

Práctica 3:

Algoritmos

Greedy

Equipo 2 - Grupo C1

Joaquín García Venegas

Alejandro Escalona García

Pedro Bedmar López

Joaquín Alejandro España Sánchez

ÍNDICE

Introducción. algoritmos greedy.....	3
Ejercicio 1. Problema del viajante de comercio	3
1.- <i>Enfoque 1: cercanía.</i>	3
2- <i>Enfoque 2: inserción.</i>	5
3- <i>Enfoque 3: Propio</i>	7
4.- <i>Comparación entre los diferentes algoritmos</i>	8
4.1 Tiempos de ejecución.....	8
4.2 Costes totales del recorrido	9
4.3 Comparación mapas	9
6.- <i>¿Cómo compilar y ejecutar automáticamente?</i>	15
7.- <i>¿Cómo ejecutar manualmente?</i>	15
Ejercicio 3.1: Asignación de tareas	16
1.- <i>¿Cómo podemos plantear este algoritmo Greedy?</i>	16
Procedimiento Greedy.....	17
2.- <i>Componentes del planeamiento Greedy.....</i>	21
3.- <i>Explicación del comportamiento del algoritmo</i>	21
4.- <i>Pseudocódigo.....</i>	22
5- <i>¿Cómo compilar los programas?</i>	22
6.- <i>¿Cómo ejecutar los programas?</i>	23
7.- <i>Varias ejecuciones</i>	23
Ejemplo con 6 tareas	24
Ejemplo con 8 Tareas.....	24
Ejemplo con 9 tareas	24
Conclusión.....	25

INTRODUCCIÓN. ALGORITMOS GREEDY

Los algoritmos de tipo voraz (o Greedy) se basan en la estrategia de seleccionar en cada momento la mejor opción de entre un conjunto de candidatos sin tener en cuenta otras decisiones tomadas anteriormente hasta obtener una solución del problema.

Para que un problema se pueda resolver mediante este enfoque, es necesario que se den seis características que no son necesarias, pero sí suficientes:

- Debe existir un conjunto de candidatos.
- Debe haber una lista con los candidatos ya usados.
- Una función solución (que indica cuando se ha alcanzado una solución válida).
- Un criterio que establece cuando un conjunto de candidatos es factible.
- Una función de selección del candidato más prometedor.
- Una función objetivo.

EJERCICIO 1. PROBLEMA DEL VIAJANTE DE COMERCIO

En su formulación más sencilla, el problema del viajante de comercio (TSP, por sus siglas en inglés Traveling Salesman Problem) se define como sigue: dado un conjunto de ciudades y una matriz con las distancias entre todas ellas, un viajante debe recorrer todas las ciudades exactamente una vez, regresando al punto de partida, de forma tal que la distancia recorrida sea mínima. Más formalmente, dado un grafo G, conexo y ponderado, se trata de hallar el ciclo Hamiltoniano de mínimo peso de ese grafo.

Este problema sería muy difícil de abordar mediante el cálculo de todas las permutaciones posibles, ya que su eficiencia es $O(n!)$ y con $n=12$ hablaríamos de 479001600 posibilidades. Esto es inviable en términos de coste computacional, pero la estrategia Greedy nos permite encontrar una solución, normalmente no la más óptima, pero que si se acerca a un máximo local y reduce ampliamente el coste computacional.

1.-ENFOQUE 1: CERCANÍA

Es una forma sencilla de abordar el problema: Para ello, se parte de una ciudad inicial y en cada iteración se elige como ciudad siguiente aquella más cercana a la actual. Esto se ejecuta hasta visitar todas las ciudades. Pero la simpleza de este enfoque tiene un problema, que radica en que las primeras ciudades a elegir si se escogen correctamente (aquellas que menos aumentan el peso circuito) pero es un algoritmo ciego, ya que en las últimas iteraciones quedarán ciudades sueltas que se encontrarán muy distantes entre sí y aumentarán en gran medida el coste del recorrido.

Para elegir la ciudad inicial hay varios enfoques posibles. Uno de ellos se basa en elegir una ciudad aleatoria del conjunto de estas. Otro, el que hemos implementado nosotros, prueba con todas las ciudades iniciales posibles y se queda con aquella que minimiza el coste del recorrido.

PSEUDOCÓDIGO

$C = \text{conjunto de ciudades del mapa}$

Función CERCANÍA

Begin

$\text{costetotalminimo} = \infty$

Repetir desde $i=1$ hasta $C.size()$

$\text{ciudadesvisitadas}[] = C[i]$

$\text{ciudadesporvisitar}[] = C - \text{ciudadesvisitadas}[]$

Mientras $\text{ciudadesporvisitar}[]$ no esté vacío

$\text{distanciaminima} = \infty$

Repetir desde $k=1$ hasta $\text{ciudadesporvisitar}[].size()$

$\text{distancia} = \text{distanciaEuclideaEntreCiudades}(\text{ciudadesporvisitar}[k],$
 $\text{ultimoelemento de } \text{ciudadesvisitadas}[]))$

Si $\text{distancia} < \text{distanciaminima}$ entonces

$\text{distanciaminima} = \text{distancia}$

$\text{ciudadminima} = \text{ciudadesporvisitar}[k]$

$\text{posicionminima} = k$

End Si

End Repetir

Insertar ciudadminima al final de $\text{ciudadesvisitadas}[]$ y eliminarla de $\text{ciudadesporvisitar}[]$

End Mientras

$\text{costetotal} = \text{calcularCosteCamino}(\text{ciudadesvisitadas}[])$

Si $\text{costetotal} < \text{costetotalminimo}$ entonces

$\text{ciudadesvisitadasminimas}[] = \text{ciudadesvisitadas}[]$

$\text{costetotalminimo} = \text{costetotal}$

End Si

End Repetir

End

Devolver ciudadesvisitadas[]

EJEMPLO DE EJECUCIÓN

En este ejemplo de ejecución aplicamos el enfoque de cercanía al mapa ulysses16. Este va a ser el mapa utilizado para todos los ejemplos de ejecución.

Para generar esta captura de ejecución aplicamos lo indicado en los puntos 6 y 7 de este apartado del guión.

```

16 39.36 19.56
2 39.57 26.15
3 40.56 25.32
4 36.26 23.12
1 38.24 20.42
8 37.52 20.44
15 35.49 14.32
5 33.48 10.54
9 41.23 9.1
10 41.17 13.05
11 36.08 -5.21
12 38.47 15.13
14 37.51 15.17
13 38.15 15.35
7 38.42 13.11
6 37.56 12.19
Tiempo de ejecucion: 0.008364
Peso total: 91.51

```

Se imprimen las ciudades (con su identificador, su coordenada x y su coordenada y) en el orden en que van a ser visitadas. También se muestra el tiempo de ejecución del algoritmo y el peso total del grafo que se genera recorriéndolas en el orden mostrado.

2-ENFOQUE 2: INSERCIÓN

En el algoritmo de inserción escogemos 3 ciudades: la ciudad más al norte, es decir, con la coordenada y mayor; la ciudad más al sur, aquella con la coordenada y más pequeña; y la ciudad este, siendo ésta la ciudad con la coordenada x más grande. Una vez seleccionadas estas tres ciudades, las insertamos en un vector de ciudades ya visitadas y el resto se insertan en un vector de ciudades por visitar. Después se comprueba cual es la ciudad de ciudades por visitar que, insertándola en cualquier posible posición de nuestro vector de ciudades visitadas, hace que el coste total del recorrido sea mínimo. Y entonces insertamos esa ciudad en la posición que hace mínimo el anterior coste mencionado, provocando que la borremos del vector de ciudades por visitar. Repetimos este proceso hasta que el vector de ciudades por visitar queda vacío: esto supone que se han visitado todas las ciudades y, por lo tanto, el vector de ciudades visitadas está completo.

PSEUDOCÓDIGO

$C = \text{conjunto de ciudades del mapa}$

Función INSERCIÓN

Begin

 ciudadesvisitadas[] = cnorte + csur + ceste

 ciudadesporvisitar[] = $C - \text{cnorte} - \text{csur} - \text{ceste}$

Mientras ciudadesporvisitar[] no esté vacío

 costeminimo = ∞

Repetir desde $k=1$ hasta ciudadesporvisitar[].size()

Repetir desde $i=1$ hasta ciudadesvisitadas [].size() + 1

 aux[] = ciudadesvisitadas[]

 Insertar ciudadesporvisitar[k] en la posición i de aux

 coste = calcularCosteCamino(aux[]) {Calcula el coste de recorrer ciudades en este orden}

Si coste < costeminimo entonces

 coste = costeminimo

 ciudadminima = ciudadesporvisitar[k]

 posicionminimavisitadas = i

End Si

End Repetir

End Repetir

 Insertar ciudadminima en ciudadesvisitadas[i] y eliminarla de ciudadesporvisitar[]

End Mientras

End

Devolver ciudadesvisitadas[]

EJEMPLO DE EJECUCIÓN

A este ejemplo se aplica lo comentado en el anterior ejemplo de ejecución, con la diferencia de que ahora aplicamos el enfoque de inserción.

```
12 38.47 15.13
13 38.15 15.35
14 37.51 15.17
16 39.36 19.56
1 38.24 20.42
8 37.52 20.44
4 36.26 23.12
2 39.57 26.15
3 40.56 25.32
10 41.17 13.05
9 41.23 9.1
11 36.08 -5.21
5 33.48 10.54
15 35.49 14.32
6 37.56 12.19
7 38.42 13.11
Tiempo de ejecucion: 0.000984
Peso total: 74.6288
```

3-ENFOQUE 3: PROPIO

En nuestra versión, partimos de un vector de ciudades que rellenamos a partir de los datos dados. Después realizamos a través de dos bucles for una serie de intercambios entre dos ciudades del vector indicadas. El cambio se realiza invirtiendo el orden de las ciudades en ese intervalo y finalmente calculamos el coste total del recorrido. Si una vez realizado este cambio el coste total es menor que el coste antes del cambio, guardamos ese orden de ciudades en una variable y el coste total que es menor en otra variable. Esta serie de cambios la realizamos tantas veces como se pueda, según el tamaño del vector, realizando intercambios entre todos con todos. El algoritmo se seguirá ejecutando mientras se sigan produciendo mejoras en los costes, estando esto gestionado por un valor umbral donde si hay n iteraciones seguidas sin cambios se para el algoritmo. En nuestro caso, hemos fijado el valor de n en 100, ya que es un valor equilibrado entre tiempo de ejecución y resultados. Como ejemplo del intercambio que aplicamos, si tenemos un vector 1 2 3 4 5 6 7 con a=2 y b=5 como ciudades que marcan el intervalo de intercambio, el vector resultante sería 1 5 4 3 2 6 7.

Este enfoque se ve influenciado por el algoritmo de búsqueda local 2-opt. Actúa como un algoritmo de descenso por gradiente donde en cada iteración se escala por una montaña y se para cuando se alcanza la cima. Esta cima representa un máximo local, es decir, una posible solución pero que no tiene por qué ser la más óptima. Por lo tanto, no es un algoritmo óptimal ya que no siempre alcanza el mejor resultado.

PSEUDOCÓDIGO

Umbral = {Número de iteraciones máxima sin que se mejore el coste}

Función PROPIO

```
Begin
    contador = 0
    Mientras contador < umbral
        costeminimo = calcularCosteCamino(ciudades[])
        Repetir desde i=1 hasta ciudades[].size()
            Repetir desde j=i+1 hasta ciudades[].size()
                Swap_vector(ciudadesprovisional[],i,j)
                costetotal = calcularCosteCamino(ciudadesprovisional[])
                Si costetotal < costeminimo entonces
                    ciudades[] = ciudadesprovisional[]
                    costeminimo = costetotal
                    contador = 0
                End Si
            End Repetir
        End Repetir
        contador = contador + 1
    End Mientras
End

Devolver ciudades[]
```

EJEMPLO DE EJECUCIÓN

A este ejemplo se aplica lo comentado en el primer ejemplo de ejecución, con la diferencia de que ahora aplicamos el enfoque propio.

```
16 39.36 19.56
2 39.57 26.15
3 40.56 25.32
4 36.26 23.12
1 38.24 20.42
8 37.52 20.44
15 35.49 14.32
5 33.48 10.54
9 41.23 9.1
10 41.17 13.05
11 36.08 -5.21
12 38.47 15.13
14 37.51 15.17
13 38.15 15.35
7 38.42 13.11
6 37.56 12.19
Tiempo de ejecucion: 0.008364
Peso total: 91.51
```

4.-COMPARACIÓN ENTRE LOS DIFERENTES ALGORITMOS

4.1 TIEMPOS DE EJECUCIÓN

Para calcular los tiempos de ejecución, hemos ejecutado 5 veces cada algoritmo y hemos calculado la media de esas ejecuciones. De esta forma obtenemos unos tiempos más precisos y se corrigen posibles valores atípicos debidos al despachador del SO o otros factores externos al programa.

Fichero Tsp/ Tiempo ejecución	Algoritmo Cercanía	Algoritmo Propio	Algoritmo Inserción
a280.tsp	0.552175	18.8016	8.80934
ulysses16.tsp	0.000308	0.007568	0.000756
ulysses22.tsp	0.000747	0.02219	0.002394
att48.tsp	0.008712	0.166956	0.017199

4.2 COSTES TOTALES DEL RECORRIDO

Para el cálculo de los costes no es necesario calcular la media, ya que su valor es siempre el mismo.

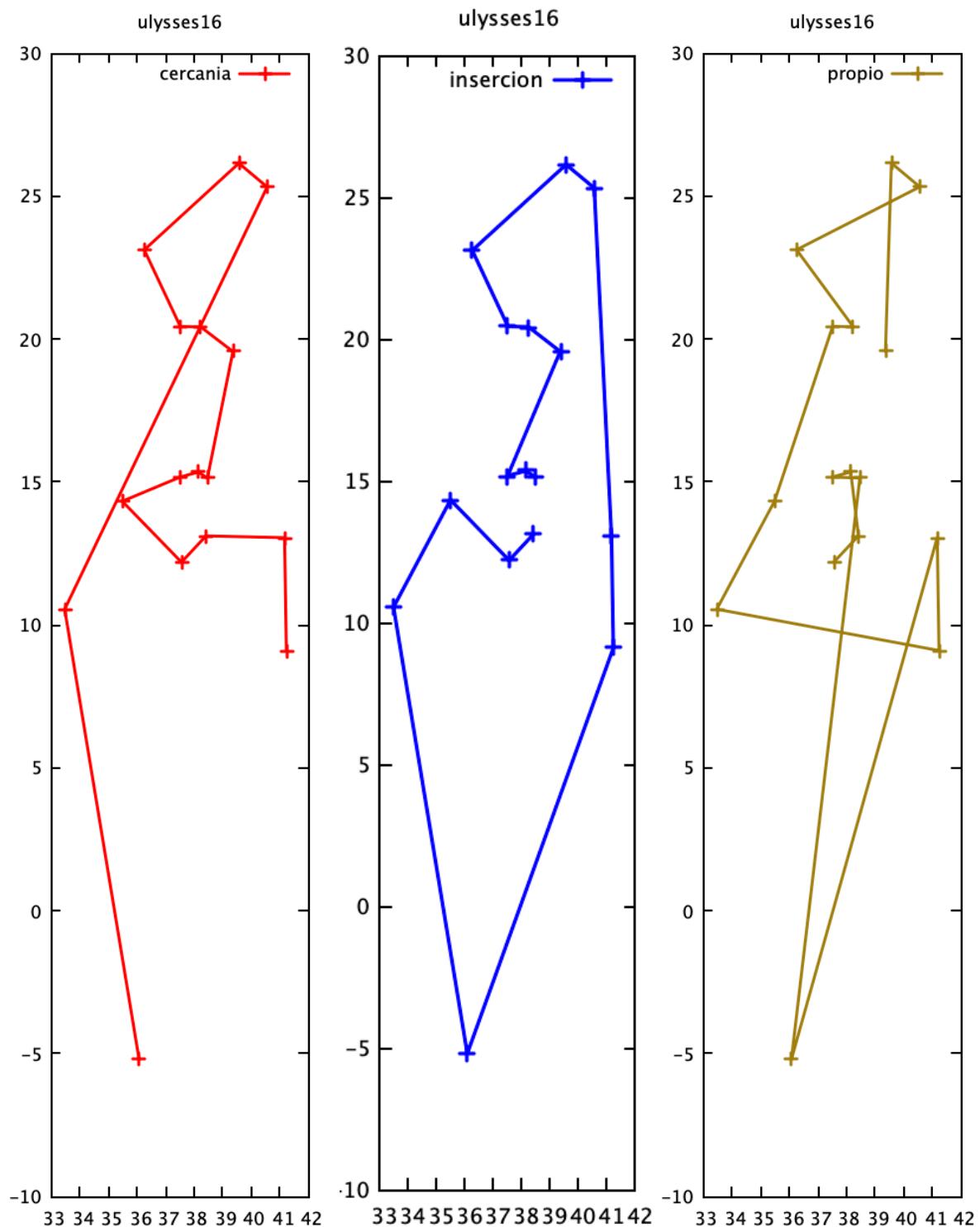
Fichero Tsp/ Coste total del recorrido	Algoritmo Cercanía	Algoritmo Propio	Algoritmo Inserción
a280.tsp	3094.28	2815.32	3051.35
ulysses16.tsp	77.1269	91.51	74.6288
ulysses22.tsp	86.9058	112.82	75.9683
att48.tsp	39236.9	117011	35140.1

4.3 COMPARACIÓN MAPAS

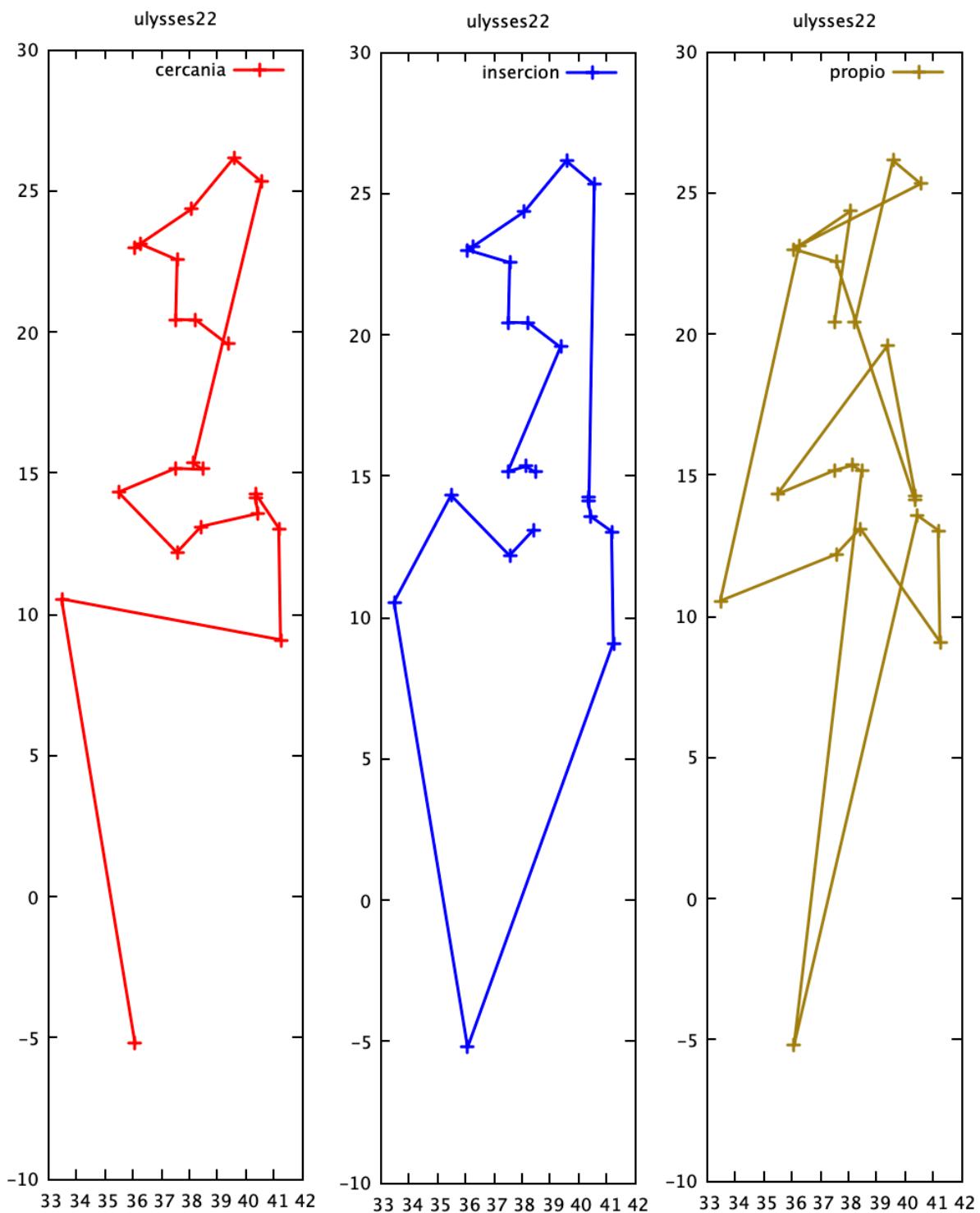
Para la comparación de los algoritmos, los aplicamos a 4 mapas diferentes. Aquí mostramos el mapa graficado para los resultados de estos 3 algoritmos. Para graficar utilizamos gnuplot y aplicamos la instrucción *set size ratio -1* para ajustar los ejes de forma que tengan la misma escala y representen el mapa de forma fiel a la realidad.

También comentar que en el gráfico no dibujamos el arco que une el vértice final con el inicial, pero si lo tenemos en cuenta a la hora de calcular el coste.

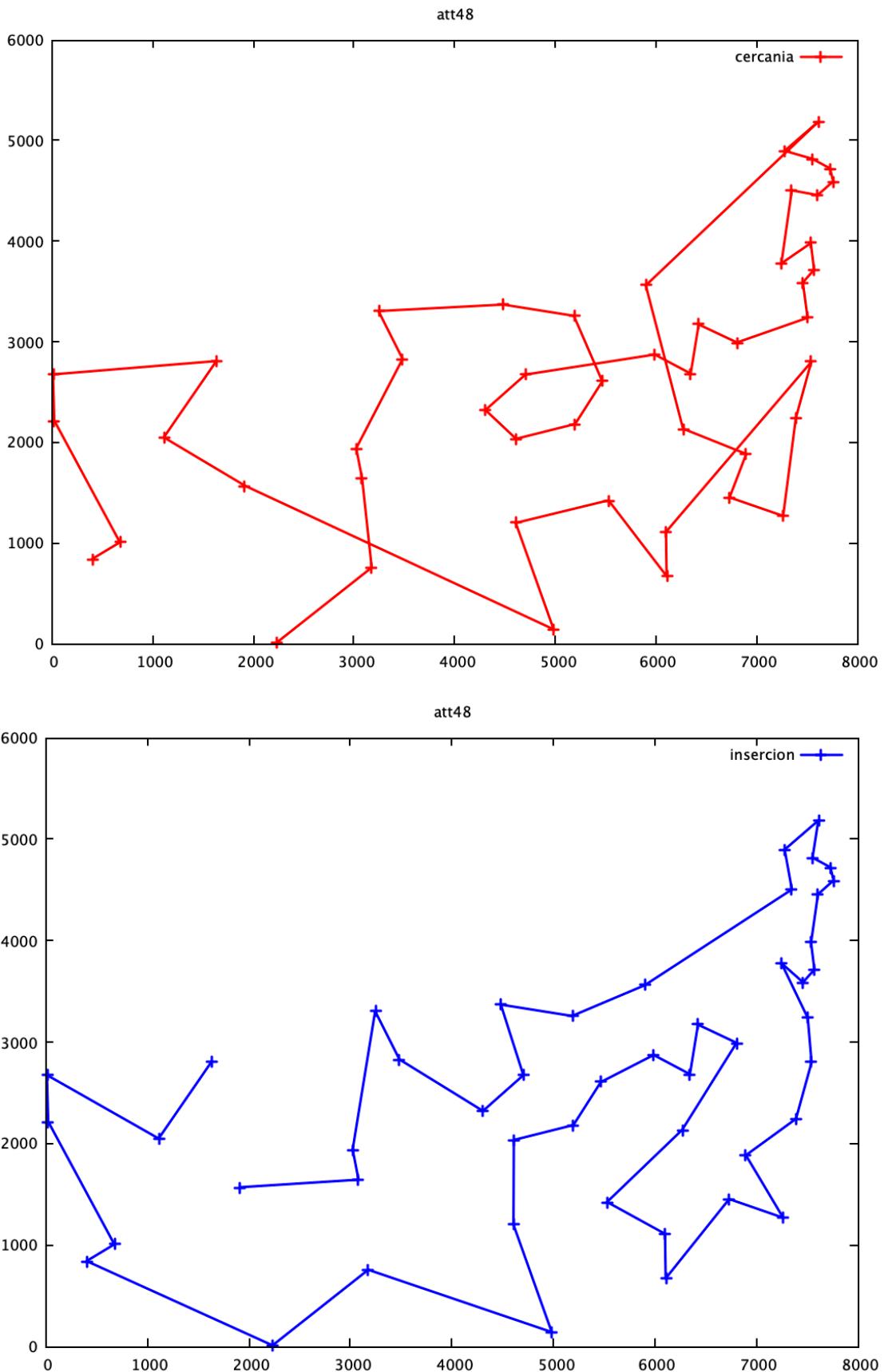
4.3.1 ULYSSES16.TSP:

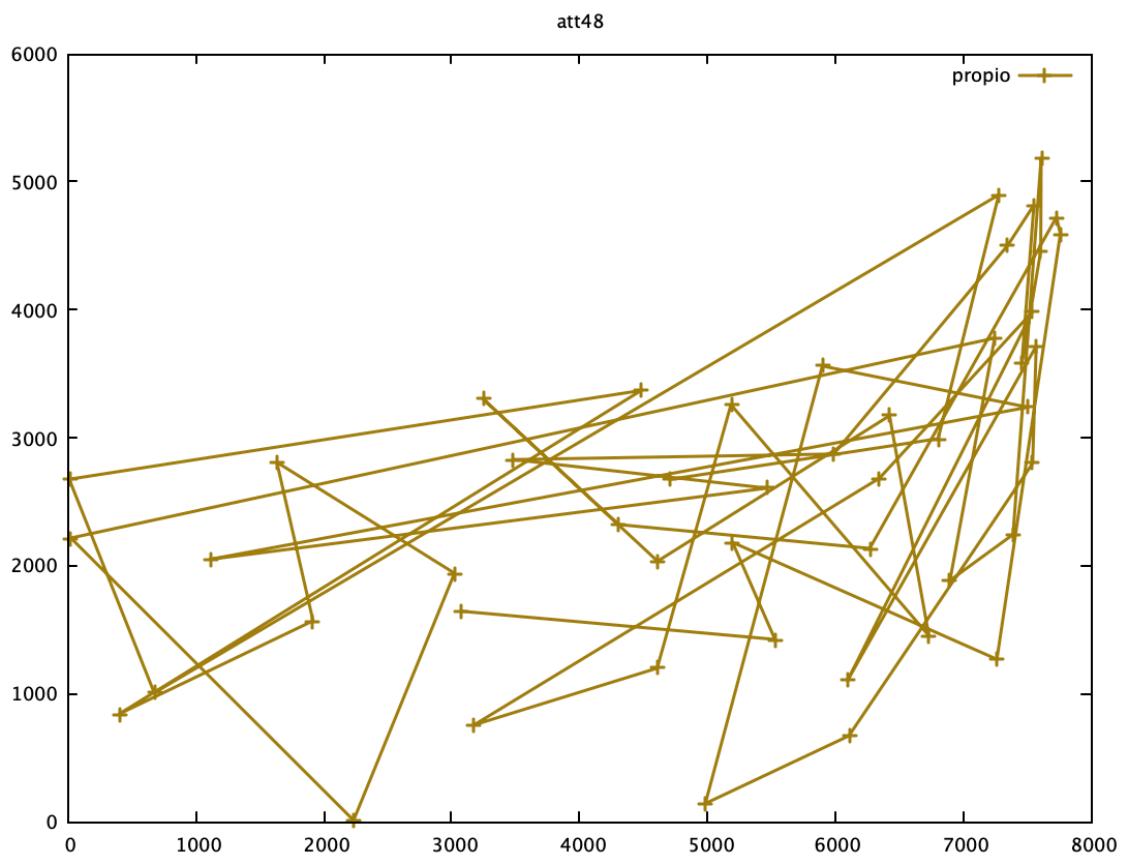


4.3.2 ULYSSES22.TSP:

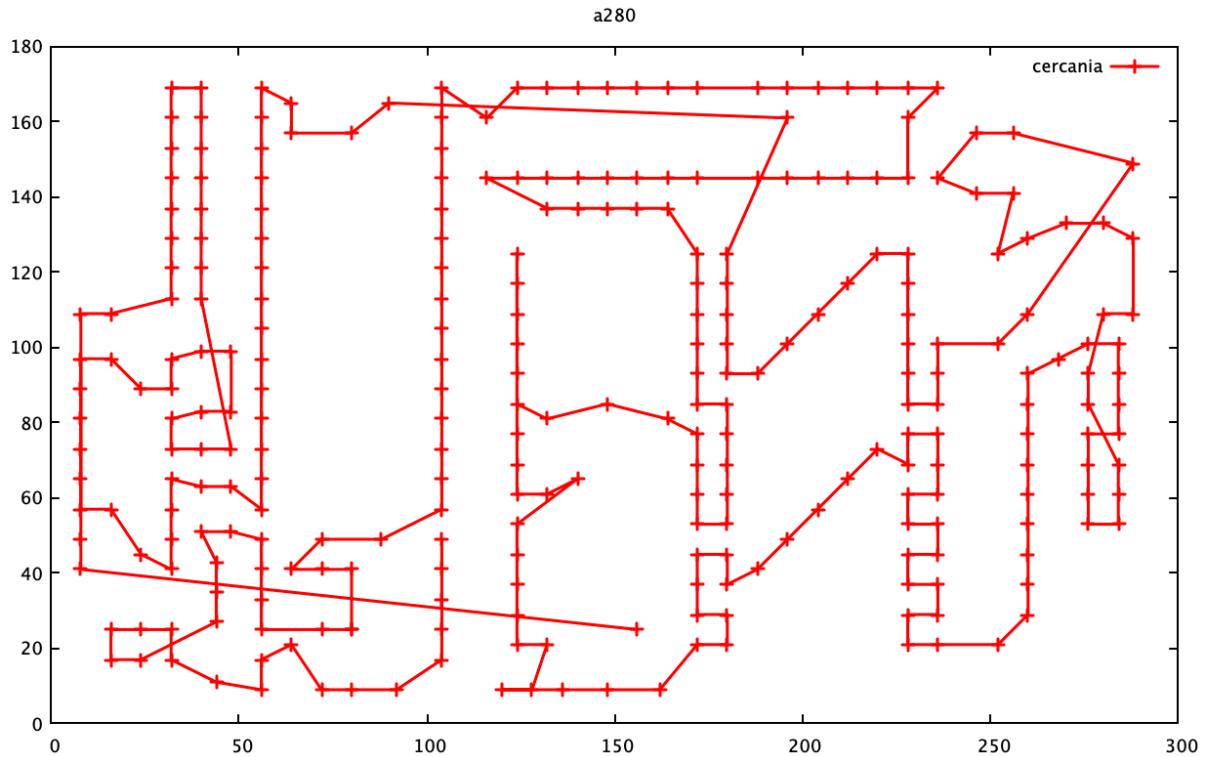


4.3.3 ATT48.TSP:

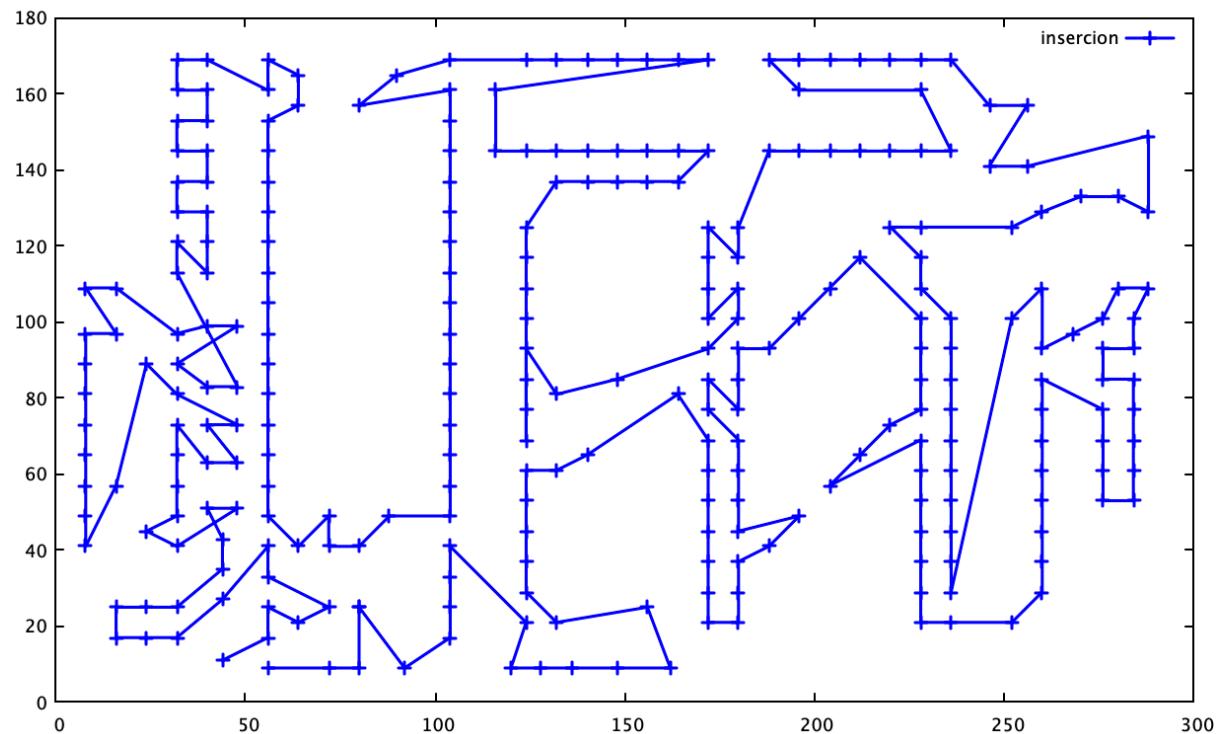




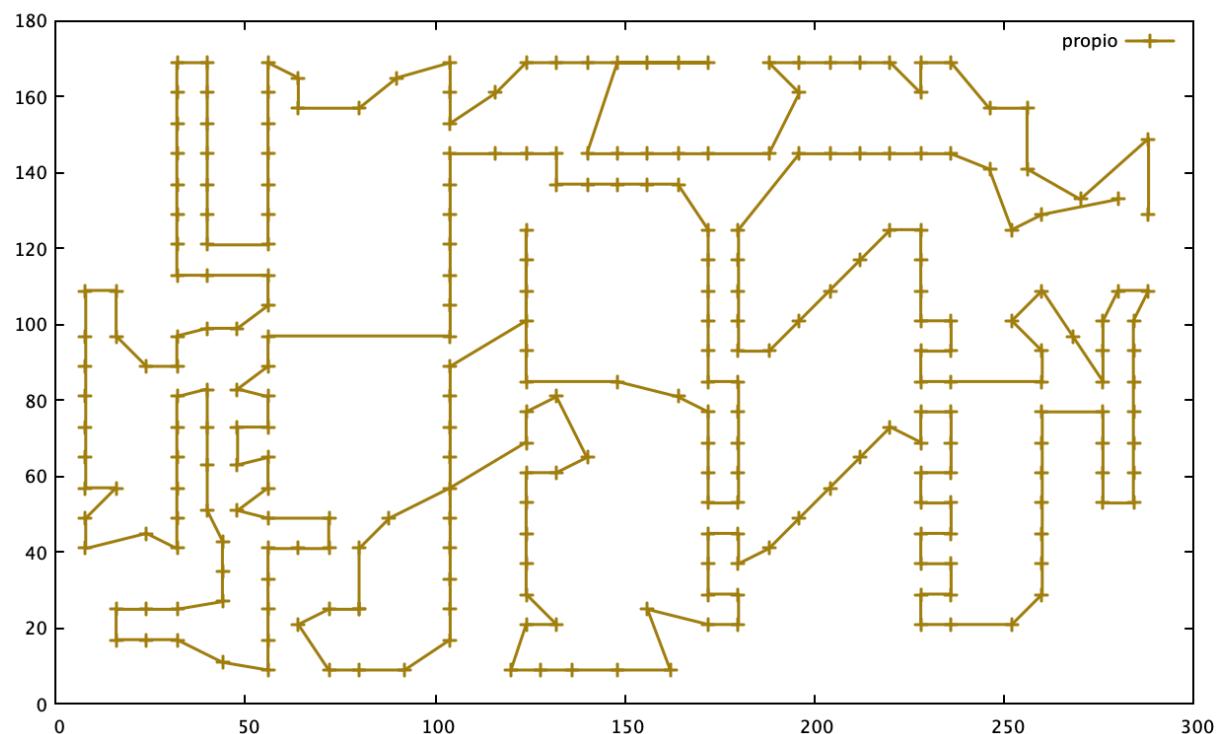
4.3.4 A280.TSP:



a280



a280



6.-¿CÓMO COMPILAR Y EJECUTAR AUTOMÁTICAMENTE?

Para llevar a cabo la compilación de los programas realizaremos las siguientes órdenes en el terminal:

- Para compilar: Hemos incluido un makefile para compilar todos los ficheros de esta práctica. Aplicamos la orden *make* en el terminal.
- Para compilar y ejecutar: *./ejecucion.sh* ejecuta el makefile y además realiza la ejecución de los scripts. Para ello, genera los datos de cada ejecución y con ellos se grafican los mapas en gnuplot. Estos resultados quedan almacenados como .txt (carpeta datos) y los gráficos en .png (carpeta mapas).
- Para eliminar los ficheros ejecutables: *make clean*

El script *ejecucion.sh* sólo lo hemos probado en Linux y en macOS. Para linux funciona sin problemas, y para macOS antes de ejecutarlo es necesario ejecutar *xattr -d com.apple.quarantine ejecucion.sh* en la terminal. Si no lo hacemos, obtendremos un error del tipo */bin/bash: bad interpreter: Operation not permitted*. Creemos que se debe a algún problema de metadatos que es resuelto por la ejecución de la anterior orden. Aún así, por si persistiera el problema, hemos incluido en la entrega la carpeta *ejemplo_ejecucion_tsp* con los resultados que genera este script.

7.-¿CÓMO EJECUTAR MANUALMENTE?

Los tres algoritmos se ejecutan de forma muy similar

./<nombre del ejecutable> <fichero con las ciudades>

Ejemplo:

./propio /tsp_escenarios_ejecucion/a280.tsp

./insercion tsp_escenarios_ejecucion/att48.tsp

./cercania tsp_escenarios_ejecucion/ulysses16.tsp

Para la graficación, utilizamos la herramienta gnuplot:

- *set size ratio -1*
- *plot ‘salida.txt’ using 2:3 w lp*

Con esas dos sentencias seremos capaces de, primero, mantener la misma escala en ambos ejes y, segundo, de graficar a partir de un fichero de datos que contenga una ciudad por línea en el siguiente formato:

nº_nodo coordenada_x coordenada_y

EJERCICIO 3.1: ASIGNACIÓN DE TAREAS

Tenemos que completar un conjunto de n tareas con plazos límite. Cada una de las tareas consume la misma cantidad de tiempo (una unidad) y, en un instante determinado, podemos realizar únicamente una tarea. La tarea i tiene como plazo límite d_i y produce un beneficio g_i (con $g_i > 0$) únicamente si la tarea se realiza en un instante de tiempo $t \leq d_i$. Diseñe un algoritmo voraz que nos permita seleccionar el conjunto de tareas que nos asegure el mayor beneficio posible.

1.- ¿CÓMO PODEMOS PLANTEAR ESTE ALGORITMO GREEDY?

Para empezar, aclararemos cuando un conjunto de tareas es factible, es decir, un conjunto que permite que todas sus tareas se ejecuten antes del final de sus respectivos plazos.

Un algoritmo Greedy para este problema podría consistir en construir el conjunto paso a paso, añadiendo en cada paso la tarea que tenga el mayor valor de beneficio (g_i), entre las que aún no se han considerado, siempre que el conjunto siga siendo factible.

Para explicarlo con más detalle, procederemos a poner un ejemplo:

i	1	2	3	4
Gi	50	10	15	30
Di	2	1	2	1

Primero obtenemos el número máximo de tareas que puede contener nuestro conjunto, que será el mínimo entre el número de tareas y el instante máximo de estas tareas, en este caso es 2 ($\min(4,2) = 2$), entonces como mucho podremos tener 2 tareas. Además, deberemos descartar aquellos conjuntos que no son factibles, es decir, aquellos formados por dos tareas y la segunda tarea sea la 2 o la 4, ya que su instante máximo es 1, por ejemplo $\{1,2\}$, $\{1,4\}$, $\{3,2\}$, $\{3,4\}$.

A continuación, procedemos a poner aquellos conjuntos factibles junto con su beneficio para ver cuál sería el óptimo:

Conjunto	Beneficio
{1}	50
{2}	10
{3}	15
{4}	30
{1,3}	65
{3,1}	65
{2,1}	60
{2,3}	25
{4,1}	80
{4,3}	45

Como se observa, el conjunto que produce mayor beneficio es aquel en el que se ejecuta primero la tarea 4 y luego la tarea 1.

PROCEDIMIENTO GREEDY

1. Calculamos el número de tareas máximo: $\text{máx_tareas} = \min(4, \max(d_i)) = \min(4, 2) = 2$.
2. Ordenamos las tareas de forma decreciente de beneficio.
3. Realizamos tantos intentos como máx_tareas tengamos para asignar la tarea a la posición del conjunto resultante correspondiente con su d_i .

- **Tareas ordenadas:**

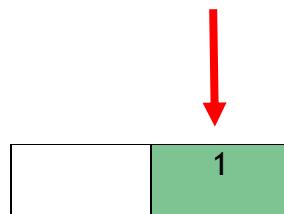
i	1	2	3	4
G_i	50	10	15	30
D_i	2	1	2	1

- **Conjunto resultante:**

--	--

- 1er intento: Cogemos la primera tarea y la colocamos en la posición indicada por su di:

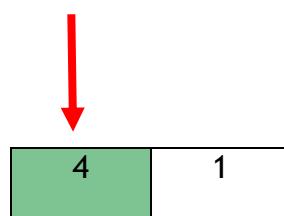
i	1	4	3	2
Gi	50	30	15	10
Di	2	1	2	1



Nota: Si hubiera estado ocupada, la hubiéramos colocado en un hueco libre anterior a esa posición.

- 2º intento: Cogemos la segunda tarea y la colocamos en la posición indicada por su di:

i	1	4	3	2
Gi	50	30	15	10
Di	2	1	2	1



Como hemos alcanzado ya el máximo de tareas, nuestro algoritmo termina, la secuencia sería 4,1 con un valor de **beneficio de 80**.

En este caso, el ejemplo era muy sencillo, y con sólo dos intentos ya obtenemos el conjunto óptimo para este conjunto de tareas. En casos más complejos, hay que controlar qué posiciones están libres y cuales ya han sido ocupadas para ir recorriendo hacia atrás el conjunto e introducir la tarea en un hueco libre si es que lo hubiera, si no hay ningún hueco en el que se pueda introducir esa tarea, ésta sería descartada, no pertenecería a un conjunto factible.

Vamos a demostrar otro ejemplo con 6 tareas:

I	1	2	3	4	5	6
Gi	10	30	20	30	50	20
Di	4	3	1	1	2	2

$\min(6, \max(d_i)) = \min(6, 4) = 4$
Este es el máximo nº de tareas que podrá tener la secuencia

Ordenamos de forma decreciente de beneficio

I	5	4	2	3	6	1
Gi	50	30	30	20	20	10
Di	2	1	3	1	2	4

- Vamos a proceder a ir introduciéndolas en el conjunto resultante:

I	5	4	2	3	6	1
Gi	50	30	30	20	20	10
Di	2	1	3	1	2	4

	5					
--	---	--	--	--	--	--

Conjunto resultante

Primer intento

I	5	4	2	3	6	1
Gi	50	30	30	20	20	10
Di	2	1	3	1	2	4

4	5					
---	---	--	--	--	--	--

Conjunto resultante

Segundo intento

I	5	4	2	3	6	1
Gi	50	30	30	20	20	10
Di	2	1	3	1	2	4

4	5	2	
---	---	---	--

Conjunto resultante

Tercer intento

I	5	4	2	3	6	1
Gi	50	30	30	20	20	10
Di	2	1	3	1	2	4

4	5	2	
---	---	---	--

Conjunto resultante

Cuarto intento

Como no hay posición libre anterior, se descarta la tarea

I	5	4	2	3	6	1
Gi	50	30	30	20	20	10
Di	2	1	3	1	2	4

4	5	2	
---	---	---	--

Conjunto resultante

Quinto intento

Como no hay posición libre anterior, se descarta la tarea

I	5	4	2	3	6	1
Gi	50	30	30	20	20	10
Di	2	1	3	1	2	4

4	5	2	1	
---	---	---	---	--

Conjunto resultante

Sexto intento

4	5	2	1
---	---	---	---

= 120 → Beneficio máximo

Como se puede comprobar, coincide con el resultado de nuestro algoritmo:

```
joaquin@joaquin-X550VX:~/Escritorio/Dropbox/2º-Informatica/Segundo_cuatrimestre
/Practicas/P3/asignacionTareas$ ./asignacionTareas 6_tareas
Las tareas introducidas con las siguientes:
I    1    2    3    4    5    6
Gi   10   30   20   30   50   20
Di   4     3    1    1    2    2

La secuencia de tareas es: 4 5 2 1
Con un beneficio total de: 120
```

2. -COMPONENTES DEL PLANEAMIENTO GREEDY

Para resolver este problema, se han considerado las componentes que forman el algoritmo Greedy, entre estas componentes se encuentran:

- **Candidatos:** conjunto de tareas a ejecutar.
- **Candidatos ya usados:** tareas que ya han sido seleccionadas.
- **Función Selección:** elegir siempre la tarea cuyo beneficio es mayor.
- **Función de factibilidad:** comprobar que la tarea no haya sido seleccionada anteriormente y que el instante máximo de la tarea elegida no sea inferior al instante actual.
- **Función de solución:** cuando se ha completado el vector de la secuencia solución o se han recorrido todas las tareas.
- **Función objetivo:** Suma de los beneficios de las tareas elegidas. Esta será la función a optimizar.

3. -EXPLICACIÓN DEL COMPORTAMIENTO DEL ALGORITMO

Un conjunto S de n tareas será factible si solo si se puede formar una planificación de tareas que contenga a todas las tareas de S . Para ello, se debe de seguir la siguiente metodología. Comenzamos con una planificación de tamaño n y que inicialmente se encuentra vacía. Para cada tarea i perteneciente al conjunto S , se planifica la tarea i en el instante t_i , donde t_i es el instante de finalización de la tarea i tal que $1 \leq t_i \leq \min(n, \max(d_i))$, siempre y cuando no se haya planificado una tarea j con $t_j = t_i$ anteriormente.

Cuando se prueba a añadir una nueva tarea, la secuencia que se está construyendo siempre debe tener al menos un hueco libre. Si suponemos que no se puede añadir una tarea i cuyo plazo de finalización es d_i esto quiere decir que todos los instantes desde $t=1$ hasta $t=\min(d_i, n)$ están ocupados. Suponemos c el menor entero tal que la posición de la secuencia $t=c$ está vacía, entonces ya se habrán planificado $c-1$ tareas, siendo por lo tanto $d_i < c$.

4.-PSEUDOCÓDIGO

```
1. Funcion PLANTAREAS( lista_tareas[0..n], max_tareas)
2. Begin
3.     tareas_introducidas ← 0          {Tareas incluidas en la solución}
4.     secuencia[0..max_tareas]         {Vector que contendrá la solución}
5.     i ← 0
6.     Ordenar (lista_tareas)          {ordenar la lista en orden decreciente de beneficio}
7.
8.         While (tareas_introducidas != max_tareas and i != n-1)
9.             pos_insertar ← instante de fin de la tarea i
10.            If secuencia[pos_insertar] no está libre {Posición ocupada}
11.                pos_anterior ← pos_insertar
12.                If pos_anterior != 0
13.                    pos_anterior ← pos_anterior-1
14.                    While (secuencia[pos_anterior] no esté libre and
15.                        pos_anterior >= 0)
16.                        pos_anterior ← pos_anterior-1
17.                        If pos_anterior >=0
18.                            secuencia[pos_anterior] ← i
19.                            tareas_introducidas ← tareas_introducidas + 1
20.                        End if
21.                End if
22.            Else {Posición libre}
23.                secuencia[pos_anterior] ← i
24.                tareas_introducidas ← tareas_introducidas + 1
25.            End if
26.            i ← i +1      {Avanzamos a la siguiente tarea}
27.        End while
28.
29.        Repeat from j=0 to max_tareas
30.            indice_tarea_insertada ← secuencia[j]
31.            {Cambiemos el indice de la tarea ordenada por el índice inicial}
32.            If indice_tarea_insertada != -1
33.                secuencia[j]←indice de la tarea indice_tarea_insertada de lista_tareas
34.            End if
35.        Return secuencia[0..max_tareas]
36.    End
```

5.-¿CÓMO COMPILAR LOS PROGRAMAS?

Para llevar a cabo la compilación de los programas realizaremos las siguientes órdenes en el terminal:

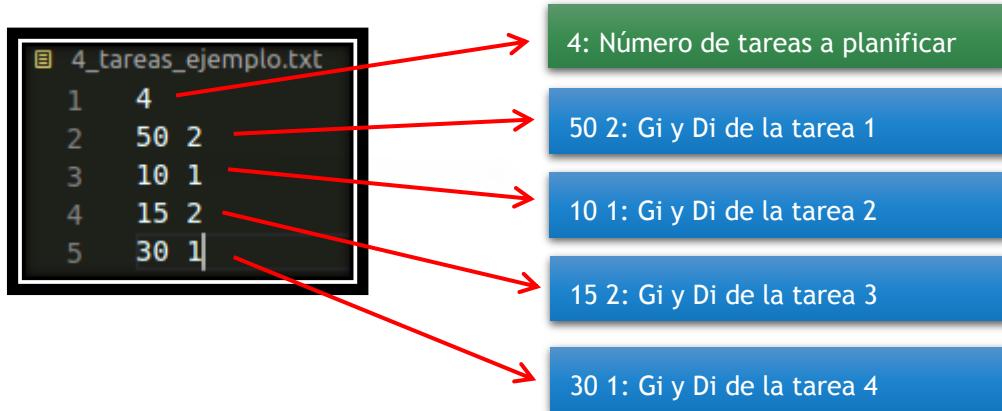
- Para compilar: make
- Para eliminar los ficheros ejecutables: make clean

6.-;CÓMO EJECUTAR LOS PROGRAMAS?

El algoritmo se ejecuta siguiendo el siguiente formato:

./<nombre del ejecutable> <fichero con las tareas>

Formato del fichero:



Ejemplo de ejecución: ./asignacionTareas 4_tareas_ejemplo.txt

7.-VARIAS EJECUCIONES

Empezaremos comprobando el ejemplo visto al comienzo de la explicación de este algoritmo:

Datos

```
4_tareas_ejemplo.txt
4
50 2
10 1
15 2
30 1
```

Resultados ejecución

```
joaquin@joaquin-X550VX:~/Escritorio/Dropbox/2º-Informatica/Segundo_cuatrimestre/ALG/Practicas/P3/asignacionTareas$ ./asignacionTareas 4_tareas_ejemplo.txt
Las tareas introducidas con las siguientes:
I      1      2      3      4
Gi    50     10     15     30
Di     2      1      2      1

La secuencia de tareas es: 4 1
Con un beneficio total de: 80
```

EJEMPLO CON 6 TAREAS

Datos

```
6_tareas
1   6
2   10 4
3   30 3
4   20 1
5   30 1
6   50 2
7   20 2
8   50 5
9   20 5
```

Resultado ejecución

```
joaquin@joaquin-X550VX:~/Escritorio/Dropbox/2º-Informatica/Segundo_cuatrimestre/Prácticas/P3/asignacionTareas$ ./asignacionTareas 6_tareas
Las tareas introducidas con las siguientes:
I      1      2      3      4      5      6
Gi    10     30     20     30     50     20
Di      4      3      1      1      2      2

La secuencia de tareas es: 4 5 2 1
Con un beneficio total de: 120
```

EJEMPLO CON 8 TAREAS

Datos

```
8_tareas
1   8
2   20 4
3   10 5
4   7 1
5   15 1
6   25 3
7   15 3
8   5 1
9   30 2
10  30 5
11  2 1
12  -
```

Resultado ejecución

```
joaquin@joaquin-X550VX:~/Escritorio/Dropbox/2º-Informatica/Segundo_cuatrimestre/ALG/Prácticas/P3/asignacionTareas$ ./asignacionTareas 8_tareas
Las tareas introducidas con las siguientes:
I      1      2      3      4      5      6      7      8
Gi    20     10      7     15     25     15      5     30
Di      4      5      1      1      3      3      1      2

La secuencia de tareas es: 4 8 5 1 2
Con un beneficio total de: 100
```

EJEMPLO CON 9 TAREAS

Datos

```
9_tareas
1   9
2   30 5
3   10 3
4   2 2
5   11 2
6   10 1
7   9 2
8   2 7
9   56 5
10  33 4
11  33 4
12  20 2
13  5 1
```

Resultado ejecución:

```
joaquin@joaquin-X550VX:~/Escritorio/Dropbox/2º-Informatica/Segundo_cuatrimestre/ALG/Prácticas/P3/asignacionTareas$ ./asignacionTareas 9_tareas
Las tareas introducidas con las siguientes:
I      1      2      3      4      5      6      7      8      9
Gi    30     10      2     11     10      9      2     56     33
Di      5      3      2      2      1      2      7      5      4

La secuencia de tareas es: 5 4 1 9 8 7
Con un beneficio total de: 351
```

CONCLUSIÓN

- **Respecto al ejercicio 1:**

Para acabar, queremos contrastar los resultados de los tres algoritmos. En general, el que mejores resultados ofrece en cuanto a coste es el algoritmo de inserción, pero este algoritmo sacrifica más tiempo de ejecución que los demás.

En cambio, el algoritmo de cercanía ofrece unos resultados algo peores que el de inserción, pero mejora en buena medida los tiempos de ejecución. Por lo tanto, en cuanto a la relación entre coste y tiempo de ejecución es el mejor.

Por último, el algoritmo propio basado en 2-opt es muy dependiente de los datos de inicio. Si los datos ya están medianamente ordenados de una forma que reduce el coste, al aplicar este algoritmo se obtendrán buenos resultados (como ocurre en a280). Por lo tanto, una buena solución para aplicar esta técnica es primero utilizar otra heurística Greedy que ordene inicialmente las ciudades y después aplicar 2-opt para que mejore el coste.

- **Respecto al ejercicio 3.1:**

A diferencia de lo que podríamos pensar, este algoritmo es mucho más eficiente de lo que puede parecer, ya que si tuviese que hacer las $n!$ combinaciones de tareas y comprobar cuál de ellas es un conjunto factible maximizando el beneficio, esto conllevaría un tiempo de ejecución de $O(n!)$, lo cual es muy ineficiente. Sin embargo, en nuestro algoritmo, ordenar las tareas tiene un coste de $O(n * \log(n))$, pero como en el peor de los casos tendríamos que recorrer las n tareas, este algoritmo tiene una eficiencia de $O(n^2)$.