UNIVERSIDAD DE GRANADA

Práctica 3

Algoritmos Voraces (Greedy)

Víctor José Rubia López
B3

Fecha de entrega 27/04/2020

Contenido

Capítulo 1: Problema del viajante de comercio

1.1. Descripción general

El problema del TSP (*Travelling Salesman Problem*), consiste en un conjunto de ciudades y distancias hacia un número finito de ciudades. Su objetivo es visitar todas las ciudades del mapa como máximo una vez y regresar al punto de partida.

Obtendremos el recorrido en función del método voraz que utilicemos. Tres posibles enfoques son:

1.1.1. Basado en cercanía

Es un tipo de estrategia que se basa en el método del vecino más cercano (dada una ciudad inicial, sólo avanzamos a la siguiente en función de la acción que tenga menor coste).

```
basura, ciudad_actual=0: integer
for i=0..n: integer
begin
        distancias[i][ciudad_actual]=0
end

resul: vector<integer>
resul.push_back(ciudad_actual+1)
for i=0..n-1
begin
        sig, distancia=max(): integer
        for j=0..n: integer
        begin
                if(distancias[ciudad_actual][j] < distancia && distancias[ciudad_actual][j] != 0)
                begin
                        distancia = distancias[ciudad_actual][j]
                        sig= j
                end
        end
        resul.push_back(sig+1)
        for j=0..n
        begin
                distancias[j][sig]=0
        end
        ciudad_actual= sig
end
resul.push_back(1)
return resul
```

1: Pseudocódigo cercanía

He utilizado dos bucles anidados para calcular el vector de ciudades. Para ello primero hemos hecho un bucle que va de 0 hasta n (n: número de ciudades), en el cual comprobamos si la distancia entre la ciudad actual y la ciudad vecina (j) es menor que la distancia inicializada al máximo valor y si esa distancia es distinta de cero. Si es así, la distancia entre estas ciudades pasa a guardarse en la variable distancia y guardando la ciudad vecina en sig.

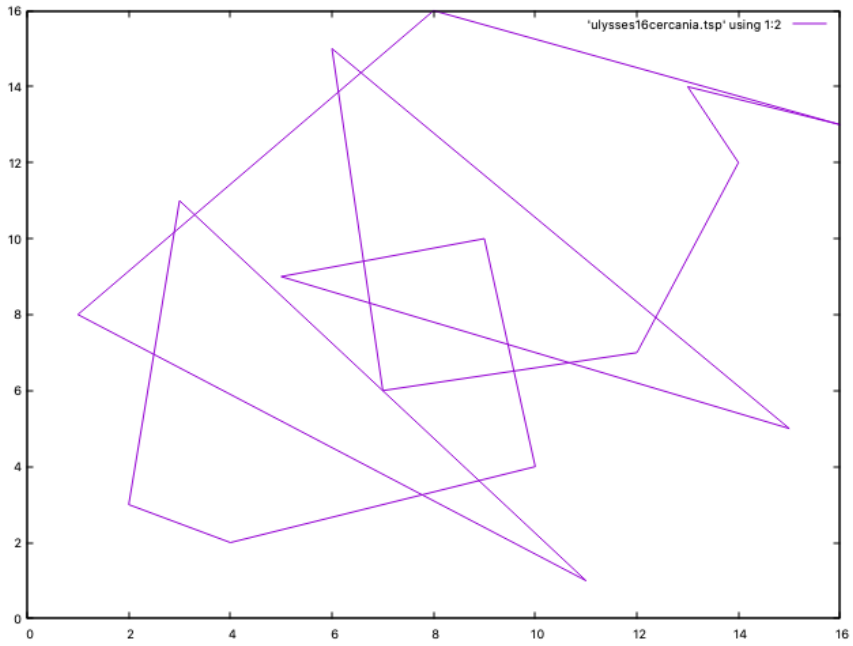
Este bucle lo anido dentro de otro que va desde 0 a n-1. En este bucle lo único que hago es inicializar las variables *sig*, *distancia* y el *vector*

con la ciudad actual. Este enfoque suele devolver, para N ciudades distribuidas aleatoriamente, un camino un 25% más largo que el camino óptimo. Sin embargo, puede ocurrir también que, debido a la distribución de las ciudades, el algoritmo retorne el peor camino.

1.1.1.1. Escenarios de ejecución

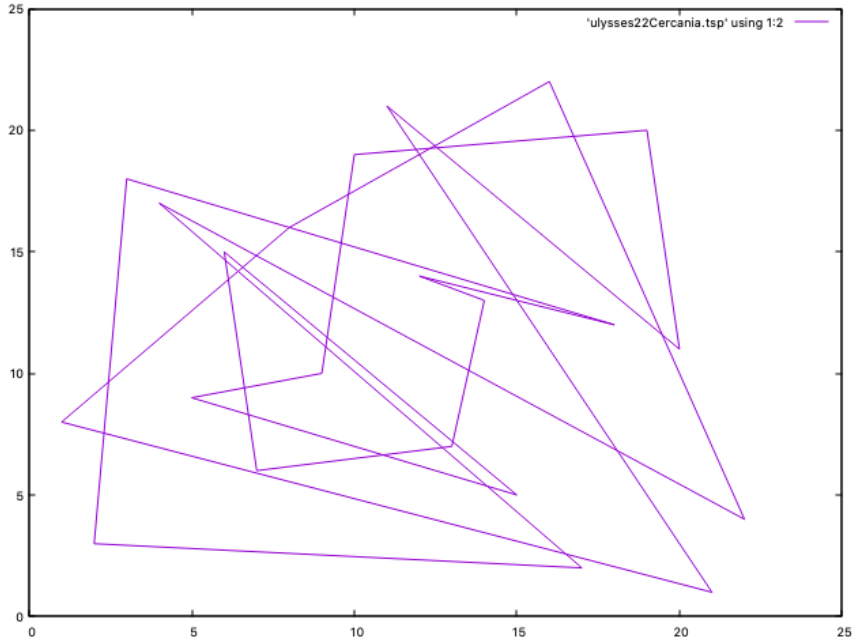
Los códigos que he utilizado para la realización de la práctica están adjuntos en el *zip*. Para su compilación, puede copiar las líneas que aparecen al principio de los 2 códigos.

Ulysses16



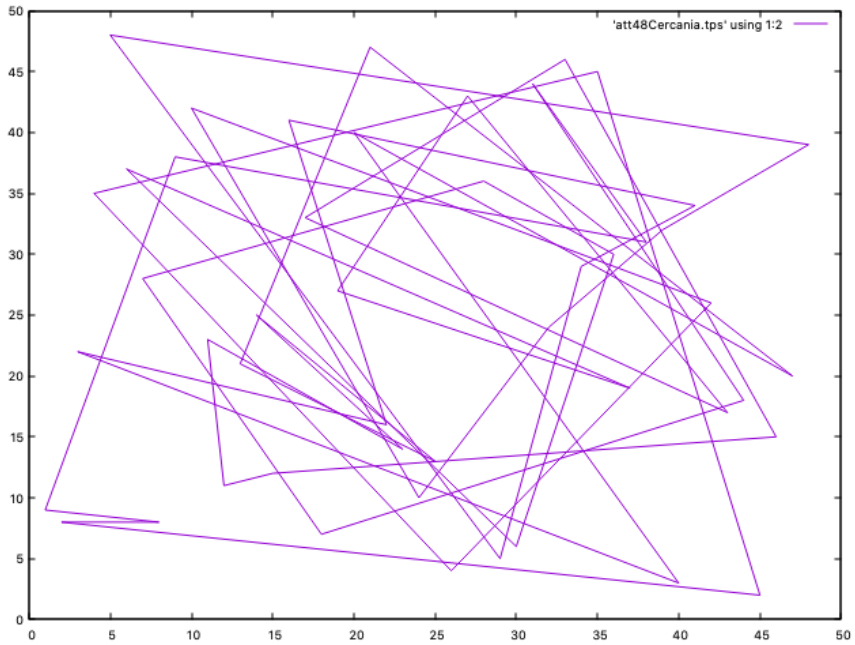

```
ALGORITMO DE CERCANÍA:
1 -> 8 -> 16 -> 13 -> 14 -> 12 -> 7 -> 6 -> 15 -> 5 -> 9 -> 10 -> 4 -> 2 -> 3 -> 11 -> 1
El coste del camino es: 104
```

Ulysses22



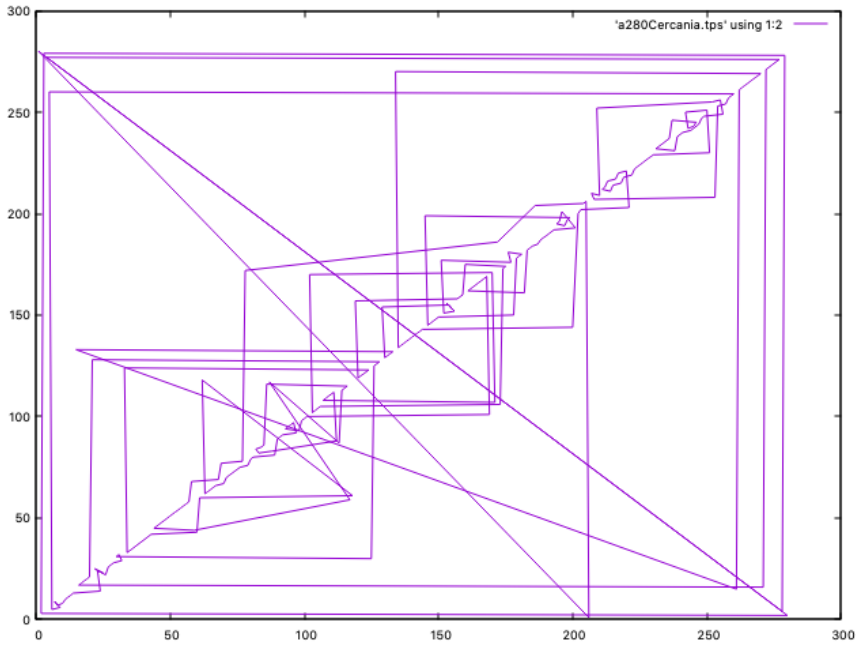

```
ALGORITMO DE CERCANÍA:
1 -> 8 -> 16 -> 22 -> 4 -> 17 -> 2 -> 3 -> 18 -> 12 -> 14 -> 13 -> 7 -> 6 -> 15 -> 5 -> 9 ->
10 -> 19 -> 20 -> 11 -> 21 -> 1
El coste del camino es: 98
```

att48




```
ALGORITMO DE CERCANÍA:
1 -> 9 -> 38 -> 31 -> 44 -> 18 -> 7 -> 28 -> 36 -> 30 -> 6 -> 37 -> 19 -> 27 -> 43 -> 17 ->
33 -> 46 -> 15 -> 12 -> 11 -> 23 -> 14 -> 25 -> 13 -> 21 -> 47 -> 20 -> 40 -> 3 -> 22 -> 16 -
> 41 -> 34 -> 29 -> 5 -> 48 -> 39 -> 32 -> 24 -> 10 -> 42 -> 26 -> 4 -> 35 -> 45 -> 2 -> 8 ->
1
El coste del camino es: 40583
```

a280




```
ALGORITMO DE CERCANÍA:
1 -> 280 -> 2 -> 3 -> 279 -> 278 -> 4 -> 277 -> 276 -> 275 -> 274 -> 273 -> 272 -> 271 -> 16
-> 17 -> 18 -> 19 -> 20 -> 21 -> 128 -> 127 -> 126 -> 125 -> 30 -> 31 -> 32 -> 29 -> 28 -> 27
-> 26 -> 22 -> 25 -> 23 -> 24 -> 14 -> 13 -> 12 -> 11 -> 10 -> 8 -> 7 -> 9 -> 6 -> 5 -> 260 -
> 259 -> 258 -> 257 -> 254 -> 253 -> 208 -> 207 -> 210 -> 209 -> 252 -> 255 -> 256 -> 249 ->
248 -> 247 -> 244 -> 241 -> 240 -> 239 -> 238 -> 231 -> 232 -> 233 -> 234 -> 235 -> 236 -> 237
-> 246 -> 245 -> 243 -> 242 -> 250 -> 251 -> 230 -> 229 -> 228 -> 227 -> 226 -> 225 -> 224 ->
223 -> 222 -> 219 -> 218 -> 215 -> 214 -> 211 -> 212 -> 213 -> 216 -> 217 -> 220 -> 221 -> 203
-> 202 -> 200 -> 144 -> 143 -> 142 -> 141 -> 140 -> 139 -> 138 -> 137 -> 136 -> 135 -> 134 ->
270 -> 269 -> 268 -> 267 -> 266 -> 265 -> 264 -> 263 -> 262 -> 261 -> 15 -> 133 -> 132 -> 131
-> 130 -> 129 -> 154 -> 155 -> 153 -> 156 -> 152 -> 151 -> 177 -> 176 -> 181 -> 180 -> 179 ->
178 -> 150 -> 149 -> 148 -> 147 -> 146 -> 145 -> 199 -> 198 -> 197 -> 194 -> 195 -> 196 -> 201
-> 193 -> 192 -> 191 -> 190 -> 189 -> 188 -> 187 -> 185 -> 184 -> 183 -> 182 -> 161 -> 162 ->
163 -> 164 -> 165 -> 166 -> 167 -> 168 -> 169 -> 101 -> 100 -> 99 -> 98 -> 93 -> 94 -> 95 ->
96 -> 97 -> 92 -> 91 -> 90 -> 89 -> 81 -> 80 -> 79 -> 76 -> 75 -> 74 -> 73 -> 72 -> 71 -> 70
-> 67 -> 66 -> 65 -> 64 -> 63 -> 62 -> 118 -> 61 -> 60 -> 43 -> 42 -> 41 -> 40 -> 39 -> 38 ->
37 -> 36 -> 35 -> 34 -> 33 -> 124 -> 123 -> 122 -> 121 -> 120 -> 119 -> 157 -> 158 -> 159 ->
160 -> 175 -> 174 -> 173 -> 106 -> 105 -> 104 -> 103 -> 102 -> 170 -> 171 -> 107 -> 108 -> 109
-> 110 -> 111 -> 112 -> 88 -> 83 -> 82 -> 84 -> 85 -> 86 -> 116 -> 115 -> 114 -> 113 -> 87 ->
117 -> 59 -> 44 -> 45 -> 46 -> 47 -> 48 -> 49 -> 50 -> 51 -> 52 -> 53 -> 54 -> 55 -> 56 -> 57
-> 58 -> 68 -> 69 -> 77 -> 78 -> 172 -> 186 -> 204 -> 205 -> 206 -> 1
El coste del camino es: 3157
```

1.1.2. Basado en inserción

Este método consiste en los siguientes pasos:

1º. Tomamos los tres puntos más alejados del mapa de ciudades formando un triángulo y medimos las distancias de sus lados.

2º. Vamos añadiendo ciudades de la siguiente forma:

   2.1º. Calculamos la distancia desde cada vértice del triángulo a ese nodo.
   2.2º. Una vez que tenemos la distancia más corta a la ciudad, eliminamos el lado asociado a ella y unimos el vértice con la ciudad que acabamos de añadir, resultando en una figura con un lado más.

3º. Repetimos el proceso hasta haber añadido todas las ciudades.

```
ciudadN, ciudadO, ciudadE: integer
cerrados: vector<integer>
abiertos: queue<integer>

[........inicialización.........]

while(abiertos.size() != 0)
begin
        sol, distAux= max(), nodoAux, ciudadA= abiertos.front():integer
        abiertos.pop()
        for i=0..cerrados.size(): integer
        begin
                sol=                    ((distancias[cerrados[i]][ciudadA]                    +
distancias[cerrados[(i+1)%cerrados.size()]][ciudadA])                                         -
distancias[cerrados[i]][cerrados[(i+1)%cerrados.size()]])
                if(sol< distAux)
                begin
                        distAux= sol;
                        nodoAux= i;
                end
        end

        i=0: integer
        it= cerrados.begin(): auto
        while(I!= nodoAux+1)
        begin
                ++it
                i++
        end
        cerrados.insert(it, ciudadA)
end

for i=0..n: integer
begin
        resul.push_back(cerrados[i]+1)
end
result.push_back(cerrados[0])
return result
```

2: Pseudocódigo inserción

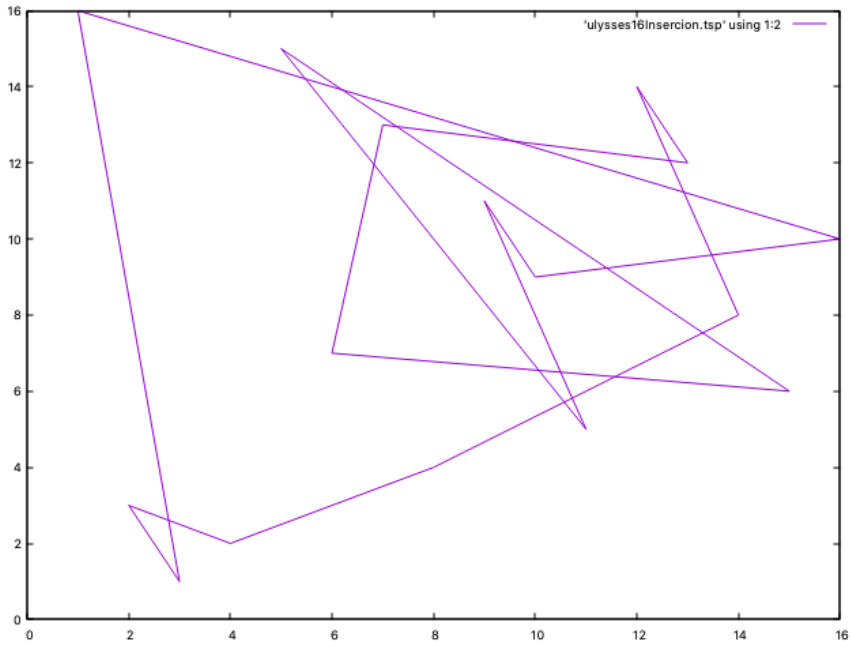
En este caso, hemos tomado las ciudades más al este, norte y oeste como referencia, conectándolas y calculando las distancias entre ellas. A continuación, cogemos una ciudad de la lista de *abiertos*. Calculamos las distancias entre esa nueva ciudad y el resto de las ciudades que se encuentran en *cerrados*. De todas esas distancias, nos quedamos con las

dos más pequeñas y conectamos las tres ciudades correspondientes, quitando la conexión entre las dos ciudades que se hallan en cerrados. Tras esto, metemos la nueva ciudad en *cerrados* y buscamos añadir otra ciudad de *abiertos* a *cerrados*.

1.1.2.1. Escenarios de ejecución

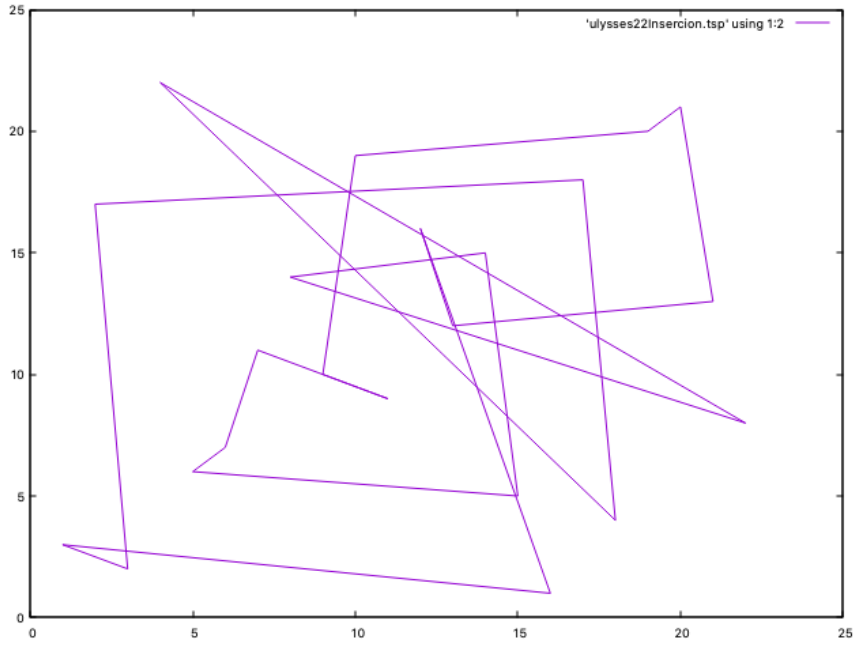
Los códigos que he utilizado para la realización de la práctica están adjuntos en el zip. Para su compilación, puede copiar las líneas que aparecen al principio de los 2 códigos.

Ulysses16



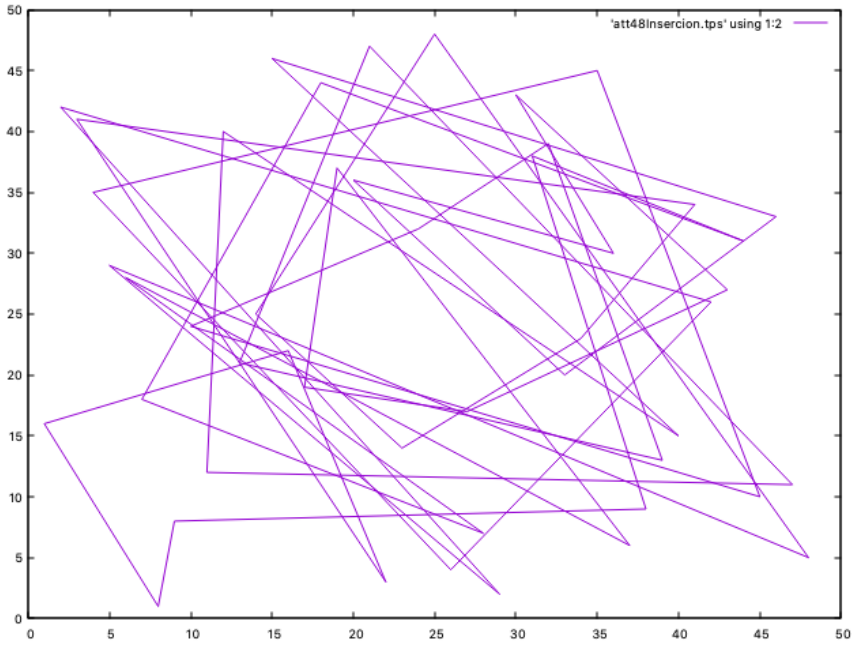

```
ALGORITMO DE INSERCION:
5 -> 15 -> 6 -> 7 -> 13 -> 12 -> 14 -> 8 -> 4 -> 2 -> 3 -> 1 -> 16 -> 10 -> 9 -> 11 -> 5
El coste del camino es: 83
```

Ulysses22



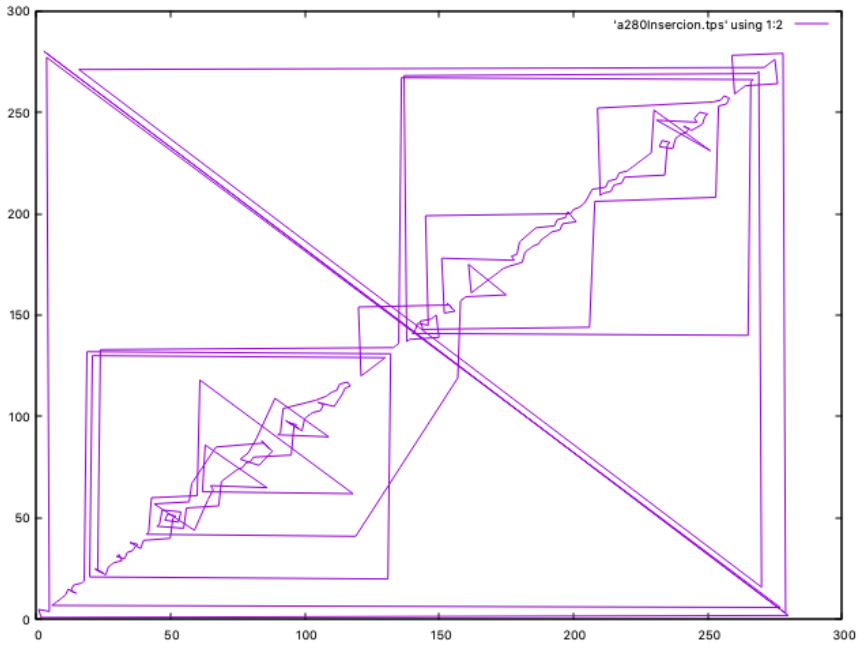

```
ALGORITMO DE INSERCION:
5 -> 6 -> 7 -> 11 -> 9 -> 10 -> 19 -> 20 -> 21 -> 13 -> 12 -> 16 -> 1 -> 3 -> 2 -> 17 -> 18 -
> 4 -> 22 -> 8 -> 14 -> 15 -> 5
El coste del camino es: 87
```

att48




```
ALGORITMO DE INSERCION:
45 -> 10 -> 24 -> 32 -> 39 -> 13 -> 21 -> 47 -> 11 -> 12 -> 40 -> 15 -> 46 -> 33 -> 20 -> 36 -> 30
-> 43 -> 27 -> 17 -> 19 -> 37 -> 6 -> 28 -> 7 -> 18 -> 44 -> 31 -> 38 -> 9 -> 8 -> 1 -> 16 -> 22
-> 3 -> 41 -> 34 -> 23 -> 14 -> 25 -> 48 -> 5 -> 29 -> 2 -> 42 -> 26 -> 4 -> 35 -> 45
El coste del camino es: 42019
```

a280




```
ALGORITMO DE INSERCION:
69 -> 68 -> 56 -> 55 -> 45 -> 46 -> 47 -> 54 -> 53 -> 48 -> 49 -> 52 -> 51 -> 50 -> 40 -> 39
-> 35 -> 38 -> 37 -> 36 -> 34 -> 33 -> 30 -> 31 -> 32 -> 29 -> 28 -> 27 -> 26 -> 22 -> 25 ->
23 -> 24 -> 133 -> 134 -> 135 -> 136 -> 267 -> 266 -> 265 -> 140 -> 141 -> 142 -> 146 -> 145
-> 199 -> 200 -> 202 -> 203 -> 204 -> 205 -> 207 -> 212 -> 213 -> 216 -> 217 -> 220 -> 221 ->
222 -> 223 -> 224 -> 225 -> 226 -> 227 -> 228 -> 229 -> 230 -> 251 -> 231 -> 246 -> 245 -> 247
-> 250 -> 249 -> 248 -> 244 -> 241 -> 243 -> 242 -> 240 -> 239 -> 238 -> 237 -> 232 -> 233 ->
236 -> 235 -> 234 -> 219 -> 218 -> 215 -> 214 -> 211 -> 210 -> 209 -> 252 -> 255 -> 256 -> 258
-> 257 -> 254 -> 253 -> 208 -> 206 -> 144 -> 143 -> 147 -> 148 -> 149 -> 150 -> 139 -> 138 ->
137 -> 268 -> 269 -> 270 -> 16 -> 271 -> 272 -> 273 -> 274 -> 275 -> 276 -> 264 -> 263 -> 262
-> 261 -> 260 -> 259 -> 278 -> 279 -> 3 -> 280 -> 2 -> 1 -> 5 -> 4 -> 277 -> 6 -> 7 -> 8 -> 9
-> 10 -> 11 -> 12 -> 15 -> 13 -> 14 -> 17 -> 18 -> 19 -> 132 -> 131 -> 20 -> 21 -> 130 -> 129
-> 128 -> 127 -> 126 -> 125 -> 124 -> 123 -> 122 -> 121 -> 120 -> 154 -> 155 -> 153 -> 156 ->
152 -> 151 -> 178 -> 177 -> 179 -> 180 -> 186 -> 193 -> 194 -> 197 -> 198 -> 201 -> 196 -> 195
-> 192 -> 191 -> 190 -> 189 -> 188 -> 187 -> 185 -> 184 -> 183 -> 182 -> 181 -> 176 -> 174 ->
173 -> 172 -> 171 -> 170 -> 169 -> 168 -> 167 -> 166 -> 165 -> 164 -> 163 -> 162 -> 161 -> 175
-> 160 -> 159 -> 158 -> 157 -> 119 -> 41 -> 42 -> 43 -> 60 -> 61 -> 118 -> 62 -> 63 -> 86 ->
65 -> 66 -> 64 -> 59 -> 44 -> 57 -> 58 -> 67 -> 85 -> 87 -> 84 -> 88 -> 83 -> 76 -> 79 -> 82
-> 89 -> 109 -> 90 -> 91 -> 92 -> 104 -> 108 -> 110 -> 112 -> 113 -> 116 -> 117 -> 115 -> 114
-> 111 -> 105 -> 107 -> 106 -> 103 -> 102 -> 101 -> 100 -> 99 -> 93 -> 98 -> 94 -> 97 -> 96 -
> 95 -> 81 -> 80 -> 78 -> 77 -> 75 -> 74 -> 73 -> 72 -> 71 -> 70 -> 69
El coste del camino es: 3430
```

1.1.3. Estrategia voraz basada en intercambio

He decidido que la nueva estrategia voraz para implementar el problema del viajante de comercio sea el método de intercambio.

Este método consiste en que, dado un mapa de ciudades, vamos intercambiando ciudades de sitio y si el coste para llegar a esa ciudad es menor que la ciudad anterior, guardamos la ciudad en el sitio de la ciudad que estaba anteriormente.

```
aux, pos, coste_aux: integer
resul, s_aux: vector<integer>
mejor_enc, cambio=true: boolean
mejor_coste: double

resul= algoritmo_inserccion(distancias, ciudades, n)
mejor_coste=calcularCostePlan(reul, distancias);

while(cambio)
begin
    cambio=false
    for i=0..resul.size()-1
    begin
        mejor_enc=false;
        for j=i+1..resul.size()
        begin
            s_aux=resul
            aux= s_aux[i]
            s_aux[i] = s_aux[j]
            s_aux[j] = aux
            coste_aux=calcularCostePlan(s_aux, distancias)
            if(coste_aux < mejor_coste)
            begin
                mejor_enc=true
                pos=j
                mejor_coste=coste_aux
            end
        end
        if(mejor_enc)
        begin
            cambio=true
            aux=resul[i]
            resul[i]= resul[pos]
            resul[pos]=aux
        end
    end
end

return resul
```

3: Pseudocódigo intercambio

En este caso se opta por intentar mejorar los caminos resultantes del algoritmo basado en inserción. En primer lugar, se llama a la función que realiza dicho algoritmo y calculamos su coste.

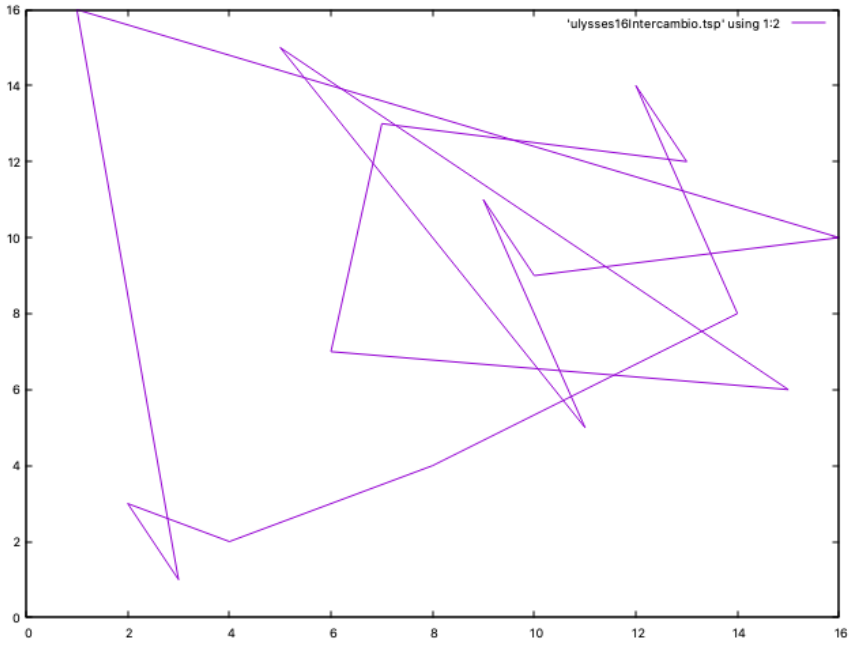
Después se realiza un *while* con dos *for* anidados donde se intercambia de manera preventiva las ciudades y se calcula su coste. Si es menor que el calculado con el algoritmo de inserción, se realiza el cambio definitivamente.

Finalmente, se devuelve un vector resultado que, en ocasiones, es mejor que el resultado que se obtiene con el algoritmo de inserción.

1.1.3.1. Escenarios de ejecución

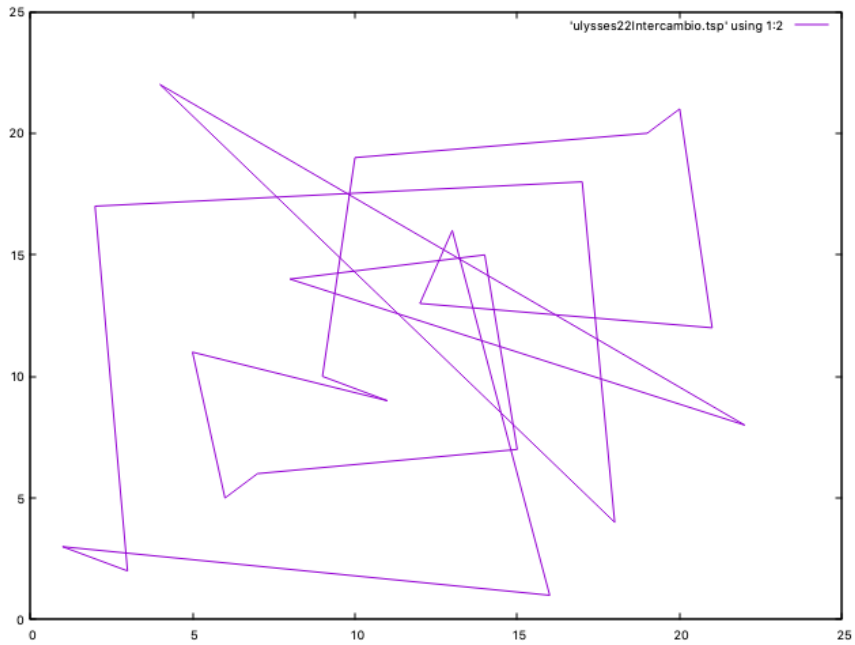
Los códigos que he utilizado para la realización de la práctica están adjuntos en el *zip*. Para su compilación, puede copiar las líneas que aparecen al principio de los 2 códigos.

Ulysses16



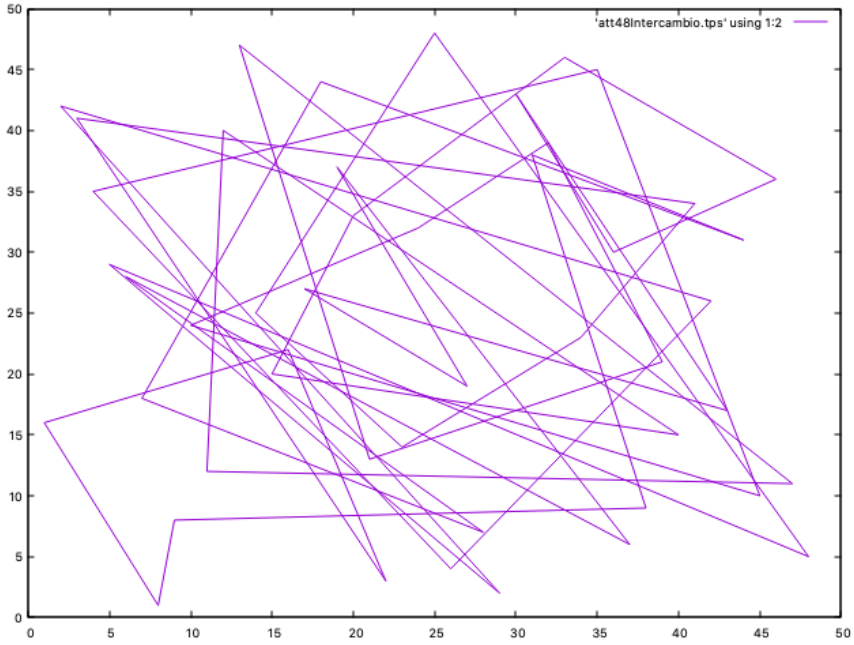

```
ALGORITMO DE INTERCAMBIO:
Coste inicial: 83
Coste final: 83
5 -> 15 -> 6 -> 7 -> 13 -> 12 -> 14 -> 8 -> 4 -> 2 -> 3 -> 1 -> 16 -> 10 -> 9 -> 11 -> 5
El coste del camino es: 83
```

Ulysses22



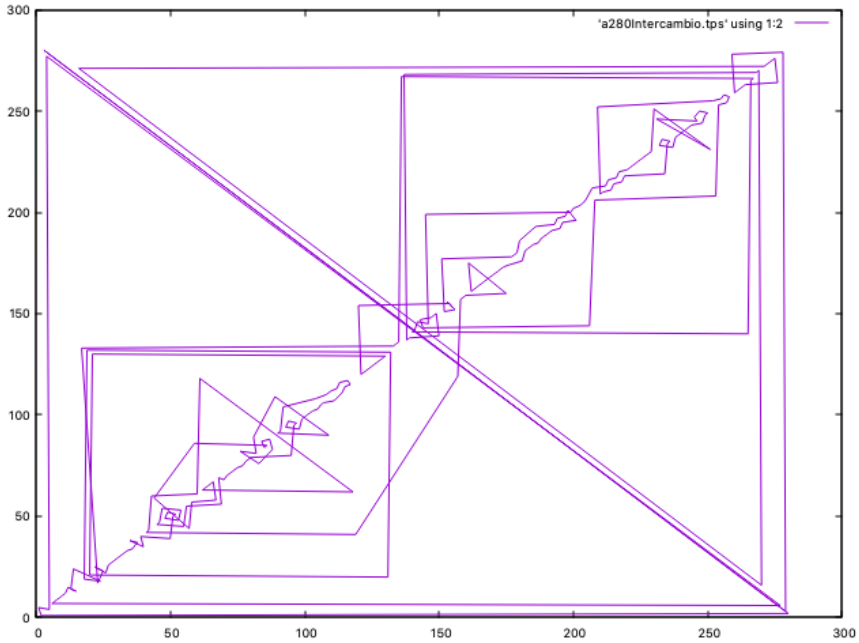

```
ALGORITMO DE INTERCAMBIO:
Coste inicial: 87
Coste final: 79
7 -> 6 -> 5 -> 11 -> 9 -> 10 -> 19 -> 20 -> 21 -> 12 -> 13 -> 16 -> 1 -> 3 -> 2 -> 17 -> 18 -
> 4 -> 22 -> 8 -> 14 -> 15 -> 7
El coste del camino es: 79
```

att48




```
ALGORITMO DE INTERCAMBIO:
Coste inicial: 42019
Coste final: 41807
45 -> 10 -> 24 -> 32 -> 39 -> 21 -> 13 -> 47 -> 11 -> 12 -> 40 -> 15 -> 20 -> 33 -> 46 -> 36 -> 30
-> 43 -> 17 -> 27 -> 19 -> 37 -> 6 -> 28 -> 7 -> 18 -> 44 -> 31 -> 38 -> 9 -> 8 -> 1 -> 16 -> 22
-> 3 -> 41 -> 34 -> 23 -> 14 -> 25 -> 48 -> 5 -> 29 -> 2 -> 42 -> 26 -> 4 -> 35 -> 45
El coste del camino es: 41807
```

a280

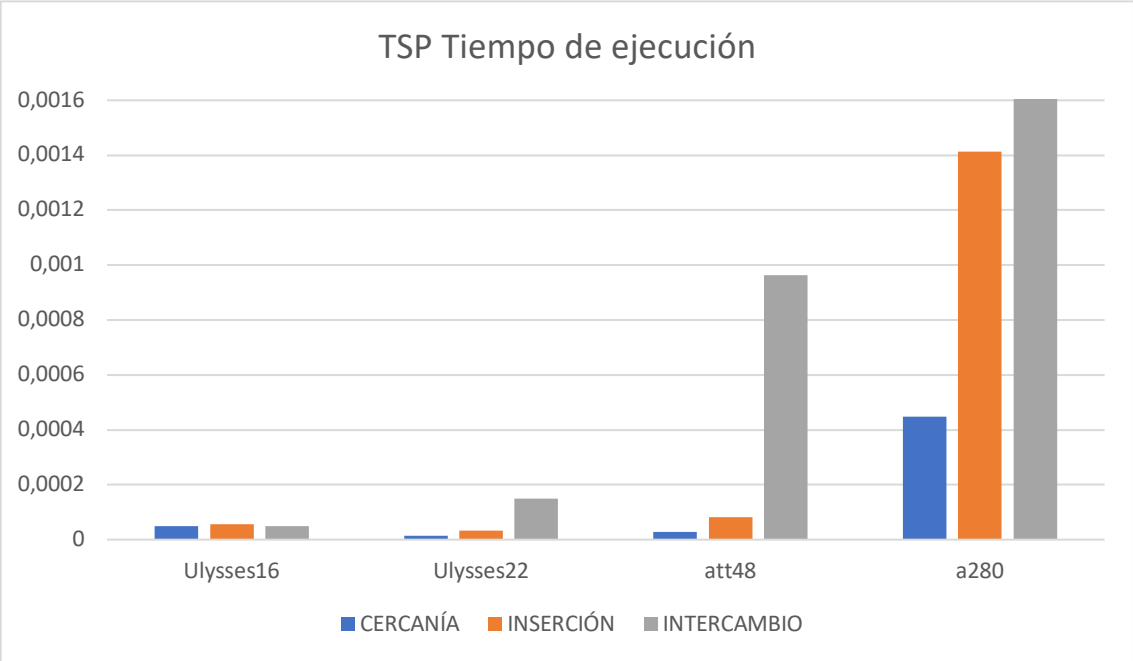


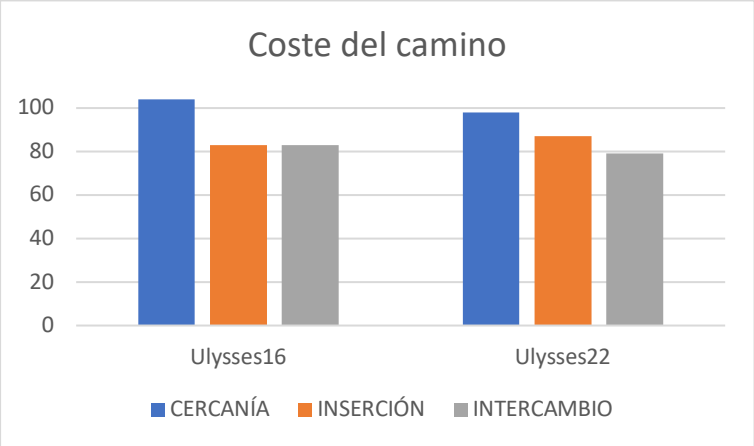

```
ALGORITMO DE INTERCAMBIO:
Coste inicial: 3430
Coste final: 3356
68 -> 69 -> 56 -> 55 -> 45 -> 46 -> 47 -> 54 -> 53 -> 48 -> 49 -> 52 -> 51 -> 50 -> 39 -> 40
-> 35 -> 38 -> 37 -> 36 -> 34 -> 33 -> 32 -> 31 -> 30 -> 29 -> 28 -> 27 -> 26 -> 22 -> 25 ->
23 -> 17 -> 133 -> 134 -> 135 -> 136 -> 267 -> 266 -> 265 -> 140 -> 141 -> 142 -> 146 -> 145
-> 199 -> 200 -> 202 -> 203 -> 204 -> 205 -> 207 -> 212 -> 213 -> 216 -> 217 -> 220 -> 221 ->
222 -> 223 -> 224 -> 225 -> 226 -> 227 -> 228 -> 229 -> 230 -> 251 -> 231 -> 246 -> 245 -> 247
-> 250 -> 249 -> 248 -> 244 -> 243 -> 242 -> 241 -> 240 -> 239 -> 238 -> 237 -> 232 -> 233 ->
236 -> 235 -> 234 -> 219 -> 218 -> 215 -> 214 -> 211 -> 210 -> 209 -> 252 -> 255 -> 256 -> 258
-> 257 -> 254 -> 253 -> 208 -> 206 -> 144 -> 143 -> 147 -> 148 -> 149 -> 150 -> 139 -> 138 ->
137 -> 268 -> 269 -> 270 -> 16 -> 271 -> 272 -> 273 -> 274 -> 275 -> 276 -> 264 -> 263 -> 262
-> 261 -> 260 -> 259 -> 278 -> 279 -> 3 -> 280 -> 2 -> 1 -> 5 -> 4 -> 277 -> 6 -> 7 -> 8 -> 9
-> 10 -> 11 -> 12 -> 15 -> 13 -> 14 -> 24 -> 18 -> 19 -> 132 -> 131 -> 20 -> 21 -> 130 -> 129
-> 128 -> 127 -> 126 -> 125 -> 124 -> 123 -> 122 -> 121 -> 120 -> 154 -> 155 -> 153 -> 156 ->
152 -> 151 -> 177 -> 178 -> 179 -> 180 -> 186 -> 193 -> 194 -> 197 -> 198 -> 201 -> 196 -> 195
-> 192 -> 191 -> 190 -> 189 -> 188 -> 187 -> 185 -> 184 -> 183 -> 182 -> 181 -> 176 -> 174 ->
173 -> 172 -> 171 -> 170 -> 169 -> 168 -> 167 -> 166 -> 165 -> 164 -> 163 -> 162 -> 161 -> 175
-> 160 -> 159 -> 158 -> 157 -> 119 -> 41 -> 42 -> 43 -> 60 -> 61 -> 118 -> 62 -> 63 -> 64 ->
65 -> 66 -> 67 -> 58 -> 57 -> 44 -> 59 -> 86 -> 85 -> 84 -> 87 -> 88 -> 83 -> 76 -> 82 -> 81
-> 89 -> 109 -> 90 -> 91 -> 92 -> 104 -> 108 -> 110 -> 112 -> 113 -> 116 -> 117 -> 115 -> 114
-> 111 -> 107 -> 106 -> 105 -> 103 -> 102 -> 101 -> 100 -> 99 -> 98 -> 93 -> 94 -> 97 -> 96 -
> 95 -> 80 -> 79 -> 78 -> 77 -> 75 -> 74 -> 73 -> 72 -> 71 -> 70 -> 68
El coste del camino es: 3356
```

1.2. Comparativa de tiempos de ejecución

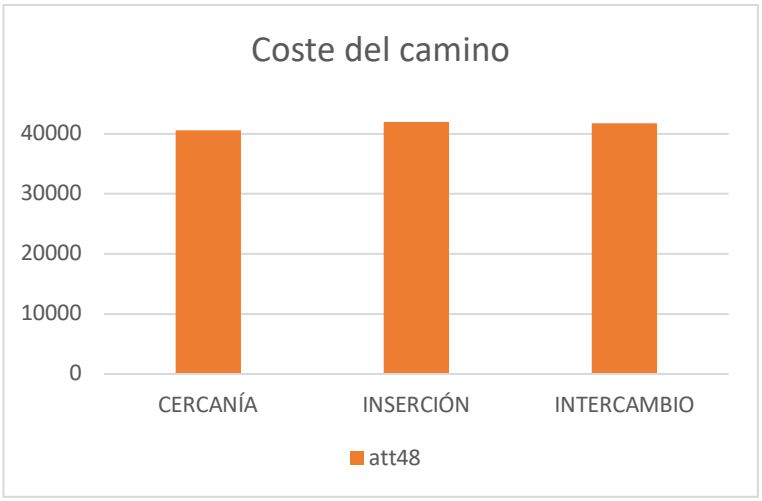
	CERCANÍA	INSERCIÓN	INTERCAMBIO
Ulysses16	0,00005	0,000056	0,00005
Ulysses22	0,000014	0,000033	0.00015
att48	0,000027	0,000081	0.000963
a280	0.000449	0.001413	0.224771

Podemos observar que en los 4 casos la estrategia basada en cercanía es la mejor en términos de tiempo de ejecución. Además, observamos que para casos más pequeños no es notoria la diferencia en tiempo de ejecución. Sin embargo, en casos cada vez más grandes vemos cómo la estrategia cercanía llega a tardar 1/3 de lo que tardaría a través de la estrategia de inserción.

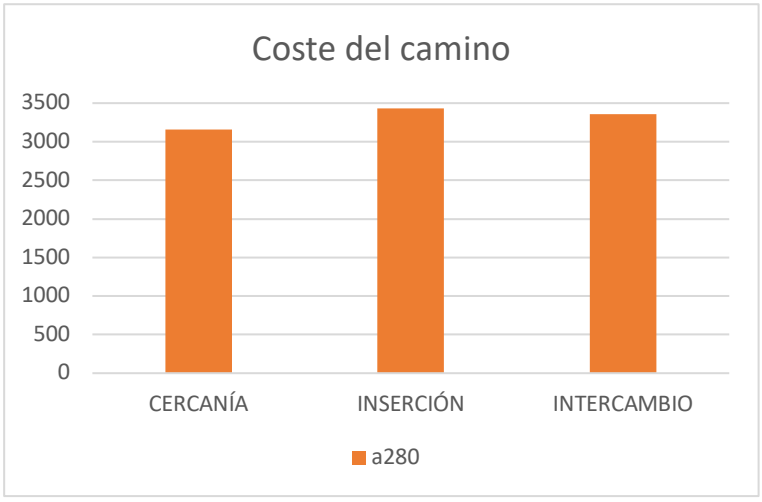




En las muestras Ulysses puedo discernir que a pesar de que la estrategia basada en cercanía sea la que menor tiempo tarda, es la que peor solución nos da, debido a su coste un 20% superior al de inserción.



En la muestra att48 podemos observar que las 3 estrategias son bastante similares, donde la estrategia inserción es levemente peor que las otras dos.



En la muestra a280, igual que en el caso anterior, la inserción es levemente peor en calidad de solución que el resto de las estrategias.

Capítulo 2: Contenedores en un barco

El problema de contenedores en un barco consiste en realizar un algoritmo que, dado un buque con capacidad de carga K (en toneladas) y un número de contenedores c cuyos pesos son p, maximice el número de contenedores cargados y maximice el número de toneladas cargadas.

2.1 Optimizar número de contenedores

```
solucion, contenedor: vector<integer>
obj, candidato, p, tam_max, peso_actual=0, minpos=0: integer

[......inicialización......]

while(!contenedor.empty() && peso_actual <= peso_max)
begin
    min=contenedor[0]
    for i=1..contenedor.size()
    begin
        p=contenedor[i]
        if(min >= p)
        begin
            min=p
            minpos=i
        end
    end

    candidato = minpos
    obj= contenedor[candidato]
    contenedor[candidato]=contenedor[contenedor.size()-1]
    contenedor.pop_back()
    if((peso_actual + obj) <= peso_max)
    begin
        solucion.push_back(obj)
        peso_actual+= obj;
    end
end

return solucion
```

4: Pseudocódigo optimizar número de contenedores

He utilizado dos bucles anidados para calcular la solución al problema. Para ello he usado un bucle que se realizará mientras el contenedor no sea vacío y que el peso, cada vez que cargo el barco con un contenedor, sea menor que el peso máximo que éste puede alcanzar.

Dentro de este bucle nos encontramos con otro, de tipo *for* que recorrerá el contenedor desde la posición 1, ya que la 0 se la hemos asumido a la variable *min*, la que almacenará el mínimo de los pesos de los contenedores. Esto es precisamente lo que se hace dentro de este bucle, obtener la posición y el peso del contenedor de mínimo peso no escogido anteriormente.

Posteriormente, solo queda colocar dentro de nuestro barco, el contenedor con mínimo peso hasta que deje de cumplirse la condición indicada en el primer párrafo de esta explicación.

2.2 Optimizar número de toneladas

```
solucion, contenedor: vector<integer>
obj, candidato, p, tam_max, peso_actual=0, maxpos=0: integer

[......inicialización......]

while(!contenedor.empty() && peso_actual <= tam_max)
begin
    max=contenedor[0]
    for i=1..contenedor.size()
    begin
        p=contenedor[i]
        if(max <= p)
        begin
            max=p
            maxpos=i
        end
    end

    candidato= maxpos
    obj= contenedor[candidato]
    contenedor[candidato]=contenedor[contenedor.size()-1]
    contenedor.pop_back()
    if((peso_actual + obj) <= tam_max)
    begin
        solucion.push_back(obj)
        peso_actual+= obj;
    end
end

return solucion
```

5: Pseudocódigo optimizar número de toneladas

La explicación de este caso es idéntica a la anterior, salvo a que en vez de encontrar en el segundo bucle el contenedor de peso mínimo, buscaremos el de peso máximo, para así conseguir que el barco se llene con los contenedores más pesados.

2.3 Generación del código y escenarios de ejecución

El código generado se puede encontrar adjunto al fichero *zip*. Para su compilación se invita a copiar las dos primeras líneas del código.

```
victorjoserubialopez@Mac-mini-de-Victor codigos % g++ -std=c++11 maximizarelnumerodetoneladas.cpp -o maximizarelnumerodetoneladas
victorjoserubialopez@Mac-mini-de-Victor codigos % ./maximizarelnumerodetoneladas
Formato ./maximizarelnumerodetoneladas <numero contenedores>
victorjoserubialopez@Mac-mini-de-Victor codigos % ./maximizarelnumerodetoneladas 150
Para tamaño: 150 tiempo: 6.2853e-05
El buque lleva 93 paquetes finalmente
```

6: Ejecución Optimizar Número de Toneladas

```
victorjoserubialopez@Mac-mini-de-Victor codigos % g++ -std=c++11 numerodecontenedorescargados.cpp -o numerodecontenedorescargados
victorjoserubialopez@Mac-mini-de-Victor codigos % ./numerodecontenedorescargados
Formato ./numerodecontenedorescargados <numero contenedores>
victorjoserubialopez@Mac-mini-de-Victor codigos % ./numerodecontenedorescargados 150
Para tamaño: 150 tiempo: 6.588e-05
El buque lleva 140 paquetes finalmente
```

6: Ejecución Optimizar Número de Contenedores

Capítulo 3: Conclusiones

El algoritmo de viajante de comercio, para el método basado en el vecino más cercano, retorna, dada una ciudad, la ciudad más cerca de esta. El coste promedio de este algoritmo es un 25% superior al coste mínimo posible, por tanto, no es un algoritmo adecuado si queremos encontrar una solución óptima.

Para el método basado en inserción, al ir reconectando las ciudades conforme las vamos añadiendo, se produce un circuito Hamiltoniano que produce una solución óptima que el método basado en el vecino más cercano.

Para el método de intercambio, se basa en coger dos ciudades que se crucen y reconectarlos en dos cominos resultantes, para que así nos salga un coste menor al inicial.

Por último, para el algoritmo de contenedores en un barco, hemos comprobado que, si queremos optimizar el número de contenedores que deben caber en un barco, debemos coger primero aquellos contenedores que pesen menos, con el fin de que así quepan más. Por otro lado, si queremos optimizar el número de toneladas dentro del barco, debemos coger primero aquellos contenedores que pesen más, para así llenar ocupar el barco con la carga máxima posible.