

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

```
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  *   */
```



```
19 * La funcion encuentra el menor elemento del vector v y
    ↳ devuelve la posicion dentro
20 * del vector de dicho elemento.
21 */
22 int BuscaMenor(vector<double> v, vector<int> & candidatos)
    ↳{
23     int menor;
24     double minimo = INF;
25     for(int i = 0; i < v.size(); ++i){
26         if( v[i] < minimo && candidatos[i] != -1){
27             minimo = v[i];
28             menor = i;
29         }
30     }
31     return(menor);
32 }
33
34
35 int main(int argc, char **argv){
36
37     ifstream input_file(argv[1]);
38     string line;
39     int dimension;
40
41     int ciudad;
42     double coor_x, coor_y;
43
44     vector<pair<double, double> > v;
45     pair<double, double> p;
46
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
75     ↪ que representa
76     //el grafo
77     for(int i=0; i<dimension; ++i){
78         for (int j=i+1; j<dimension; ++j){
79             m[i][j]=sqrt(pow(v[j].first - v[i].first, 2) +
80                 ↪ pow(v[j].second - v[i].second, 2));
81             m[j][i]=m[i][j];
82         }
83     }
84
85     //Algoritmo basado en cercanías. La idea es ir a la
86     ↪ ciudad mas cercana
87     //sin formar un ciclo. Exceptuando la ultima.
88
89     //Declaramos los vectores que albergaran los conjuntos
90     ↪ Candidato y Solucion
91     vector<int> solucion;
92     vector<int> candidatos;
93
94     // Inicializamos el conjunto de candidatos, el rango
95     ↪ sera [0,15].
96     for(int i = 0; i<dimension;++i){
97         candidatos.push_back(i);
98     }
99
100    // Abergamos la primera ciudad en el conjunto solucion
101    ↪.
102    int i = 0;
103    solucion.push_back(0);
104    candidatos[0] = -1;
105
106    // variable donde guardaremos el indice, es decir, la
107    ↪ ciudad a donde nos dirigimos.
108    int menor;
109
110    /*CUERPO DEL ALGORITMO:
111    * La idea es encontrar la ciudad mas cercana haciendo
112    ↪ 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 }
```



## 1.2. 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(); //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 posición 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!=
        ↪candidatos.end(); ++cand_it){
120                 if ((distancias[*sol_it][*cand_it] <
        ↪distancia_mas_cercana) && (*cand_it
        ↪!= -1)){
121                     ciudad_origen_it = sol_it;
122                     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 }
```





```
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 }
```



### 1.3. Algoritmo con otra estrategia



#### 1.4. Comparación de algoritmos



## 2. Problema específico - Ahorro de gasolina