



# UNIVERSIDAD DE GRANADA

Escuela Técnica Superior de Ingeniería Informática y  
Telecomunicaciones

## PRÁCTICA 3: ALGORITMOS GREEDY

*Doble Grado Ingeniería Informática y Matemáticas*

Autores:

Jose Alberto Hoces Castro

Javier Gómez López

Moya Martín Castaño

Mayo 2022



Este trabajo se distribuye bajo una licencia CC BY-NC-SA 4.0.

Eres libre de distribuir y adaptar el material siempre que reconozcas a los autores originales del documento, no lo utilices para fines comerciales y lo distribuyas bajo la misma licencia.

[creativecommons.org/licenses/by-nc-sa/4.0/](https://creativecommons.org/licenses/by-nc-sa/4.0/)

# Índice

<b>1. Introducción</b>	<b>3</b>
<b>2. Desarrollo</b>	<b>3</b>
2.1. Ejercicio 1. Contenedores . . . . .	3
2.1.1. Primer ejercicio . . . . .	3
2.1.2. Segundo ejercicio . . . . .	4
2.2. Ejercicio 2. El problema del viajante de comercio . . . . .	5
2.2.1. Heurística del vecino más cercano . . . . .	6
2.2.2. Heurística por inserción . . . . .	8
2.2.3. Heurística propia: perturbaciones . . . . .	8
2.2.4. Comparación de heurísticas . . . . .	11
<b>3. Conclusiones</b>	<b>11</b>

# 1. Introducción

El objetivo de esta práctica es aprender a implementar y utilizar algoritmos “*greedy*” o voraces para resolver problemas de manera rápida aunque no por ello menos óptima. Para ello, se plantean los siguientes dos problemas:

- **Ejercicio 1** (Contenedores): Se quiere rellenar un buque mercante con una cierta capacidad de peso con contenedores, cada uno de los cuales tiene su propio peso.
- **Ejercicio 2** (TSP): El problema del viajero. Se quiere recorrer una serie de ciudades, pasando por ellas solo una vez y volviendo al punto de partida. Se quiere encontrar la ruta más óptima.

# 2. Desarrollo

Para el primer ejercicio, nos centraremos en identificar el problema de los contenedores como un problema Greedy identificando sus características. También, justificaremos la optimalidad o la no optimalidad de los algoritmos Greedy desarrollados.

Para el análisis de los algoritmos del viajante de comercio que desarrollaremos, hemos realizado los siguientes pasos:

1. Un **análisis teórico** donde se comentará el código desarrollado para la resolución de los problemas propuestos.
2. Un **análisis empírico** donde hemos ejecutado los algoritmos en nuestros ordenadores bajo las mismas normas y condiciones. Hemos compilado usando la optimización `-Og`. Además, hemos usado como *datasets* de pruebas los datos proporcionados por la profesora en el caso del TSP, y valores aleatorios de los pesos de los contenedores para el primer problema. Por otro lado, para automatizar el proceso, hemos creado unos *scripts* de generación de datos de prueba y de ejecución de nuestros programas. Hemos ejecutado cada algoritmo 15 veces en cada uno de los tamaños probados, y hemos hecho la media de ellos para reducir perturbaciones que puedan alterar el resultado.
3. Un **análisis híbrido** donde hemos tomado los datos de cada uno de los alumnos del grupo y hemos hallado la  $K$  (constante oculta). Para ello hemos usado `gnuplot`.

## 2.1. Ejercicio 1. Contenedores

El enunciado del problema es el siguiente: *Se tiene un buque mercante cuya capacidad de carga es de  $K$  toneladas y un conjunto de contenedores  $c_1, \dots, c_n$  cuyos pesos respectivos son  $p_1, \dots, p_n$  (expresados también en toneladas). Teniendo en cuenta que la capacidad del buque es menor que la suma total de los pesos de los contenedores:*

- Diseñe un algoritmo que maximice el número de contenedores cargados, y demuestre su optimalidad.
- Diseñe un algoritmo que intente maximizar el número de toneladas cargadas.

### 2.1.1. Primer ejercicio

Nuestro objetivo es que podamos cargar el máximo número de contenedores en un buque mercante de  $K$  toneladas. Para ello, el algoritmo que nosotros proponemos es tomar los contenedores, ordenarlos de menor a mayor peso, y comenzar añadiendo los de menor peso. De esta forma, podemos cargar más contenedores, ya que si empezásemos por los de pesos intermedios o mayores, acabaríamos cargando menos.

Veamos las 6 características de nuestro problema Greedy:

- **Un conjunto de candidatos:** En este caso, los contenedores a cargar.
- **Una lista de candidatos ya usados:** Los contenedores que ya han sido cargados.
- **Un criterio que dice cuándo un conjunto de candidatos forma una solución:** El criterio es que la suma de los pesos de un conjunto de contenedores no sea superior a las  $K$  toneladas del buque.
- **Un criterio que dice cuándo un conjunto de candidatos es factible (podrá llegar a ser una solución):** el conjunto de contenedores que se evalúe no debe superar en peso las  $K$  toneladas del buque.

- **Una función de selección que indica en cualquier instante cuál es el candidato más prometededor de los no usados todavía:** El contenedor de menor peso de los que aún no están cargados, de ahí que los ordenemos de menor a mayor peso.
- **La función objetivo que intentamos optimizar:** El número de contenedores a cargar, es lo que queremos maximizar.

A continuación pasamos a mostrar el código de nuestro algoritmo. Para representar el conjunto de contenedores, hemos considerado un vector de enteros, siendo cada entero el peso de cada contenedor. Para asegurarnos de que la suma de los pesos de todos los contenedores supera a las K toneladas del buque, el vector es de dimensión K con enteros aleatorios desde 0 hasta K (en nuestro código lo representamos por el parámetro n, introducido por el usuario). Nuestra función devuelve el número de contenedores que se han podido cargar:

```
int contenedoresGreedy1(int *T, int n){
    int used = 0;
    int result = 0;
    vector<int> myvector(T,T+n);
    sort(myvector.begin(),myvector.end());

    for(int i = 0; (i < n) && (used <= n); i++){
        used += T[i];
        result++;
    }

    return result;
}
```

- **Estudio de la optimalidad**

### 2.1.2. Segundo ejercicio

En este segundo ejercicio, lo que queremos es maximizar las toneladas cargadas en el buque sin sobrepasar su capacidad total. Para ello, seguimos el pensamiento inverso al planteado en el anterior ejercicio. Como lo que nos interesa es cargar el máximo de toneladas posibles, empezaremos cargando aquellos contenedores cuyo peso sea el más grande. Para ello, los ordenamos de mayor a menor peso, justo al contrario que antes. Identificamos las 6 características de un problema Greedy:

- **Un conjunto de candidatos:** En este caso, los contenedores a cargar.
- **Una lista de candidatos ya usados:** Los contenedores que ya han sido cargados.
- **Un criterio que dice cuándo un conjunto de candidatos forma una solución:** El criterio es que la suma de los pesos de un conjunto de contenedores no sea superior a las K toneladas del buque.
- **Un criterio que dice cuándo un conjunto de candidatos es factible (podrá llegar a ser una solución):** el conjunto de contenedores que se evalúe no debe superar en peso las K toneladas del buque.
- **Una función de selección que indica en cualquier instante cuál es el candidato más prometededor de los no usados todavía:** El contenedor de mayor peso de los que aún no están cargados, de ahí que los ordenemos de mayor a menor peso.
- **La función objetivo que intentamos optimizar:** El número de toneladas a cargar, es lo que queremos maximizar.

En el código del algoritmo, al igual que antes, simulamos los contenedores con sus respectivos pesos con un vector de enteros, el cual ordenamos de mayor a menor usando el sort de la STL. Se van sumando los pesos hasta que se sobrepase el tope de toneladas del buque. Nuestra función devuelve el número de toneladas que se han podido cargar en total:

```
int contenedoresGreedy2(int *T, int n){
    int used = 0;
    vector<int> myvector(T,T+n);
    sort(myvector.begin(),myvector.end(), greater<int>());

    for(int i = 0; (i < n) && (used <= n); i++){
```

```

        used += T[i];
    }
    return used;
}

```

### ■ Estudio de la optimalidad

Sin embargo, en este caso nuestro algoritmo no nos da la solución óptima. Veámoslo con un contraejemplo. Con  $n = 10$ , imaginemos que tenemos el vector  $[5, 4, 6, 1, 1, 2, 7, 9, 8, 3]$ . Nuestro algoritmo lo ordenaría de mayor a menor, obteniendo el vector  $[9, 8, 7, 6, 5, 4, 3, 2, 1, 1]$ . Tras esto, se incluiría el contenedor de peso 9, pero el siguiente ya no sería posible cargarlo ya que  $9 + 8 = 17 > 10$ . La solución óptima en este caso sería tomar un contenedor de peso 1, otro de 2, otro de 3 y otro de 4, aprovechando así las 10 toneladas en su totalidad.

## 2.2. Ejercicio 2. El problema del viajante de comercio

El enunciado del problema es el siguiente: *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.*

Además, se nos pide enfocarlo usando dos heurísticas distintas:

- **Vecino más cercano:** dada una ciudad inicial  $v_0$ , se agrega como ciudad siguiente aquella  $v_i$  (no incluida en el circuito) que se encuentre más cercana a  $v_0$ . El procedimiento se repite hasta que todas las ciudades se hayan visitado.
- **Inserción:** la idea es comenzar con un recorrido parcial, que incluya algunas de las ciudades, y luego extender este recorrido insertando las ciudades restantes mediante algún criterio de tipo greedy.

Además, se debe proponer por parte del equipo otra heurística propia para resolver el problema.

### Elementos comunes

A lo largo de la solución del problema usaremos la siguiente notación para todas las heurísticas a desarrollar:

- $n$  es el **número de ciudades**.
- $D$  es la **matriz de distancias**.
- $r$  es el **vector de recorrido**, que contiene una ruta que pasa por todas las ciudades, es decir,  $n$  elementos no repetidos.
- $W_r$  es el **coste** de un recorrido, es decir, la distancia de un recorrido  $r$ .

Adicionalmente, se han implementado una serie de funciones comunes a todas las heurísticas trabajadas en esta práctica:

```

#Creating a struct point
@dataclass
class Point:
    x: float
    y: float

def parse_input(input):
    points = []

    with open(input, "r") as archive:
        archive.readline()

        for line in archive.readlines():
            line = line.strip().split(' ')

            p = Point(float(line[1]), float(line[2]))

            points.append(p)
        return points

def distance(p1, p2):
    if p1 != p2:
        result = sqrt(pow((p1.x - p2.x), 2) + pow((p1.y - p2.y), 2))

```

```

        result = round(result.real)
    else:
        result = 0
    return result

def gen_distance_matrix(points, num_cities):
    matrix = [[0 for x in range(num_cities)] for x in range(num_cities)]

    for i in range(num_cities):
        for j in range(num_cities):
            matrix[i][j] = distance(points[i], points[j])

    return matrix

def get_road_distance(road):
    road_distance = 0

    for i in range(len(road)-1):
        road_distance += distance(road[i], road[i+1])

    #First and last point
    road_distance += distance(road[0], road[len(road)-1])

    return road_distance

```

Hemos definido una “estructura” `Point` para representar los puntos, una función `parse_input` para parsear el input según el formato dado en los archivos de clase, `distance` que mide la distancia entre dos puntos, `gen_distance_matrix` que genera la matriz de distancias dada una lista de puntos y `get_road_distance` que da la longitud de un vector de recorrido.

Cabe destacar que los datos que se van a mostrar son de los *datasets* dados por la profesora, pero se han trabajados con conjuntos de datos más grandes. En estas experiencias con datos más grandes hemos observado la importancia de jugar con más de un tipo distinto de enfoque *greedy* pues en algunos casos el tiempo de ejecución puede ser demasiado alto.

### 2.2.1. Heurística del vecino más cercano

Este algoritmo *greedy* es muy simple:

1. Partimos de un nodo cualquiera (en nuestro caso siempre usaremos el primer elemento de nuestro *input*).
2. Encontramos el nodo más cercano a este nodo, y los añadimos al recorrido.
3. Repetimos el proceso hasta cubrir todos los nodos.

Hacemos uso del siguiente código:

```

def get_min_row_element(matrix, position):
    min_pos = 0
    starting_pos = 0

    while matrix[position][starting_pos] == 0 and starting_pos < len(matrix)-1:
        starting_pos += 1

    #Security check
    if starting_pos == len(matrix)-1:
        return -1

    min_pos = starting_pos

    min_val = math.inf

    for j in range(starting_pos, len(matrix[position])):
        if matrix[position][j] < min_val and position != j and matrix[position][j] != -1:
            min_pos = j
            min_val = matrix[position][j]

    return min_pos

def clean_position(matrix, pos):
    for col in range(len(matrix)):
        matrix[pos][col] = -1

```

```

    for row in range(len(matrix)):
        matrix[row][pos] = -1

def get_best_solution(points):
    road = []
    order = []

    distance_matrix = gen_distance_matrix(points, len(points))

    #We start always at first point
    last_point = 0
    road.append(points[last_point])
    order.append(0)

    while len(road) < len(points):
        best_position = get_min_row_element(distance_matrix, last_point)

        road.append(points[best_position])
        order.append(best_position)

        clean_position(distance_matrix, last_point)

        last_point = best_position

    road_distance = get_road_distance(road)

    return road, road_distance, order

file = sys.argv[1]
points = parse_input(file)
recorrido, distancia, orden = get_best_solution(points)
str_orden = [str(n) for n in orden]

print("El mejor orden a seguir: " + ', '.join(str_orden) + ")")
print("Su distancia es: " + str(distancia))

```

En este programa, básicamente tenemos lo que hacemos es partir de el primer punto pasado en el *input* y tras ello vamos buscando los nodos más cercanos sin repetir los ya añadidos. Primero parseamos el input, generamos la matriz de distancia, y la vamos transformando para descartar los nodos ya considerados.

## Análisis empírico

A continuación, mostramos los resultados obtenidos tras probar nuestro programa con los *datasets* proporcionados en la asignatura:

- **ulysses16.tsp**: El mejor orden para este *dataset* (teniendo en cuenta el orden del fichero original) es:

[0, 7, 15, 12, 11, 13, 6, 5, 14, 4, 8, 9, 3, 1, 2, 10]

y la distancia a recorrer es: 103

Si representamos los puntos con el recorrido generado obtenemos el siguiente gráfico:

- **bayg29.tsp**: El mejor orden para este *dataset* (teniendo en cuenta el orden del fichero original) es:

[0, 27, 5, 11, 8, 4, 20, 1, 19, 9, 3, 14, 17, 13, 21, 16, 10, 18, 24, 6, 22, 26, 7, 23, 15, 12, 28, 25, 2]

y la distancia a recorrer es: 10209

Si representamos los puntos con el recorrido generado obtenemos el siguiente gráfico:

- **eil76.tsp**: El mejor orden para este *dataset* (teniendo en cuenta el orden del fichero original) es:

[0, 72, 32, 62, 15, 2, 43, 31, 8, 38, 71, 57, 9, 37, 64, 10, 65, 52, 13, 18, 34, 6, 7, 45, 33, 51, 26, 44, 28,  
47, 46, 20, 73, 27, 61, 1, 29, 3, 74, 75, 66, 25, 11, 39, 15, 50, 5, 67, 4, 36, 19, 69, 59, 70, 35, 68, 60, 2  
1, 41, 40, 42, 22, 55, 48, 23, 17, 49, 24, 54, 30, 58, 53, 12, 56, 14, 63]

y la distancia a recorrer es: 642

Si representamos los puntos con el recorrido generado obtenemos el siguiente gráfico:



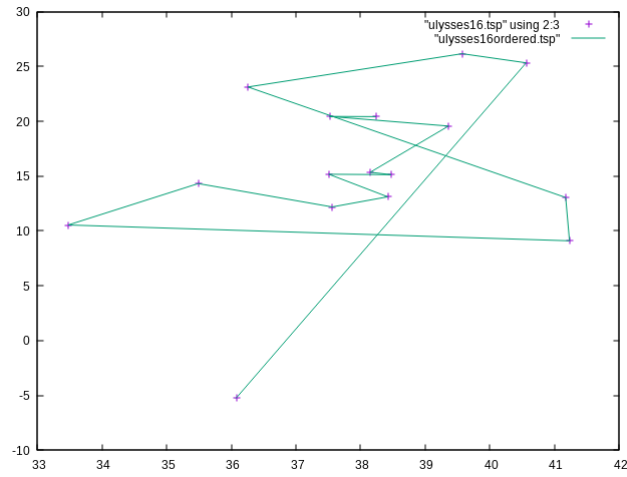


Figura 1: Gráfica de Ulysses mediante el Vecino más cercano

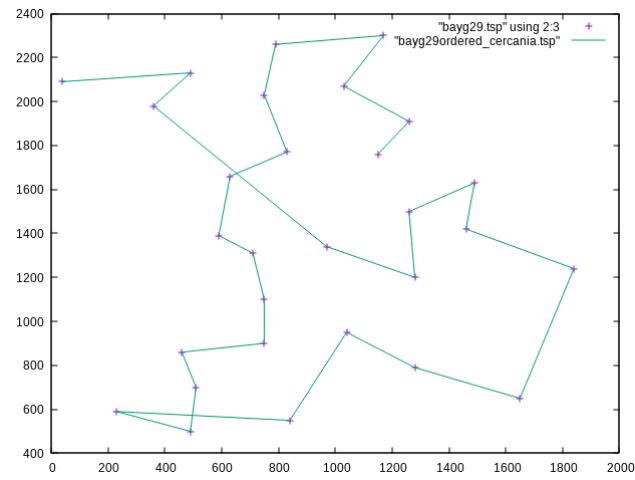


Figura 2: Gráfica de Bayg mediante el Vecino más cercano

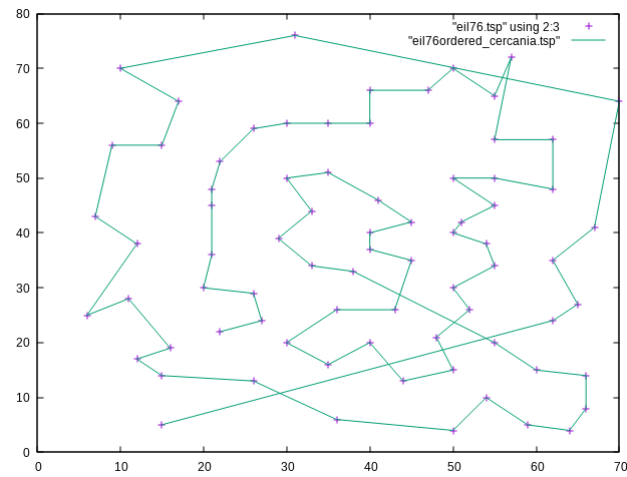


Figura 3: Gráfica de Eil mediante el Vecino más cercano

### 2.2.2. Heurística por inserción

### 2.2.3. Heurística propia: perturbaciones

Este enfoque, de nuevo *greedy*, realiza las perturbaciones indicadas por un parámetro sobre un recorrido dado para intentar mejorarlo. Tenemos el siguiente código:

```
#Find the most separated nodes (only one of them)
```

```

def get_worst_node(road):
    worst_distance = distance(road[0], road[1])
    worst_pos = 0

    for i in range(len(road)-1):
        if distance(road[i], road[i+1]) > worst_distance:
            worst_distance = distance(road[i], road[i+1])
            worst_pos = i

    return worst_pos

#Perturbate the road from a pos
def perturbate(road, orden, pos):
    current_perb = road
    best_gain = math.inf
    best_perturbation = pos
    base_distance = get_road_distance(road)

    #Calcualte all perturbations
    for i in range(len(road)):
        current_perb = get_swap(road, pos, i)
        swap_distance = get_road_distance(current_perb)

        if swap_distance < base_distance:
            best_perturbation = i
            base_distance = swap_distance

    #Make the change with the best perturbation
    road = get_swap(road, pos, best_perturbation)
    orden = get_swap(orden, pos, best_perturbation)

def get_best_solution_perturbations(points, orden, perturbations):
    base_road = points

    for i in range(perturbations):
        pos = get_worst_node(points)

        perturbate(base_road, orden, pos)

    return base_road, get_road_distance(base_road), orden

```

Primero lo que hacemos es obtener una solución mediante la heurística del vecino más cercano. Tras esto, tenemos la función `get_worst_node` que dado un recorrido, haya el nodo que más dista de su nodo siguiente. Por ello es necesario usar ya una solución previa (mediante el vecino más cercano) para calcular este “nodo peor”. Por último, tenemos la función `perturbate` que prueba todas las posibles permutaciones que podemos hacer desde el respectivo “peor nodo” y se queda con la mejor de ellas.

## Análisis empírico

A continuación, mostramos los resultados obtenidos tras probar nuestro programa con los *datasets* proporcionados en la asignatura:

- **ulysses16.tsp**: Aplicando 10 perturbaciones, obtenemos que el mejor orden (teniendo en cuenta el orden del fichero original):

[0, 7, 15, 12, 11, 13, 6, 5, 14, 4, 8, 9, 2, 1, 3, 10]

y su distancia es 101.

Si representamos los puntos con el recorrido generado para 10 perturbaciones obtenemos el siguiente gráfico:

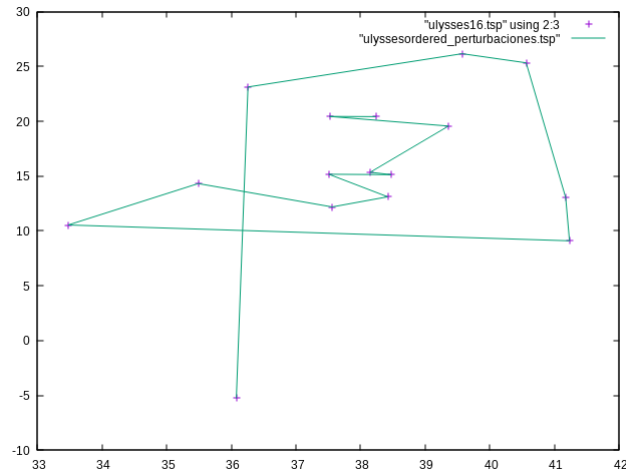


Figura 4: Gráfica de Ulysses mediante perturbaciones

- **bayg29.tsp**: Aplicando 10 perturbaciones, obtenemos que el mejor orden (teniendo en cuenta el orden del fichero original):

[0, 27, 5, 11, 8, 4, 20, 1, 19, 9, 3, 14, 17, 13, 21, 16, 10, 18, 24, 6, 22, 26, 7, 23, 15, 12, 28, 25, 2]

y su distancia es 10209.

Si representamos los puntos con el recorrido generado para 10 perturbaciones obtenemos el siguiente gráfico:

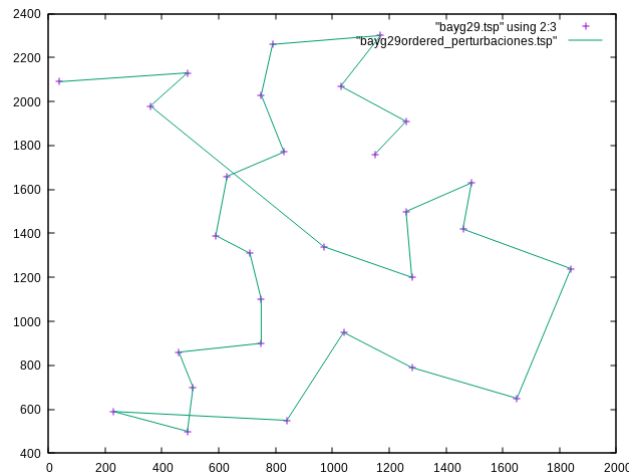


Figura 5: Gráfica de Bayg mediante perturbaciones

- **eil176.tsp**: Aplicando 10 perturbaciones, obtenemos que el mejor orden (teniendo en cuenta el orden del fichero original):

[0, 72, 32, 62, 15, 2, 43, 31, 8, 38, 71, 57, 9, 37, 64, 10, 65, 52, 13, 18, 34, 6, 7, 45, 33, 51, 26, 44, 28,  
47, 46, 20, 73, 27, 61, 1, 29, 3, 74, 75, 66, 25, 11, 39, 16, 50, 5, 67, 4, 36, 19, 69, 59, 70, 35, 68, 60, 2,  
1, 41, 40, 42, 22, 55, 48, 23, 17, 49, 24, 54, 30, 58, 53, 12, 56, 14, 63]

y su distancia es 642.

Si representamos los puntos con el recorrido generado para 10 perturbaciones obtenemos el siguiente gráfico:

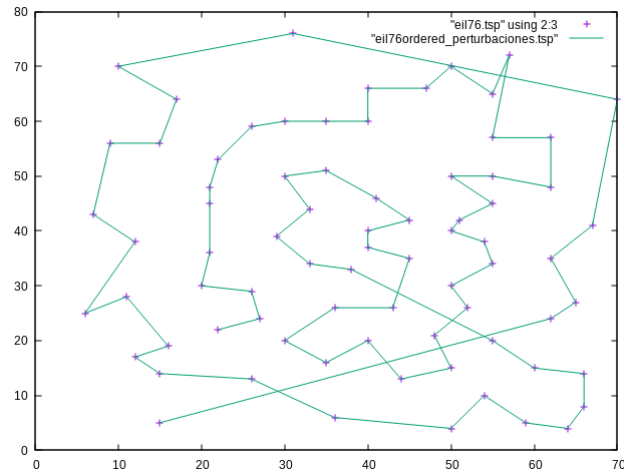


Figura 6: Gráfica de Eil mediante perturbaciones

#### 2.2.4. Comparación de heurísticas

Los tres algoritmos que tenemos son: cercanía (el vecino más cercano), inserción y por perturbaciones. Vamos a comprobar por separado las gráficas de las distancias y los tiempos de estos algoritmos:

### 3. Conclusiones

Con esta práctica, hemos aprendido a crear algoritmos voraces para resolver problemas que, en su versión obvia o de fuerza bruta tienen una complejidad en ocasiones demasiado elevada.

Además, hemos comprobado como dentro de esta misma técnica podemos atacar los problemas de manera distinta. Debemos de prestar atención a la dificultad de obtener la solución óptima, y primar el “acercarnos” a ella mediante algoritmos que van mejorando la solución anterior.

Esto es algo de especial relevancia a la hora de trabajar con cantidades ingentes de datos, en el que dar una solución inicial puede ser muy complicado, pudiendo partir incluso de soluciones arbitrarias, mejorándolas conforme nuestros algoritmos se van ejecutando.