Práctica 3 - Algoritmos voraces

Memoria de práctica

Germán Castilla López
Jorge Gangoso Klöck
Pedro Morales Leyva
Clara Mª Romero Lara

Índice de Contenidos

1. Descripción general de la práctica

Esta segunda práctica se centra en la técnica de diseño de algoritmos voraces, o *greedy*. Los problemas a considerar son los siguientes:

Un problema común, **el problema del viajante de comercio**, en el que dado un conjunto de ciudades y una matriz con las distancias entre todas ellas, el viajante recorra todas las ciudades exactamente una vez y regrese al punto de partida, de forma tal que la distancia recorrida sea la mínima. Estudiaremos tres enfoques distintos del problema:

- Por cercanía
- Por recorrido parcial
- Por nuestro propio método: *recorrido de cercanía con rentabilidades*

Un problema elegido aleatoriamente, el **3.3: Trabajadores y tareas.** Suponiendo que disponemos de *n* trabajadores y *n* tareas, y siendo $c_{i,j} > 0$ el coste de asignarle la tarea *j* al trabajador *i*; diseñar un algoritmo voraz para la asignación de cada tarea a un trabajador de forma óptima

2. Problema del viajante de comercio

2.1. Por cercanía

2.1.1. Descripción

- **Conjunto de candidatos:** Ciudades sin visitar, elementos del `vector <ciudad>` cuyo equivalente en el vector `visitado[]` tiene valor “false”.
- **Conjunto de candidatos usados:** Ciudades, visitadas, elementos del `vector<ciudad>` cuyo equivalente en el vector `visitado[]` tiene valor “true”.
- **Criterio de solución:** El vector `visitado[]` tiene “true” en todas sus componentes y la última ciudad visitada es por la que se comenzó.
- **Criterio de factibilidad:** En todo momento se podrá llegar a una solución total, a partir de una solución parcial, sólo es necesario continuar seleccionando.
- **Criterio de selección:** Se escoge la ciudad más cercana a la última ciudad visitada y se incluye en la lista de ciudades solución.
- **Objetivo:** Visitar todas las ciudades y volver al principio.

Este método consiste en, teniendo una ciudad inicial, ir seleccionando aquella que se encuentra más próxima como siguiente destino; y repetir este proceso hasta que todas las ciudades hayan sido visitadas una única vez y se haya vuelto a la primera ciudad.

Este algoritmo presenta uno de los mayores problemas de los Algoritmos Greedy. Si bien éste siempre nos dará una solución válida, no tiene en absoluto por qué ser una solución óptima. La razón de esto es que al escoger siempre mirando únicamente al siguiente paso, tiende a tomar elecciones que, aunque óptimas a corto plazo, pueden ser muy contraproducentes a largo plazo.

2.1.2. Pseudocódigo

```
función TSP_cercanía:
begin
C <- ciudades //Conjunto Candidatos
S <- C[0]     //Conjunto Solución, se puede empezar por cualquiera
V <- BLANK    //Conjunto Candidatos Usados
c <- S[0]     //Ciudad actual

  while (C !empty) do
     c <- c.menorCoste(C)//Funcion Selecciona entre Candidatos
     V <- c              //Actualizamos conjunto Candidatos Usados
     S <- c              //Actualizamos conjunto Solución
     C = C\c    //Retirnamos la ciudad del Conjunto Candidatos
  end while
end
```

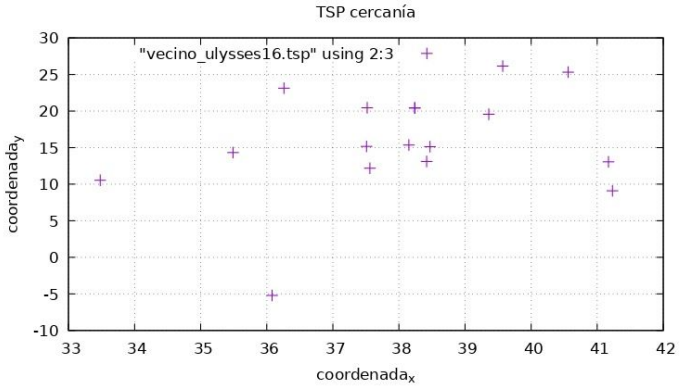
2.1.3. Generación del problema

El código recibe como entrada ficheros de datos que contienen ciudades con sus coordenadas x,y.
Éstas coordenadas son tratadas mediante una función para elaborar una tabla de distancias en la que se representa la distancia cartesiana de cada ciudad a todas las demás y es la que usaremos para simular esos vértices que serían las ciudades y las aristas que las unen siendo el valor correspondiente en la tabla de Distancias.

2.1.4. Supuesto de ejecución

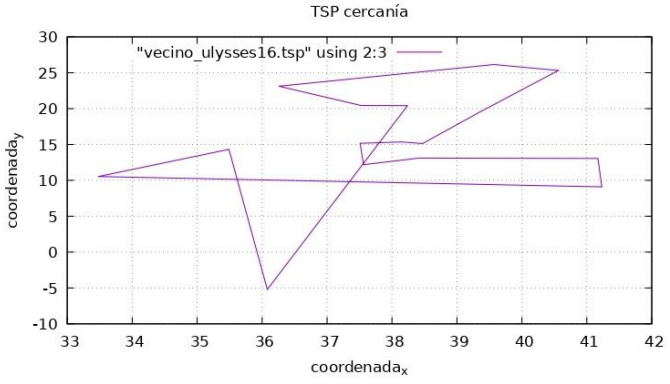
```
adduser@DESKTOP-UH8Q3EQ: /mnt/d/Jorge Gangoso Klock/Documentos/Trabajos Universidad/ALG/2020_Practica 3/Cercanía_Jorge_Gangoso_Klock/E...
adduser@DESKTOP-UH8Q3EQ:/mnt/d/Jorge Gangoso Klock/Documentos/Trabajos Universidad/ALG/2020_Practica 3/Cercanía_Jorge_Gangoso_Klock/Entradas$ ./programa ulysses16.tsp
NAME: ulysses16.tsp
1
8
4
2
3
16
12
13
14
6
7
10
9
5
15
11
1

Distancia total: 79
```



La ejecución mostrada se corresponde con la salida al problema de ulysses16, cuyas ciudades representadas se sitúan como aparece a la izquierda.

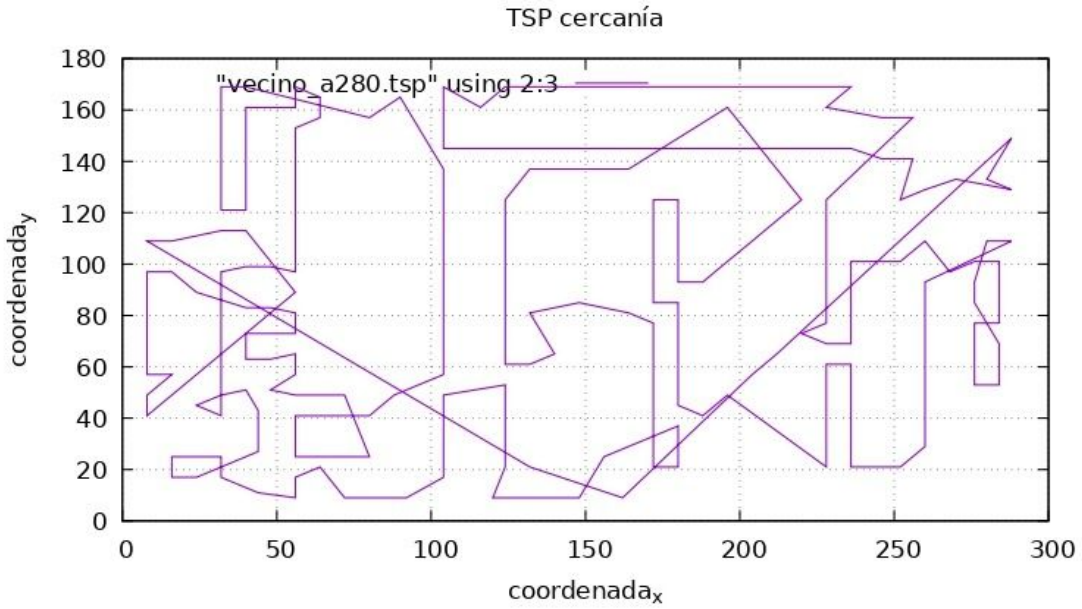
A continuación, vemos gráficamente como la heurística del vecino más cercano genera un camino con las ciudades seleccionadas, las cuales se muestran en la ejecución.



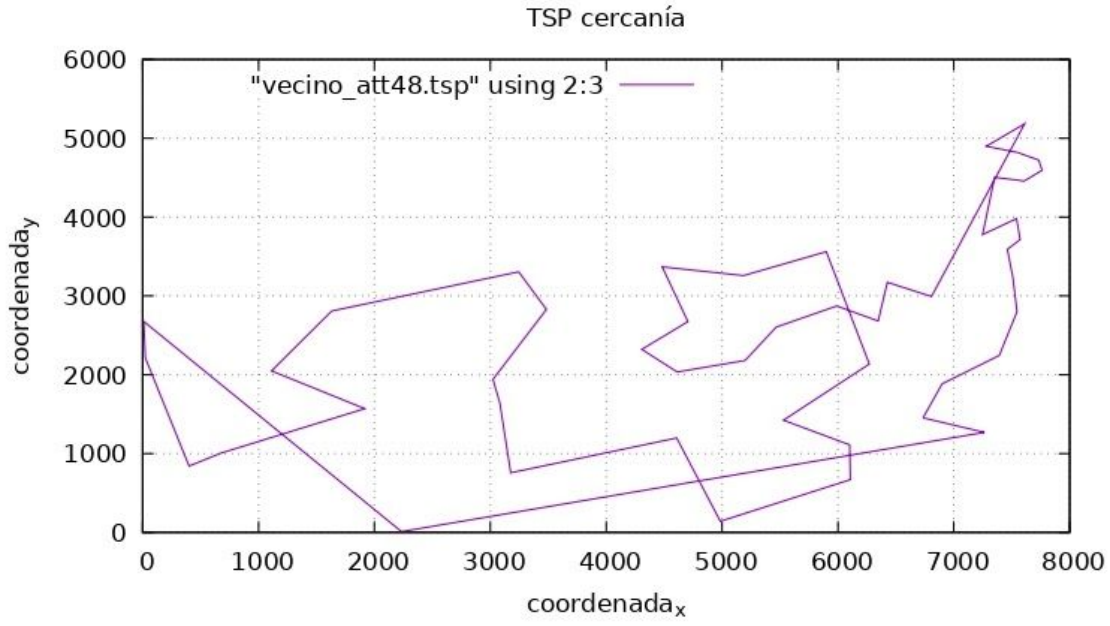
Vemos claramente como el momento en que se desvía a la ciudad más al sur conlleva una gran pérdida de eficiencia, siendo mejor visitarla en el transcurso entre la ciudad más al este y más al oeste.

Es un gran ejemplo de cómo el algoritmo genera una solución aproximada pero no óptima.

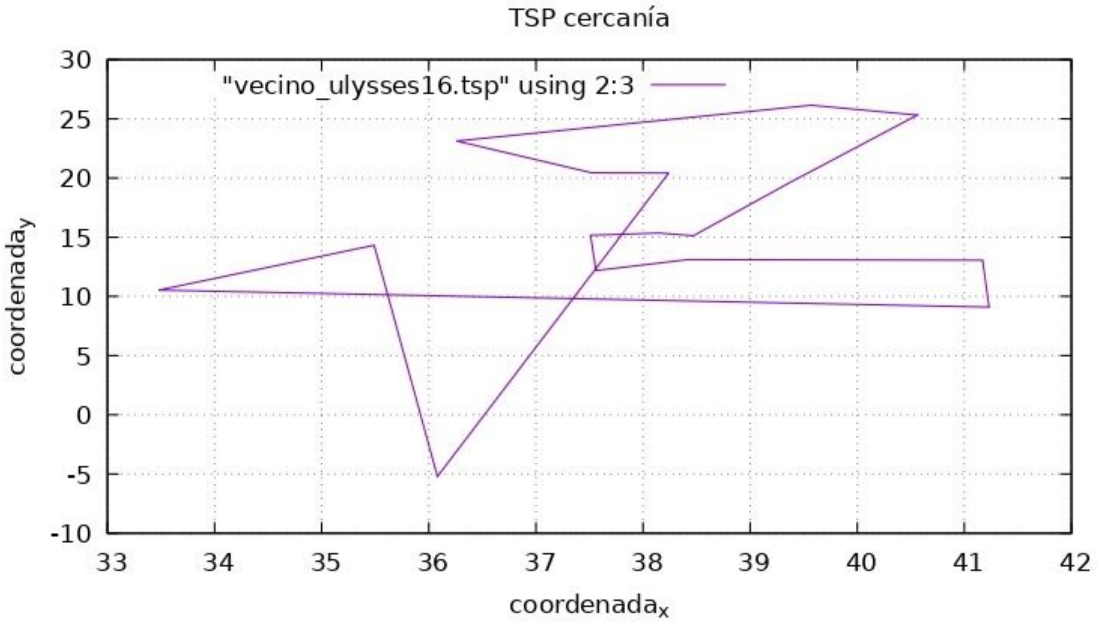
A continuación, los caminos generados y sus recorridos totales con todos los ficheros de datos:



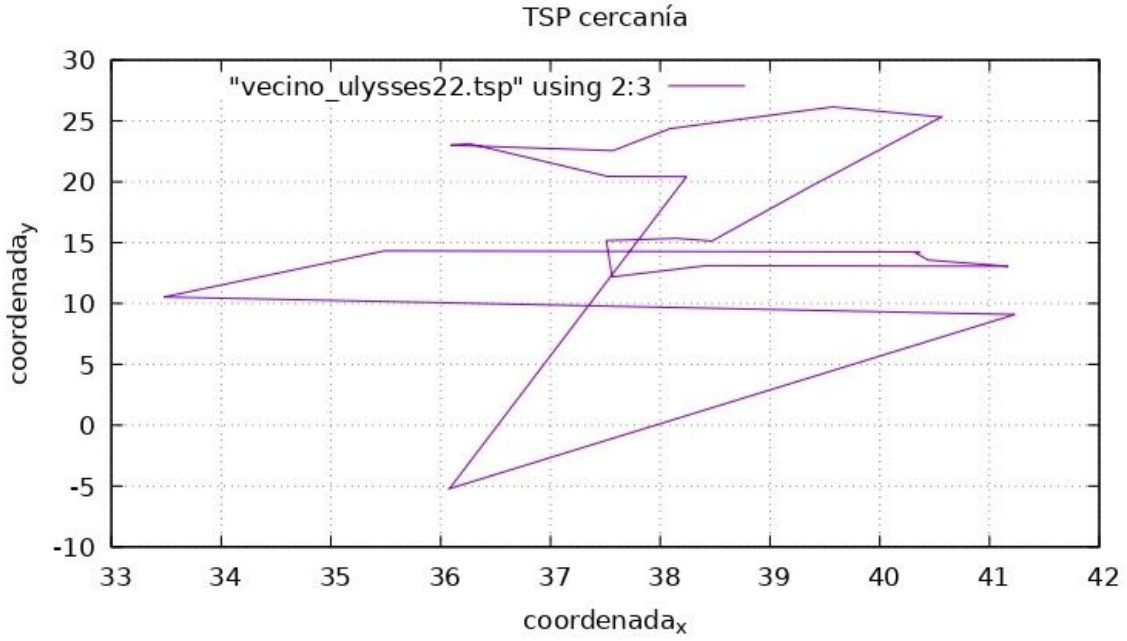
Recorrido Total “a280.tsp”: 3203



Recorrido Total “att48.tsp”: 40503



Recorrido Total "ulysses16.tsp": 79



Recorrido Total "ulysses22.tsp": 76

2.2. Por inserción

2.2.1. Descripción

- **Conjunto de candidatos:** ciudades más cercanas al ciclo inicial.
- **Conjunto de candidatos usados:** las ciudades añadidas al vector ciudades solución, que son eliminadas del vector ciudades.
- **Criterio de solución:** se ha encontrado el camino con menor costo.
- **Criterio de factibilidad:** En cualquier caso se puede llegar a una solución a partir del camino parcial.
- **Criterio de selección:** se escoge la ciudad más cercana al ciclo de ciudades solución.
- **Objetivo:** que el coste de la solución sea el mínimo posible.

Este método consiste en construir un camino parcial con los vértices más al norte, más al este y más al oeste. Este camino lo iremos extendiendo mediante la inserción de los vértices restantes.

En cada iteración se inserta un nuevo vértice en el camino solución y se elimina del vector de las ciudades. El criterio utilizado para elegir la siguiente ciudad del camino solución es el de la inserción más cercana, que selecciona la ciudad más cercana al ciclo.

2.2.2. Pseudocódigo

```
calcularRuta(var ciudades : vector que contiene las ciudades, var
ciudades_solucion : vector que contiene las ciudades ){

   obtenerCircuitoParcial(ciudades, ciudades_solucion);

   while ciudades.size do   //Mientras el vector ciudades no
                               este vacio
      seleccionarCiudad(ciudades, ciudades_solucion);
   end
end
```

El método `obtenerCircuitoParcial(vector<ciudad> &ciudades, vector<ciudad> &ciudades_solucion)` tiene el siguiente funcionamiento:

- Seleccionar las ciudades del vector ciudades que estén más al norte, al este y al oeste.
- Introducir esas ciudades en el vector solución.
- Eliminar esas ciudades del vector ciudades.

El método `seleccionarCiudad(vector<ciudad> &ciudades, vector<ciudad> &ciudades_solucion)` tiene el siguiente funcionamiento:

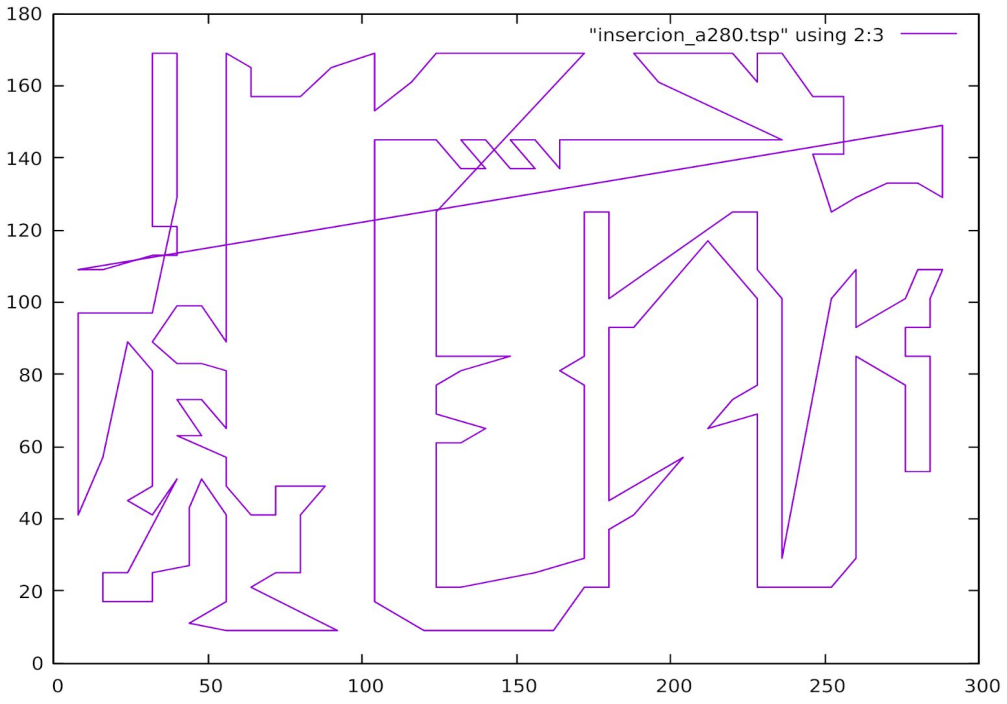
- Buscar la ciudad que esté más cercana al ciclo almacenado en el vector solución.
- Insertar dicha ciudad en el vector solución.
- Eliminar la ciudad del vector ciudades

2.2.3. Generación del problema

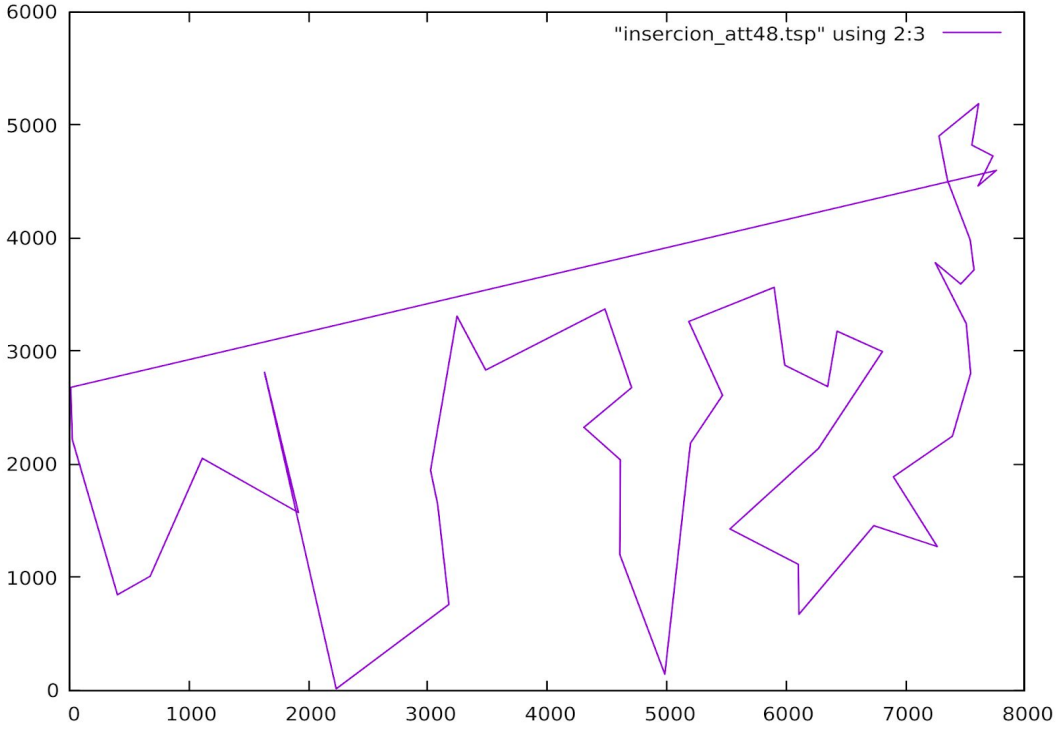
A través de unos ficheros TSP proporcionados se leen las coordenadas x e y de un conjunto de ciudades.

El algoritmo selecciona las ciudades más al norte, al este y al oeste y a partir de ese camino parcial va insertando las ciudades más cercanas al camino hasta encontrar una solución.

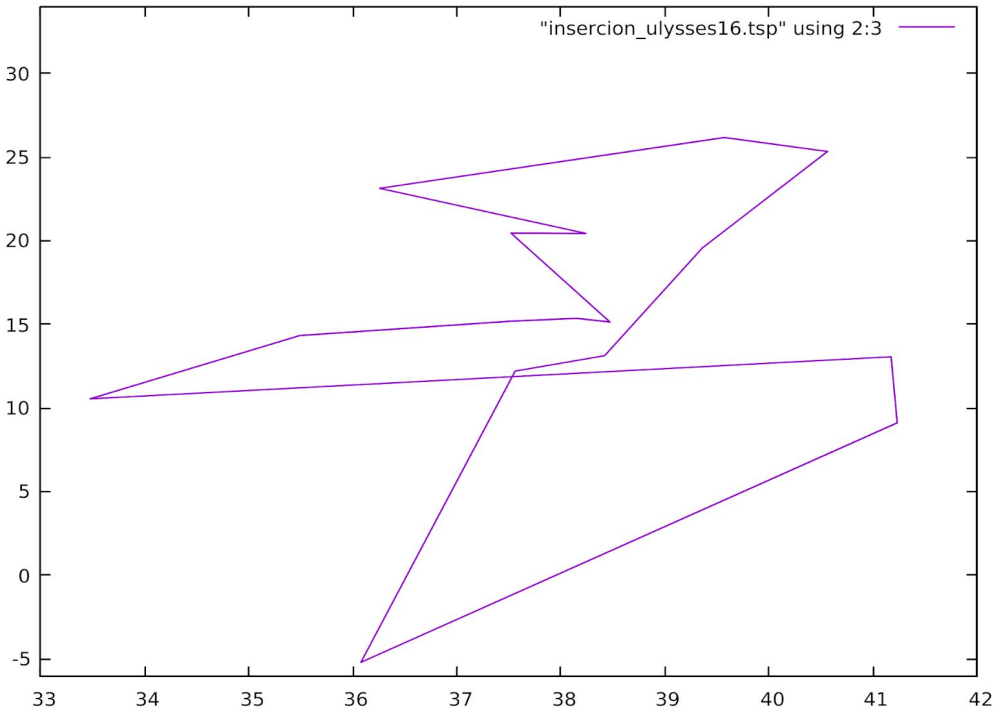
2.2.4. Supuesto de ejecución



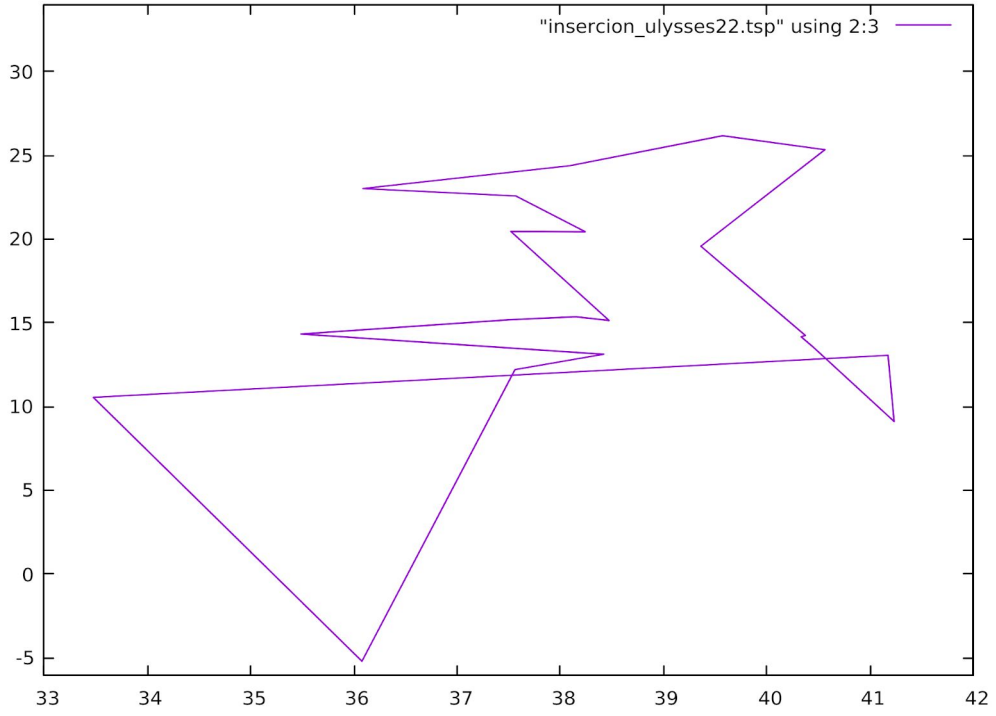
Recorrido total “a280.tsp”: 3223



Recorrido total “att48.tsp”: 42370



Recorrido total “ulysses16.tsp”: 81



Recorrido total “ulysses22.tsp”: 86

2.3. Método propio

El método que hemos diseñado está pensado como una solución alternativa observando diferentes caminos en cada iteración.

2.3.1. Descripción

Estando en una ciudad *(v0)*, comprueba cuál es su vecino más cercano *(v1)* y comprueba qué ciudad entre todas las del mapa tienen ciudades cercanas a igual o menor distancia que la que hay entre *v0* y *v1*. Tras esto comprueba si es más rentable ir a la otra ciudad y recorrer sus vecinas o ir a *v1*.
Esto lo consigue comprobando cuántos vecinos tiene esa otra ciudad, y suma la distancia necesaria para ir a esa ciudad y la distancia que tardaría en recorrer a sus vecinas. Esta distancia la divide entre el número total de ciudades que recorrería y si el resultado es menor que la distancia entre *v0* y *v1*, significa que merecerá la pena hacer ese recorrido.

Por ejemplo:

> Estamos situados en *v0*, y la ciudad más cercana está a 20m. Hay una ciudad que está a 50m *(vX)*, que tiene varias ciudades a 10m (supongamos que esta es la distancia entre ellas también). Si tuviera 2 ciudades a esta distancia, significa que para recorrer 3 ciudades *(vX + sus vecinas)* recorre 70 metros, entre 3 da un total de 23.3m, que es más que la distancia entre *v0* y *v1*, por lo tanto **NO** es rentable seguir este recorrido y nos moveremos a *v1*. Sin embargo, si esta ciudad tuviera 4 ciudades, significa que recorrería 5 ciudades en 90m, entre 5 da 18m, que es menor que la distancia entre *v0* y *v1*, por lo que **SÍ** es rentable seguir este camino.

Este algoritmo, sobre el papel, parece encontrar un camino más óptimo al camino por cercanía, ya que valora si es mejor moverse a la ciudad cercana o hacer otro recorrido, pero más adelante observaremos qué ocurre en la práctica.

- **Conjunto de candidatos:** ciudad más cercana y ciudades junto con sus ciudades vecinas
- **Conjunto de candidatos usados:** los candidatos usados se eliminan del vector `ciudades` y se almacenan en el vector `camino`.
- **Criterio de solución:** se recorren todas las ciudades, volviendo a la ciudad de partida.
- **Criterio de factibilidad:** no se puede recorrer una ciudad dos veces.
- **Criterio de selección:** se escoge la ciudad o recorrido que menor coste tenga por ciudad visitada.
- **Objetivo:** que el coste de la solución sea el mínimo posible.

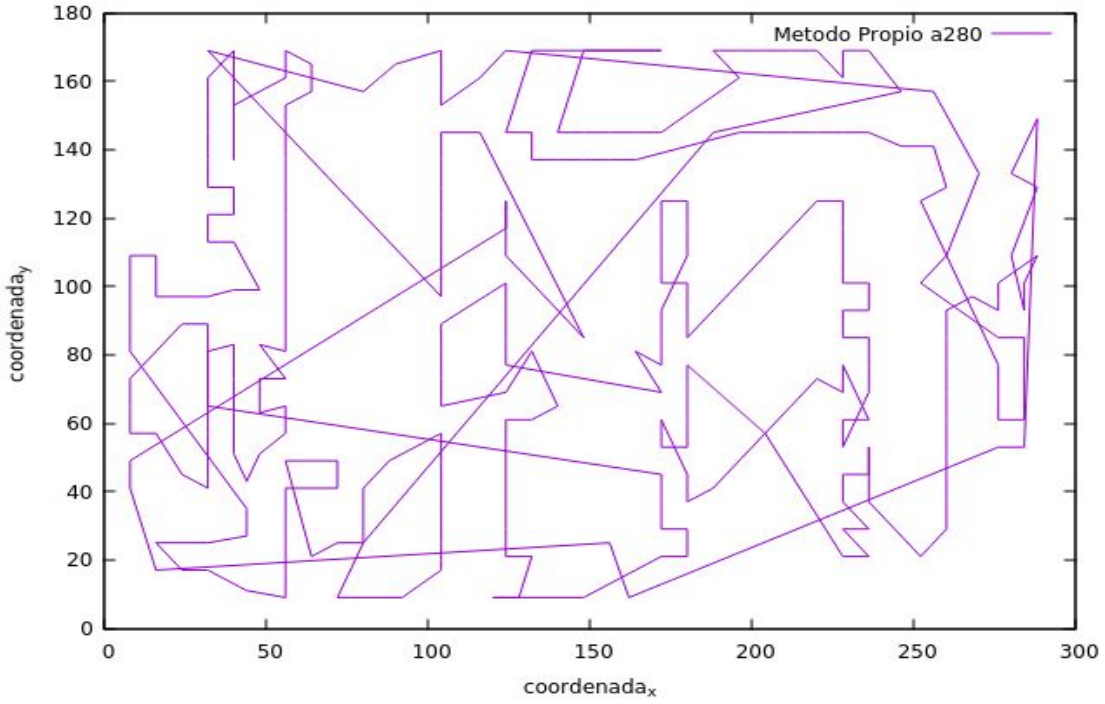
2.3.2. Pseudocódigo

```
while(camino.size < ciudades.size) //Mientras que el camino no se
complete
   if(recorrido < vecino) //Si es mejor el recorrido
      camino.add(recorrido) //Añado todo el recorrido
   else
      camino.add(vecino) //Si no, añado el vecino
```

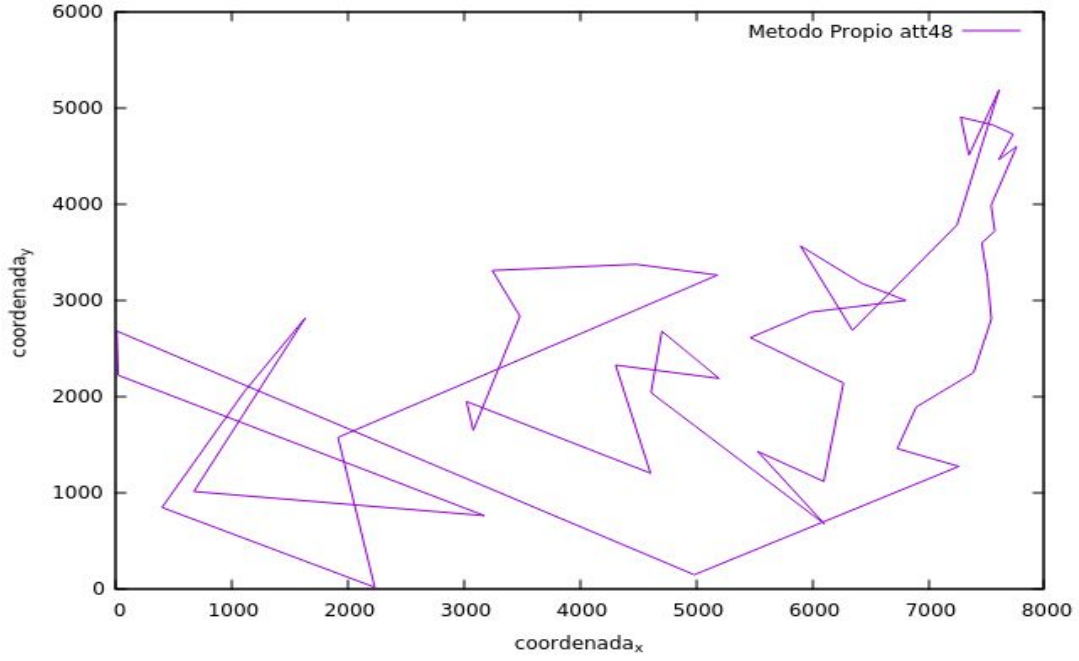
2.3.3. Generación del problema

A través de unos ficheros TSP proporcionados se leen las coordenadas de un conjunto de ciudades. El algoritmo, mediante los cálculos anteriormente explicados, introduce las ciudades según convenga en un vector resultado.

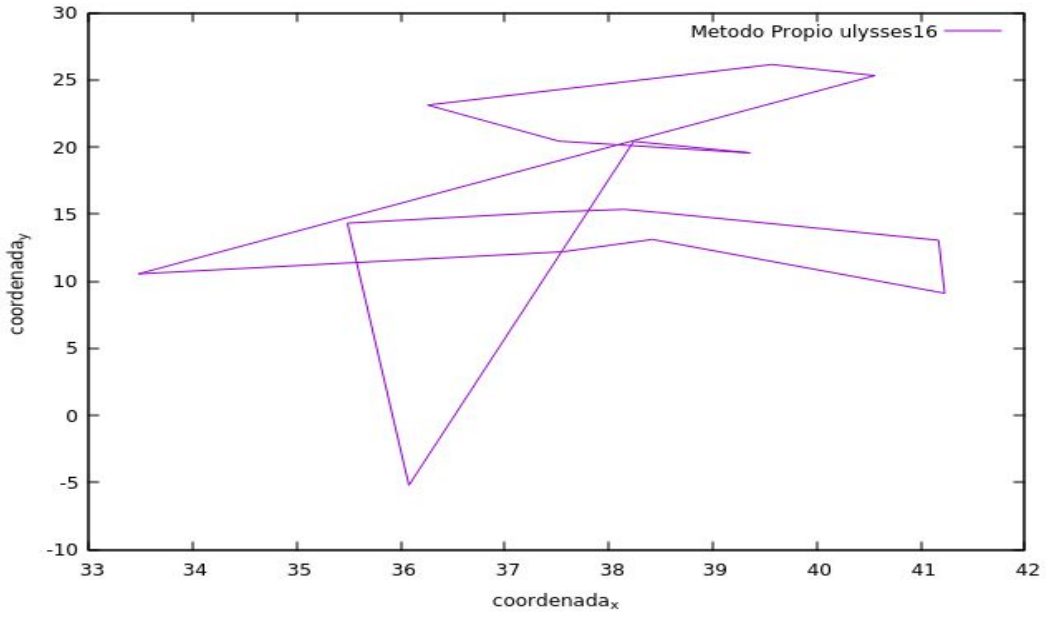
2.3.4. Supuesto de ejecución



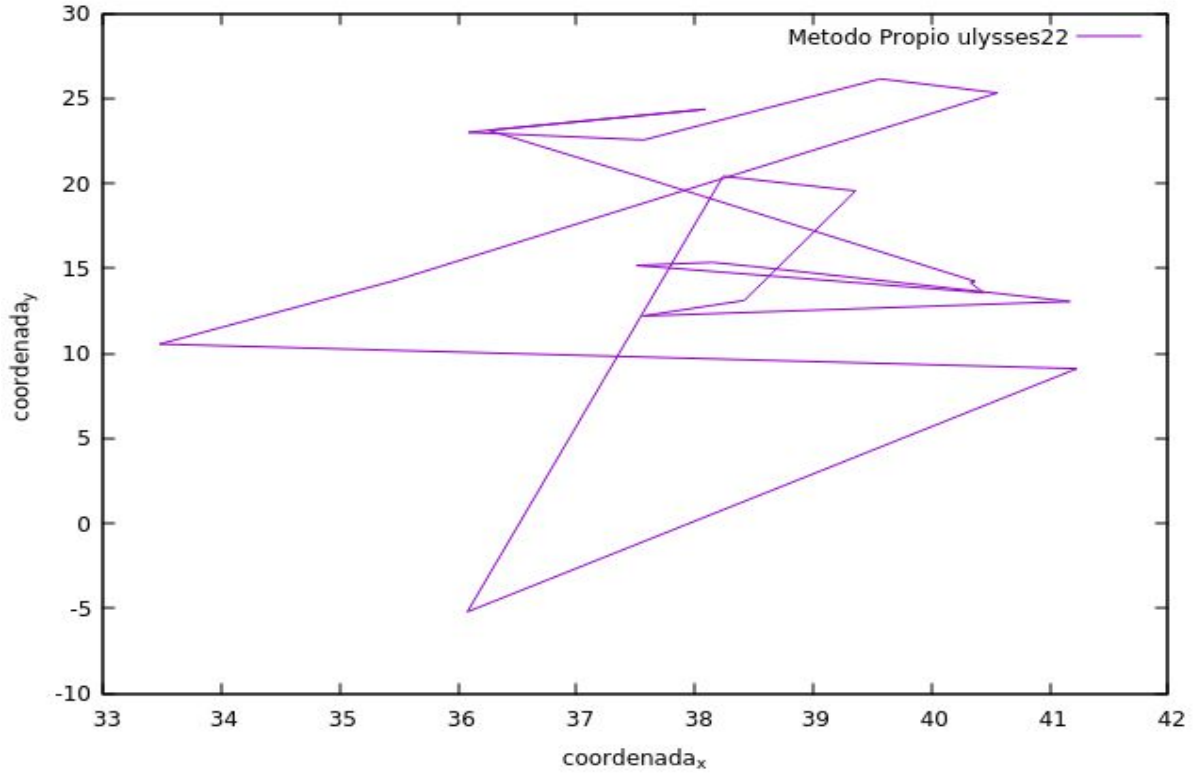
Supuesto de ejecución: a280
Total del recorrido: 4229



Supuesto de ejecución: att48
Total del recorrido: 53542



Supuesto de ejecución: ulysses16
Total del recorrido: 87



Supuesto de ejecución: ulysses22
Total del recorrido: 98

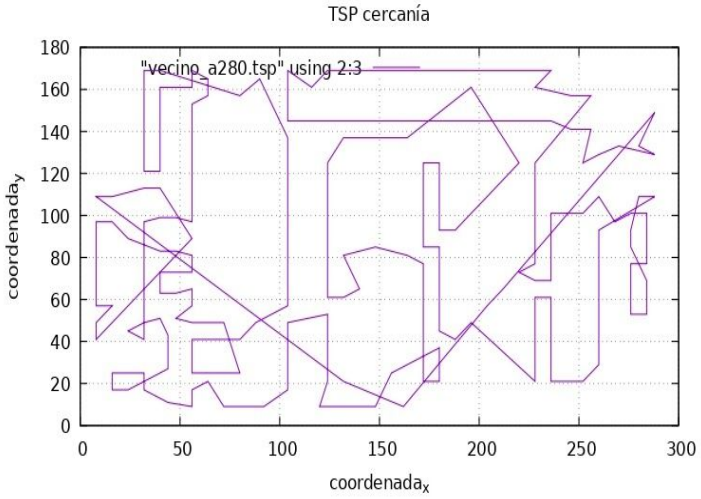
2.4. Comparativa

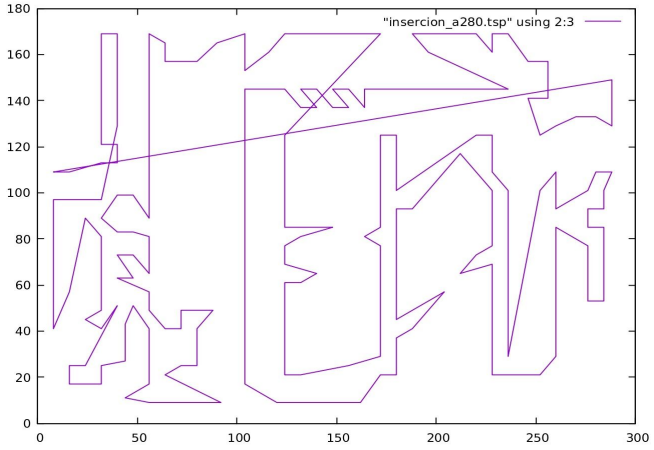
Comparación de optimalidad de los tres códigos utilizando las distintas entradas de datos:

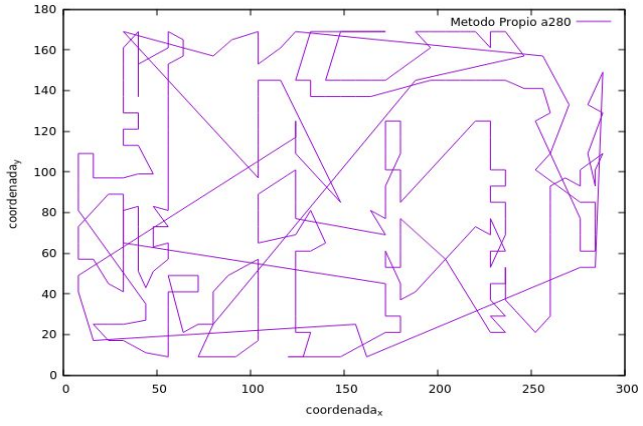
Fichero de datos : “a280.tsp”

Distancias:

1. Cercanía: **3203**
2. Inserción: **3223**
3. Propio: **4229**



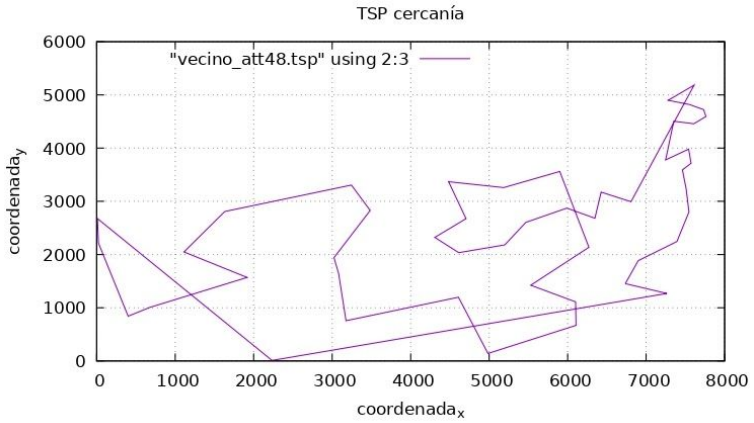


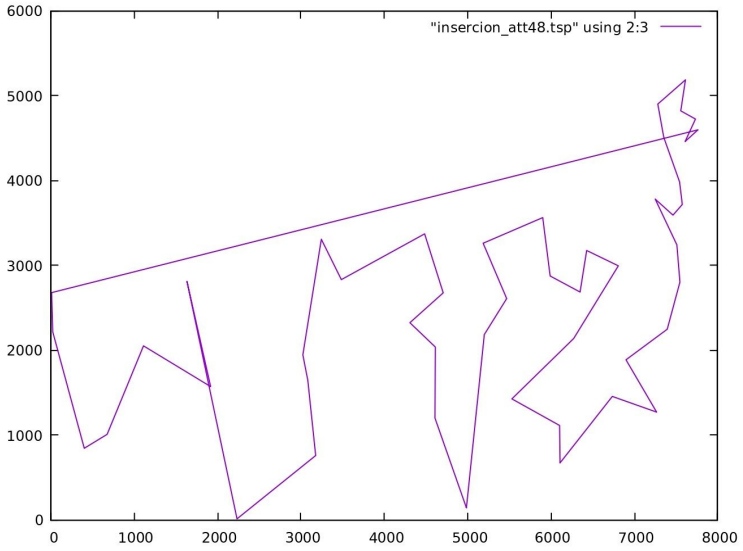


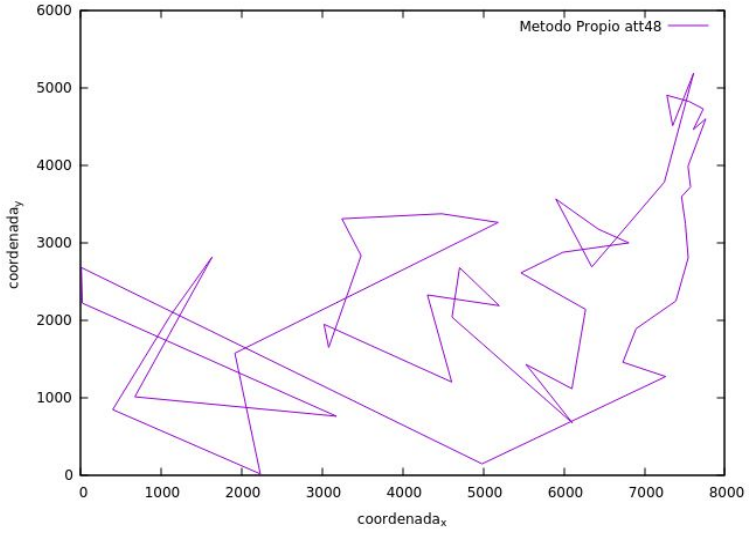
Fichero de datos: “att48.tsp”

Distancias:

1. Cercanía: **40503**
2. Inserción: **42370**
3. Propio: **53542**



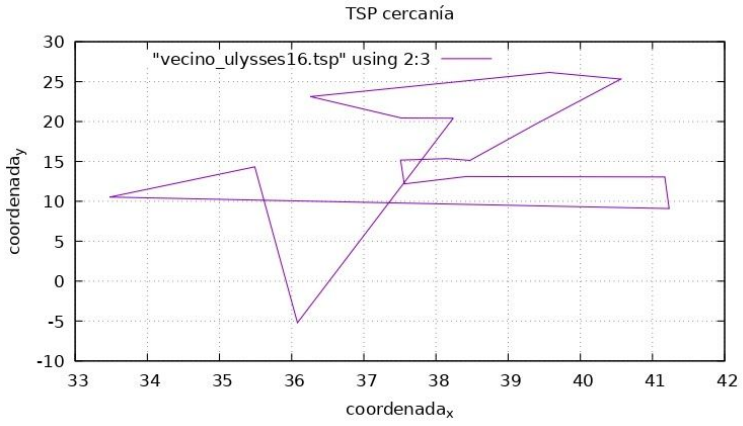


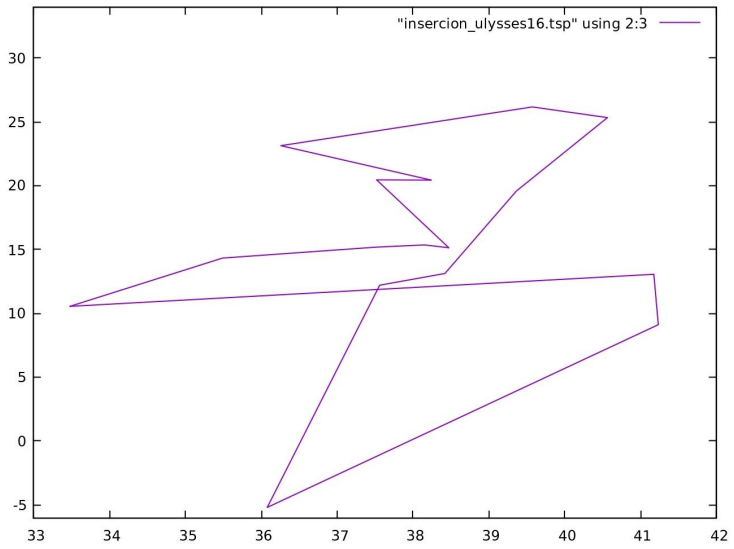


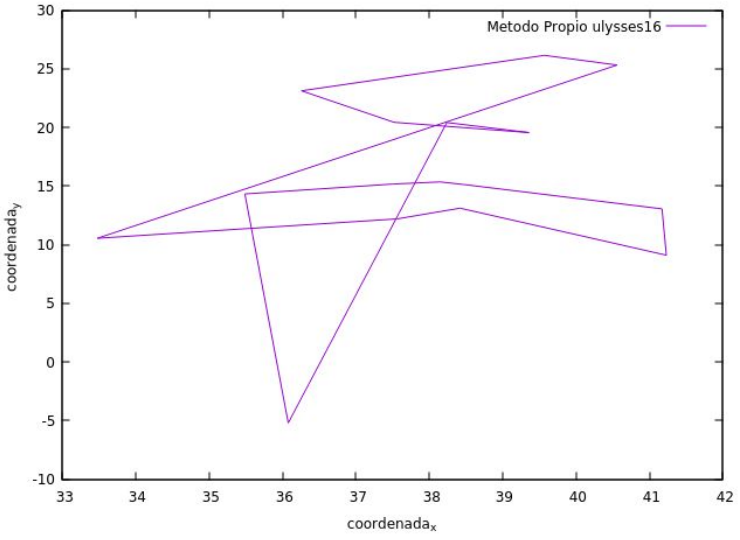
Fichero de datos: “ulysses16.tsp”

Distancias:

1. Cercanía: **79**
2. Inserción: **81**
3. Propio: **87**



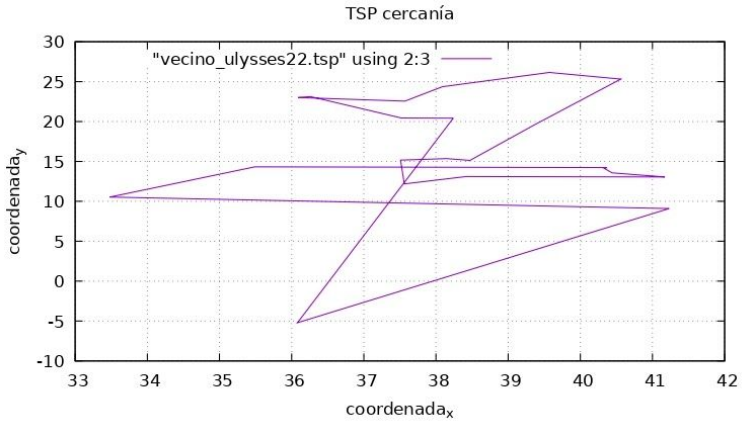


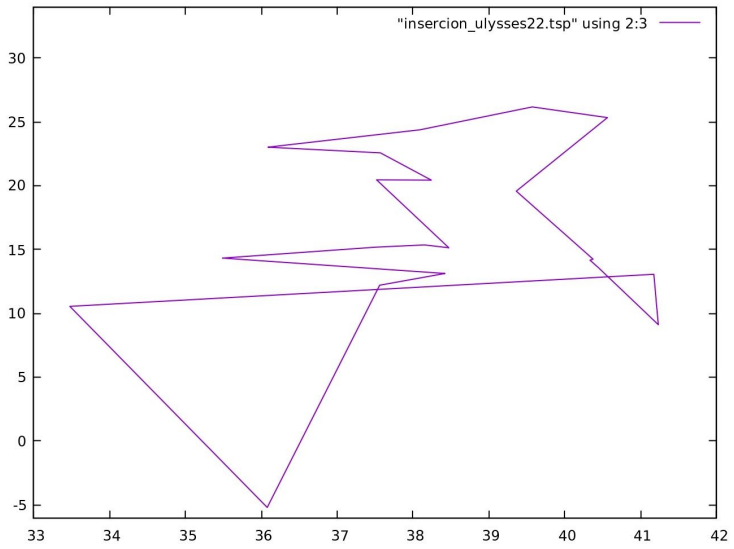


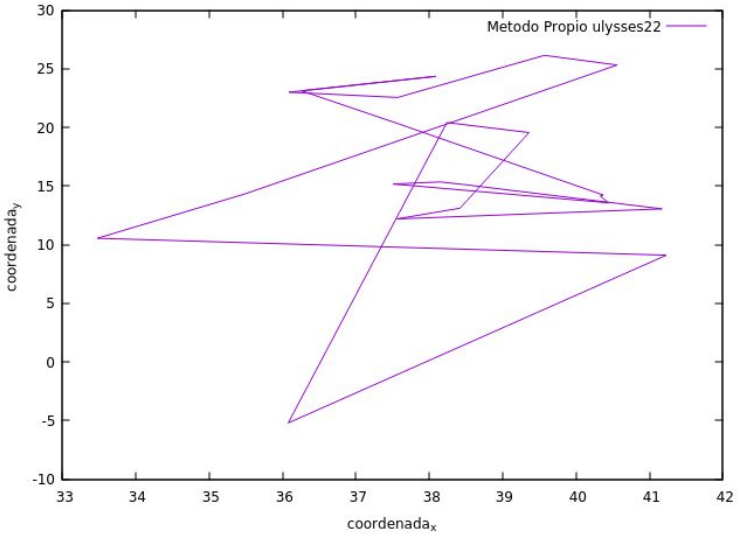
Fichero de datos: “ulysses22.tsp”

Distancias:

1. Cercanía: **76**
2. Inserción: **86**
3. Propio: **98**







3. Problema de los trabajadores y tareas

Supongamos que disponemos de *n* trabajadores y *n* tareas. Sea $c_{i,j} > 0$ el coste de asignarle la tarea *j* al trabajador *i*. Una asignación válida es aquella en la que a cada trabajador le corresponde una tarea y cada tarea la realiza un trabajador diferente.

Dada una asignación válida, definimos el coste de dicha asignación como la suma total de los costes individuales.

3.1. Descripción

- **Conjunto de candidatos:** precio asociado de cada trabajador a cada tarea (n^2 candidatos)
- **Conjunto de candidatos usados:** los definiremos y almacenaremos en el vector de enteros llamado `asignacion[n]`.
- **Criterio de solución:** todas las tareas han sido asignadas. Lo almacenaremos en la variable `asignados`.
- **Criterio de factibilidad:** que sea un trabajador no ocupado (lo comprobaremos en el vector de booleanos llamado `ocupados[n]`) y una tarea no asignada (lo comprobaremos en el mismo vector solución).
- **Criterio de selección:** se escoge el coste menor siempre y cuando ese valor no haya sido seleccionado como mínimo anteriormente.
- **Objetivo:** que el coste final sea el mínimo posible.

Dada la lista de candidatos, la recorre, selecciona el coste menor no registrado anteriormente; y si es factible lo añade al vector solución. Es un algoritmo estrictamente greedy que cumple todas sus condiciones, si bien **no siempre devuelve la solución más óptima.** Este problema se resuelve de forma óptima empleando el Algoritmo húngaro, pero su enfoque no es greedy, y su implementación es costosa.

3.2. Pseudocódigo

```
AsignacionGreedy(var M: matriz de costes, var n:numero de
trabajadores, var V:Vector asignacion)

   for i:=0..n do   //La asignacion empieza vacía
      V[i] := -1
      ocupado[i] := FALSE
   end

   asignados := 0
   seleccionado := 0

   while asignados < n do
      seleccionado := EncuentraMasBarato(M, n, M_aux)

      if !ocupado[seleccionado.trabajador] AND
         asignacion[seleccionado.tarea] == -1
      then
         asignacion[seleccionado.tarea] = seleccionado.trabajador
         ocupado[seleccionado.trabajador] = TRUE
         asignados++
   end
end
```

La función de selección `EncuentraMasBarato(int **M, int tamanio, bool **M_aux)` y la factibilidad funcionan de la siguiente manera:

- La función de selección devuelve la posición del valor más bajo que encuentra mediante un struct. El valor más bajo se marca como usado en la `M_aux`
- En cada iteración del bucle `while` se comprueba que el candidato seleccionado es factible: ni el trabajador está ocupado, ni la tarea ya asignada
- Si el candidato es factible, se añade al vector solución y se avanza en la asignación

3.3. Generación del problema

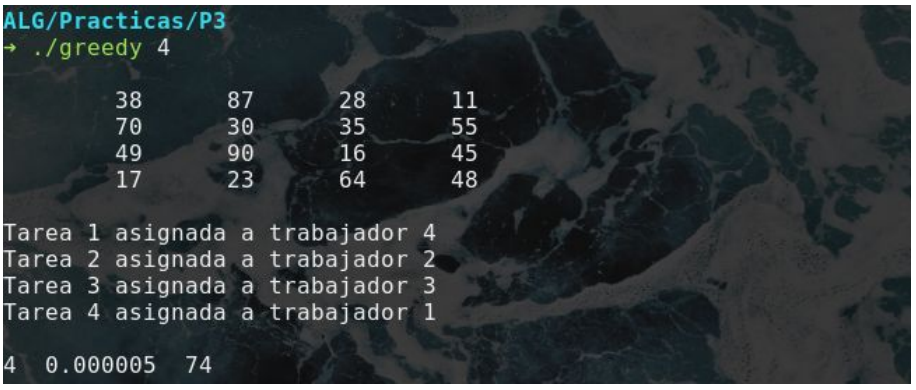
Dispondremos de una matriz de enteros, cuadrada, de un tamaño *n* (introducido al ejecutar) que se rellenará con números enteros aleatorios del 10 al 100. La semilla que se proporciona para la aleatoriedad está basada en el tiempo de ejecución.

Si introducimos una ‘D’ después de la *n* al ejecutar optamos a introducir la matriz deseada a mano. Esto obviamente es una herramienta de debug y no se permite usarla en matrices de dimensión mayor de 4. Tampoco la tendremos en cuenta en mediciones de tiempo.

3.4. Supuesto de ejecución

Compilamos el programa:

Y lo ejecutamos. Existen dos modos de ejecución: automático, y debug. El modo automático se ejecuta con `./greedy <n>`, y genera una matriz aleatoria:

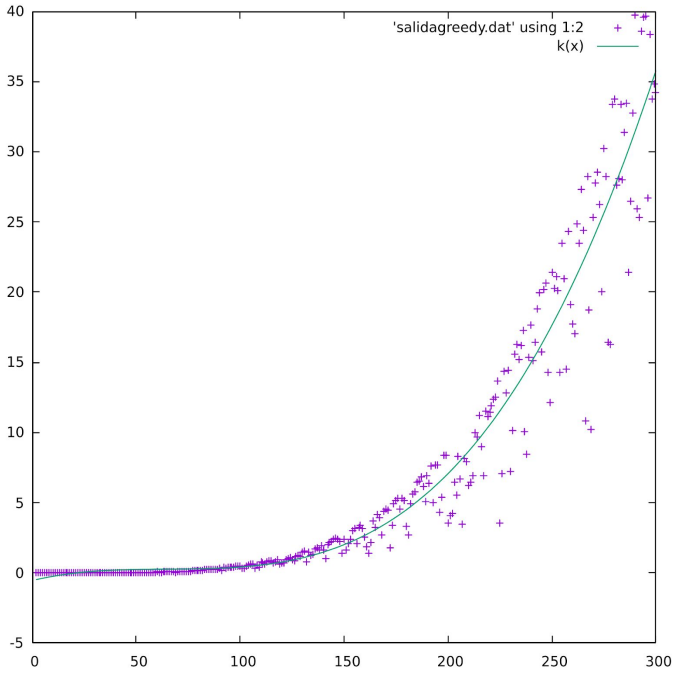


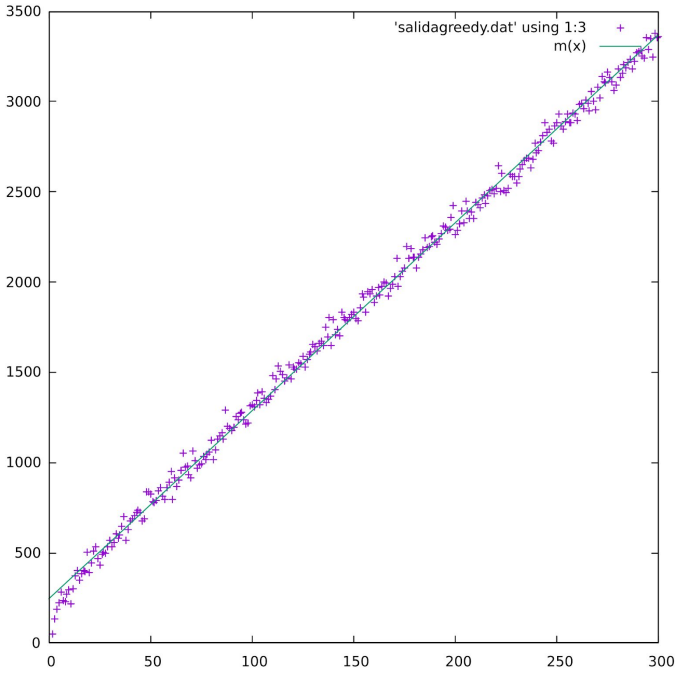
En este supuesto, la ejecución ha sido favorable. A continuación, usando el modo debug (`./greedy <n> D`), vamos a introducir manualmente una matriz no favorable:

Esta salida, como podemos comprobar, es errónea: la salida óptima sería 83.

3.5. Gráficas

Antes que nada, mencionar que, al ser matrices generadas aleatoriamente, las gráficas obtenidas tienen mucho ruido, ya que su procesamiento depende altamente de la disposición y valores de la matriz.





A la izquierda tenemos los **tiempos** de procesado de la matriz. A la derecha, los **costes** obtenidos en cada ejecución. El tiempo encaja con una eficiencia del tipo $O(n^3)$, mientras que los costes suben de forma lineal. El coste medio es de 1818’7.

Cabe destacar, además, que la generación de la matriz aleatoria es, con diferencia, lo que más tiempo de procesamiento ocupa. Si bien los tiempos obtenidos en las dimensiones mayores ronda los 30 segundos, la ejecución tomó realmente **más de 40 minutos** para la recopilación de los datos de las 300 instancias.

4. Conclusiones

Las conclusiones que hemos inferido de esta práctica son las siguientes:

- Los algoritmos greedy, si bien **presentan siempre una solución válida, no tiene por qué ser la óptima**. Esto se debe a la misma naturaleza de estos: la toma de decisiones rápidas y sin posibilidad de cambiarlas nos puede dejar con soluciones que en el momento en que se tomaron eran la mejor opción, pero a la larga nos pueden perjudicar enormemente.

  El orden a seguir de *selección antes que factibilidad* no nos permite un preprocesamiento del problema que podría ser favorecedor para la obtención de la solución óptima

- En esta práctica, además, hemos comprobado que un algoritmo en teoría superior a otro puede no serlo aplicando la técnica greedy (como es el caso del TSP de cercanía sobre inserción y método propio)

    - Esto es porque, aunque teóricamente el algoritmo que hemos diseñado nosotros en cada iteración elige qué camino va a tener un menor coste, si elige desplazarse a otra ciudad más lejana en vez de a la vecina, en algún momento tendrá que volver a la vecina, por lo que se produce unas idas y venidas que desequilibran el problema.