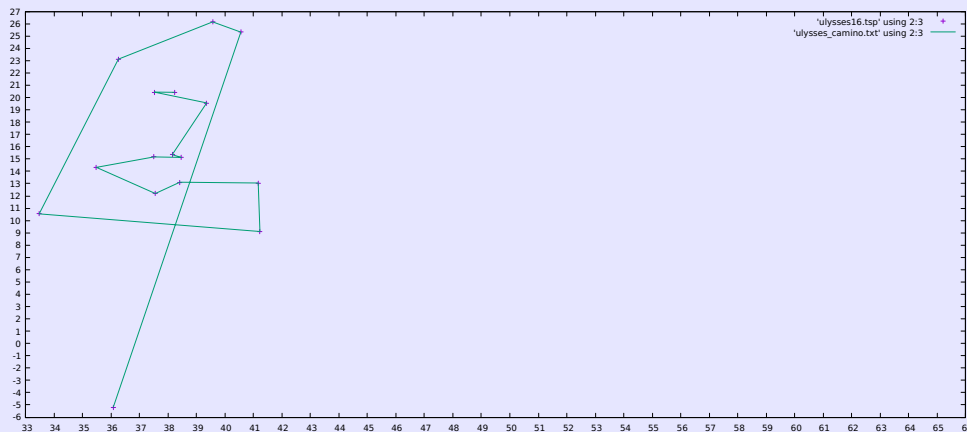
```
    ↪ sera [0,15].
91 for(int i = 0; i<dimension;++i){
92     candidatos.push_back(i);
93 }
94
95 // Abergamos la primera ciudad en el conjunto solucion
96 ↪.
97 int i = 0;
98 solucion.push_back(0);
99 candidatos[0] = -1;
100
101 // variable donde guardaremos el indice, es decir, la
102 ↪ciudad a donde nos dirigimos.
103 int menor;
104
105 /*CUERPO DEL ALGORITMO:
106 * La idea es encontrar la ciudad mas cercana haciendo
107 ↪uso de la matriz de
108 * distancias. Una vez encontrada la ciudad (indice) al
109 ↪que nos dirigimos,
110 * la posicion candidatos[indice] lo hacemos -1 para
111 ↪mostrar que esa ciudad
112 * ya la hemos visitado y introducimos el indice en el
113 ↪vector de soluciones.
114 *
115 * Para concluir, asignamos el valor del indice a la
116 ↪variable i para empezar
117 * de nuevo todo el proceso
118 */
119 while(solucion.size()< dimension ){
120     vector <double> c;
121     m.get_Fila(i,c);
122     menor =BuscaMenor(c, candidatos);
123     solucion.push_back(menor);
124     candidatos[menor] = -1;
125     i = menor;
126 }
127
128 // Imprimimos el vector solucion teniendo en cuenta
129 ↪que para la implementacion
130 // La ciudad numero 1 ha sido el indice numero 0, por
131 ↪lo tanto tenemos que sumar
132 // 1 a los valores del vector solucion.
133
134 for(int i = 0; i< solucion.size();i++){
135     cout << solucion [i] + 1<< " --> ";
136 }
137
138 // Aniadimos la ciudad inicial para indicar que
139 ↪completamos un ciclo.
```

```

129     cout << " 1 " << " FIN.";
130
131
132     // Volcamos la salida a un fichero para su
133     // ↪ visualización con la ayuda de
134     // GNUPLLOT.
135
136     ofstream ficherosalida("data/ulysses_camino.txt");
137     for (int i = 0; i < dimension; ++i){
138         int c = solucion[i];
139         ficherosalida << c+1 << " " << v[c].first << " "
140         // ↪ << v[c].second << endl;
141     }
142
143     //FIN.
144     return (0);
145 }

```

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

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

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

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



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

```
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;
41     pair<double, double> coordenadas;
42
43     vector<int> candidatos;
44     vector<int> solucion;
45
46     //Movemos el offset a la linea en la que se dice la
47     ↪dimension del grafo
48     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,
48         ↪ 0);
49
50     //Movemos el offset al comienzo de los datos
51     for(int i=0; i<3; ++i){
52         getline(input_file, line);
53     }
54
55     //Tomamos los datos
56     for (int i=0; i<num_ciudades; ++i){
57         input_file >> ciudad;
58         input_file >> coor_x;
59         input_file >> coor_y;
60         coordenadas.first = coor_x;
61         coordenadas.second = coor_y;
62         v_coordenadas.push_back(coordenadas);
63     }
64
65     //Calculamos las distancias y las metemos en la matriz
66     ↪ que representa
67     //el grafo
68     for(int i=0; i<num_ciudades; ++i){
69         for (int j=i+1; j<num_ciudades; ++j){
70             distancias[i][j]=sqrt(pow(v_coordenadas[j].
71                 ↪first - v_coordenadas[i].first, 2) +
72                 pow(v_coordenadas[j].second - v_coordenadas[i]
73                 ↪).second, 2));
74             distancias[j][i]=distancias[i][j];
75         }
76     }
77
78     distancias.draw();
79
80     //Generamos vector de candidatos
81     for(int i=0; i<num_ciudades; ++i)
82         candidatos.push_back(i);
83
84     //Elegimos el recorrido inicial
85     int E = 0, O = 0, N = 0;
86     double mas_al_E = v_coordenadas[0].first;
87     double mas_al_O = v_coordenadas[0].first;
```



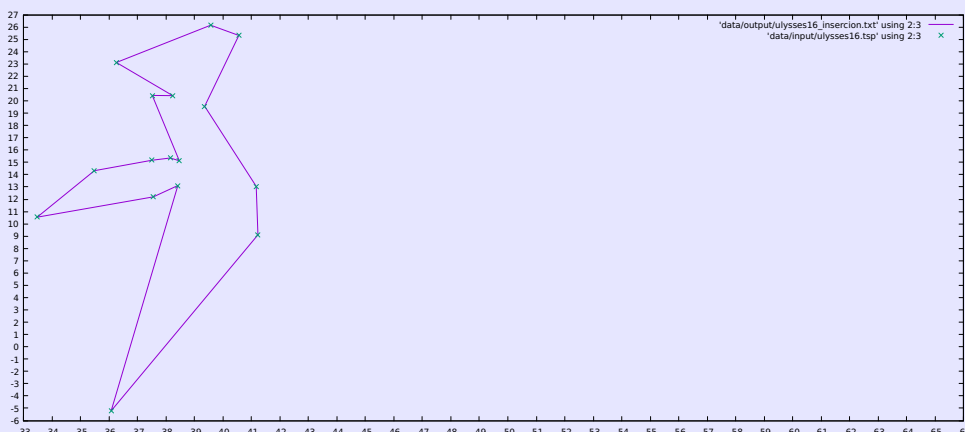
```
84     double mas_al_N = v_coordenadas[0].second;
85
86     for(int i=1; i<num_ciudades; ++i){
87         if(v_coordenadas[i].second > mas_al_N){
88             mas_al_N = v_coordenadas[i].second;
89             N = i;
90         }
91         if(v_coordenadas[i].first > mas_al_E){
92             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(); //Tamano del
106     ↪conjunto solucion
107
108     //Comienzo del algoritmo
109     vector<int>::iterator sol_it, cand_it; //Iteradores
110     ↪de los vectores de candidatos y solucion
111     vector<int>::iterator ciudad_origen_it; //Iterador que
112     ↪almacenara la posicion de la ciudad del
113     //conjunto solucion, que tiene mas cerca a una ciudad
114     //del conjunto candidatos
115
116     while(tam_solucion < num_ciudades){ //Mientras que no
117     ↪hayamos recorrido todas las ciudades
118         //Buscamos la ciudad mas cercana al conjunto
119         ↪solucion
120         int ciudad_mas_cercana = 0;
121         double distancia_mas_cercana = INF;
122
123         for(sol_it=solucion.begin(); sol_it!=solucion.end()
124         ↪(); ++sol_it){
125             for (cand_it=candidatos.begin(); cand_it!=
126             ↪candidatos.end(); ++cand_it){
127                 if ((distancias[*sol_it][*cand_it] <
128                 ↪distancia_mas_cercana) && (*cand_it
129                 ↪!= -1)){
130                     ciudad_origen_it = sol_it;
131                     ciudad_mas_cercana = *cand_it;
```



```
123         distancia_mas_cercana = distancias[*  
124             ↪sol_it][*cand_it];  
125     }  
126 }  
127  
128 //Una vez encontrada vemos en que posicion del  
129 ↪conjunto solucion insertarla para minimizar  
130 ↪el trayecto  
131 vector<int>::iterator ciudad_siguiente_it =  
132 ↪ciudad_origen_it;  
133 vector<int>::iterator ciudad_anterior_it =  
134 ↪ciudad_origen_it;  
135 vector<int>::iterator final_it = solucion.end();  
136 final_it--;  
137  
138 //Puesto que el recorrido es un ciclo (cerrado)  
139 ↪hay que contemplar el caso de que la  
140 //ciudad a insertar sea adyacente al primer o  
141 ↪ultimo elemento del conjunto solucion  
142 if(ciudad_origen_it == solucion.begin()){  
143     ++ciudad_siguiente_it;  
144     ciudad_anterior_it = final_it;  
145 }  
146 else if(ciudad_origen_it == final_it){  
147     ciudad_siguiente_it = solucion.begin();  
148     --ciudad_anterior_it;  
149 }  
150 else{  
151     ++ciudad_siguiente_it;  
152     --ciudad_anterior_it;  
153 }  
154  
155 if(distancias[ciudad_mas_cercana][*  
156     ↪ciudad_anterior_it] < distancias[  
157     ↪ciudad_mas_cercana][*ciudad_siguiente_it]){  
158     solucion.insert(ciudad_origen_it,  
159         ↪ciudad_mas_cercana);  
160 }  
161 else{  
162     solucion.insert(ciudad_siguiente_it,  
163         ↪ciudad_mas_cercana);  
164 }  
165  
166 candidatos[ciudad_mas_cercana] = -1;  
167 tam_solucion++;  
168 }  
169  
170 }
```

```
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. Visualización





1.2.2. 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

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

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

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

Para hallar esto nos debemos de fijar en el bucle que comienza en la línea 113 y analizarlo. Nos damos cuenta que el algoritmo revisa todas las ciudades para quedarse con la que aumenta el recorrido total en menor medida. Y eso lo realiza con cada ciudad nueva que se añade al bucle.

**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.

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 "índices_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
```



```
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){
56         cout << "Faltan argumentos:\n" " ./gasolineras [
57         ↪archivo] [autonomía] [ciudad origen] [ciudad
58         ↪destino]" << endl;
59     }
60 }
```




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



```
94         distancias[i][j]=Distancia(v_coordenadas[i],
95             ↪v_coordenadas[j]);
96         distancias[j][i]=distancias[i][j];
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][
113         ↪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,
121             ↪pos_actual, ciudad_destino, distancias,
122             ↪candidatos);
123
124         if(pos_actual != -1){
125             candidatos[pos_actual] = -1;
126             solucion.push_back(pos_actual);
127         }
128         else{
129             cout << "No podemos llegar a ninguna otra
130                 ↪gasolinera" << endl;
131             fin = true;
132         }
133     }
134 }
135
136 for (int i = 0; i<solucion.size(); ++i){
137     cout << solucion[i]+1 << " --> ";
138 }
139
140 cout << "FIN" << endl;
```

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



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.